arXiv:submit/3364829 [cs.AI] 11 Sep 2020

Applications of Deep Neural Networks

Jeff Heaton

Fall 2020.0 (draft)

Publisher: Heaton Research, Inc.
Applications of Deep Neural Networks
September, 2020
Author: Jeff Heaton (https://orcid.org/0000-0003-1496-4049)
ISBN: not assigned yet
Edition: 1

The text and illustrations of Applications of Deep Neural Networks by Jeff Heaton are licensed under CC BY-NC-SA 4.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0 All of the book's source code is licensed under the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. https://www.gnu.org/licenses/lgpl-3.0.en.html

Heaton Research, Encog, the Encog Logo and the Heaton Research logo are all trademarks of Jeff Heaton, in the United States and/or other countries.

TRADEMARKS: Heaton Research has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, so the content is based upon the final release of software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

SOFTWARE LICENSE AGREEMENT: TERMS AND CONDITIONS

Web: www.heatonresearch.com
E-Mail: support@heatonresearch.com

DISCLAIMER

Heaton Research, Inc. makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will Heaton Research, Inc., its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, Heaton Research, Inc. further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by Heaton Research, Inc. reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

This book is dedicated to my own neurons,
without whose support
this work would not have been possible.

Contents

0.1 Introduction xiv

1 Python Preliminaries **1**

1.1 Part 1.1: Course Overview 1
1.1.1 Assignments 1
1.1.2 Your Instructor: Jeff Heaton 2
1.1.3 Course Resources 3
1.1.4 What is Deep Learning 3
1.1.5 What is Machine Learning 3
1.1.6 Regression 4
1.1.7 Classification 4
1.1.8 Beyond Classification and Regression 4
1.1.9 What are Neural Networks 5
1.1.10 Why Deep Learning? 5
1.1.11 Python for Deep Learning 6
1.1.12 Software Installation 7
1.1.13 Python Introduction 7
1.1.14 Jupyter Notebooks 7
1.1.15 Python Versions 7
1.1.16 Module 1 Assignment 8
1.2 Part 1.2: Introduction to Python 8
1.3 Part 1.3: Python Lists, Dictionaries, Sets and JSON 14
1.3.1 Lists and Tuples 15
1.3.2 Sets 17
1.3.3 Maps/Dictionaries/Hash Tables 18
1.3.4 More Advanced Lists 20
1.3.5 An Introduction to JSON 23
1.4 Part 1.4: File Handling 25
1.4.1 Read a CSV File 26
1.4.2 Read (stream) a Large CSV File 27
1.4.3 Read a Text File 27
1.4.4 Read an Image 28
1.5 Part 1.5: Functions, Lambdas, and Map/Reduce 29
1.5.1 Map 30
1.5.2 Filter 31
1.5.3 Lambda 31
1.5.4 Reduce 31

2 Python for Machine Learning **33**
2.1 Part 2.1: Introduction to Pandas . . . 33
2.1.1 Missing Values . . . 35
2.1.2 Dealing with Outliers . . . 36
2.1.3 Dropping Fields . . . 37
2.1.4 Concatenating Rows and Columns . . . 38
2.1.5 Training and Validation . . . 39
2.1.6 Converting a Dataframe to a Matrix . . . 40
2.1.7 Saving a Dataframe to CSV . . . 41
2.1.8 Saving a Dataframe to Pickle . . . 42
2.1.9 Module 2 Assignment . . . 43
2.2 Part 2.2: Categorical and Continuous Values . . . 43
2.2.1 Encoding Continuous Values . . . 44
2.2.2 Encoding Categorical Values as Dummies . . . 45
2.2.3 Target Encoding for Categoricals . . . 47
2.2.4 Encoding Categorical Values as Ordinal . . . 50
2.3 Part 2.3: Grouping, Sorting, and Shuffling . . . 51
2.3.1 Shuffling a Dataset . . . 51
2.3.2 Sorting a Data Set . . . 52
2.3.3 Grouping a Data Set . . . 53
2.4 Part 2.4: Apply and Map . . . 55
2.4.1 Using Map with Dataframes . . . 55
2.4.2 Using Apply with Dataframes . . . 56
2.4.3 Feature Engineering with Apply and Map . . . 57
2.5 Part 2.5: Feature Engineering . . . 61
2.5.1 Calculated Fields . . . 61
2.5.2 Google API Keys . . . 62
2.5.3 Other Examples: Dealing with Addresses . . . 62

3 Introduction to TensorFlow **67**
3.1 Part 3.1: Deep Learning and Neural Network Introduction . . . 67
3.1.1 Classification or Regression . . . 68
3.1.2 Neurons and Layers . . . 68
3.1.3 Types of Neurons . . . 71
3.1.4 Input and Output Neurons . . . 72
3.1.5 Hidden Neurons . . . 73
3.1.6 Bias Neurons . . . 73
3.1.7 Context Neurons . . . 73
3.1.8 Other Neuron Types . . . 74
3.1.9 Why are Bias Neurons Needed? . . . 75
3.1.10 Modern Activation Functions . . . 76
3.1.11 Linear Activation Function . . . 77
3.1.12 Rectified Linear Units (ReLU) . . . 77
3.1.13 Softmax Activation Function . . . 78
3.1.14 Classic Activation Functions . . . 80
3.1.15 Step Activation Function . . . 80
3.1.16 Sigmoid Activation Function . . . 80
3.1.17 Hyperbolic Tangent Activation Function . . . 81
3.1.18 Why ReLU? . . . 81

3.1.19 Module 3 Assignment . . . 82
3.2 Part 3.2: Introduction to Tensorflow and Keras . . . 82
3.2.1 Why TensorFlow . . . 83
3.2.2 Deep Learning Tools . . . 83
3.2.3 Using TensorFlow Directly . . . 84
3.2.4 TensorFlow Linear Algebra Examples . . . 84
3.2.5 TensorFlow Mandelbrot Set Example . . . 86
3.2.6 Introduction to Keras . . . 88
3.2.7 Simple TensorFlow Regression: MPG . . . 88
3.2.8 Introduction to Neural Network Hyperparameters . . . 89
3.2.9 Controlling the Amount of Output . . . 90
3.2.10 Regression Prediction . . . 90
3.2.11 Simple TensorFlow Classification: Iris . . . 91
3.3 Part 3.3: Saving and Loading a Keras Neural Network . . . 97
3.4 Part 3.4: Early Stopping in Keras to Prevent Overfitting . . . 99
3.4.1 Early Stopping with Classification . . . 100
3.4.2 Early Stopping with Regression . . . 102
3.5 Part 3.5: Extracting Weights and Manual Network Calculation . . . 104
3.5.1 Weight Initialization . . . 104
3.5.2 Manual Neural Network Calculation . . . 105

4 Training for Tabular Data 111
4.1 Part 4.1: Encoding a Feature Vector for Keras Deep Learning . . . 111
4.1.1 Generate X and Y for a Classification Neural Network . . . 115
4.1.2 Generate X and Y for a Regression Neural Network . . . 116
4.1.3 Module 4 Assignment . . . 116
4.2 Part 4.2: Multiclass Classification with ROC and AUC . . . 116
4.2.1 Binary Classification and ROC Charts . . . 116
4.2.2 ROC Chart Example . . . 120
4.2.3 Multiclass Classification Error Metrics . . . 122
4.2.4 Calculate Classification Accuracy . . . 124
4.2.5 Calculate Classification Log Loss . . . 125
4.3 Part 4.3: Keras Regression for Deep Neural Networks with RMSE . . . 128
4.3.1 Mean Square Error . . . 130
4.3.2 Root Mean Square Error . . . 131
4.3.3 Lift Chart . . . 131
4.4 Part 4.4: Training Neural Networks . . . 132
4.4.1 Classic Backpropagation . . . 132
4.4.2 Momentum Backpropagation . . . 134
4.4.3 Batch and Online Backpropagation . . . 135
4.4.4 Stochastic Gradient Descent . . . 135
4.4.5 Other Techniques . . . 136
4.4.6 ADAM Update . . . 136
4.4.7 Methods Compared . . . 137
4.4.8 Specifying the Update Rule in Tensorflow . . . 137
4.5 Part 4.5: Error Calculation from Scratch . . . 139
4.5.1 Regression . . . 139
4.5.2 Classification . . . 140

5 Regularization and Dropout 143
5.1 Part 5.1: Introduction to Regularization: Ridge and Lasso . . . 143
5.1.1 L1 and L2 Regularization . . . 143
5.1.2 Linear Regression . . . 144
5.1.3 L1 (Lasso) Regularization . . . 146
5.1.4 L2 (Ridge) Regularization . . . 149
5.1.5 ElasticNet Regularization . . . 150
5.2 Part 5.2: Using K-Fold Cross-validation with Keras . . . 152
5.2.1 Regression vs Classification K-Fold Cross-Validation . . . 152
5.2.2 Out-of-Sample Regression Predictions with K-Fold Cross-Validation . . . 152
5.2.3 Classification with Stratified K-Fold Cross-Validation . . . 155
5.2.4 Training with both a Cross-Validation and a Holdout Set . . . 158
5.3 Part 5.3: L1 and L2 Regularization to Decrease Overfitting . . . 160
5.4 Part 5.4: Drop Out for Keras to Decrease Overfitting . . . 163
5.5 Part 5.5: Benchmarking Regularization Techniques . . . 167
5.5.1 Additional Reading on Hyperparameter Tuning . . . 168
5.5.2 Bootstrapping for Regression . . . 168
5.5.3 Bootstrapping for Classification . . . 171
5.5.4 Benchmarking . . . 174

6 Convolutional Neural Networks (CNN) for Computer Vision 181
6.1 Part 6.1: Image Processing in Python . . . 181
6.1.1 Creating Images (from pixels) in Python . . . 182
6.1.2 Transform Images in Python (at the pixel level) . . . 183
6.1.3 Standardize Images . . . 185
6.1.4 Adding Noise to an Image . . . 188
6.1.5 Module 6 Assignment . . . 189
6.2 Part 6.2: Keras Neural Networks for Digits and Fashion MNIST . . . 189
6.2.1 Computer Vision . . . 189
6.2.2 Computer Vision Data Sets . . . 189
6.2.3 MNIST Digits Data Set . . . 190
6.2.4 MNIST Fashion Data Set . . . 190
6.2.5 CIFAR Data Set . . . 190
6.2.6 Other Resources . . . 190
6.2.7 Convolutional Neural Networks (CNNs) . . . 191
6.2.8 Convolution Layers . . . 193
6.2.9 Max Pooling Layers . . . 195
6.2.10 TensorFlow with CNNs . . . 195
6.2.11 Access to Data Sets - DIGITS . . . 196
6.2.12 Display the Digits . . . 196
6.2.13 Training/Fitting CNN - DIGITS . . . 200
6.2.14 Evaluate Accuracy - DIGITS . . . 201
6.2.15 MNIST Fashion . . . 202
6.2.16 Display the Apparel . . . 203
6.2.17 Training/Fitting CNN - Fashion . . . 206
6.3 Part 6.3: Implementing a ResNet in Keras . . . 208
6.3.1 Keras Sequence vs Functional Model API . . . 209
6.3.2 CIFAR Dataset . . . 209
6.3.3 ResNet V1 . . . 213

6.3.4 ResNet V2 . . . 214
6.4 Part 6.4: Using Your Own Images with Keras . . . 221
6.5 Part 6.5: Recognizing Multiple Images with Darknet . . . 225
6.5.1 How Does DarkNet/YOLO Work? . . . 226
6.5.2 Using YOLO in Python . . . 226
6.5.3 Installing YoloV3-TF2 . . . 226
6.5.4 Transfering Weights . . . 228
6.5.5 Running DarkFlow (YOLO) . . . 230
6.5.6 Module 6 Assignment . . . 233

7 Generative Adversarial Networks 237
7.1 Part 7.1: Introduction to GANS for Image and Data Generation . . . 237
7.2 Part 7.2: Implementing DCGANs in Keras . . . 237
7.3 Part 7.3: Face Generation with StyleGAN and Python . . . 250
7.3.1 Keras Sequence vs Functional Model API . . . 251
7.3.2 Generating High Rez GAN Faces with Google CoLab . . . 252
7.3.3 Run StyleGan2 From Command Line . . . 253
7.3.4 Run StyleGAN2 From Python Code . . . 255
7.3.5 Examining the Latent Vector . . . 257
7.4 Part 7.4: GANS for Semi-Supervised Training in Keras . . . 259
7.4.1 Semi-Supervised Classification Training . . . 259
7.4.2 Semi-Supervised Regression Training . . . 260
7.4.3 Application of Semi-Supervised Regression . . . 260
7.5 Part 7.5: An Overview of GAN Research . . . 261
7.5.1 Select Projects . . . 261

8 Kaggle Data Sets 263
8.1 Part 8.1: Introduction to Kaggle . . . 263
8.1.1 Kaggle Ranks . . . 263
8.1.2 Typical Kaggle Competition . . . 263
8.1.3 How Kaggle Competition Scoring . . . 263
8.1.4 Preparing a Kaggle Submission . . . 264
8.1.5 Select Kaggle Competitions . . . 265
8.1.6 Module 8 Assignment . . . 265
8.2 Part 8.2: Building Ensembles with Scikit-Learn and Keras . . . 265
8.2.1 Evaluating Feature Importance . . . 265
8.2.2 Classification and Input Perturbation Ranking . . . 266
8.2.3 Regression and Input Perturbation Ranking . . . 269
8.2.4 Biological Response with Neural Network . . . 271
8.2.5 What Features/Columns are Important . . . 273
8.2.6 Neural Network Ensemble . . . 274
8.3 Part 8.3: Architecting Network: Hyperparameters . . . 277
8.3.1 Number of Hidden Layers and Neuron Counts . . . 278
8.3.2 Activation Functions . . . 278
8.3.3 Advanced Activation Functions . . . 279
8.3.4 Regularization: L1, L2, Dropout . . . 279
8.3.5 Batch Normalization . . . 279
8.3.6 Training Parameters . . . 279
8.3.7 Experimenting with Hyperparameters . . . 280

8.4 Part 8.4: Bayesian Hyperparameter Optimization for Keras . . . 282
8.5 Part 8.5: Current Semester's Kaggle . . . 288
8.5.1 Iris as a Kaggle Competition . . . 289
8.5.2 MPG as a Kaggle Competition (Regression) . . . 292
8.5.3 Module 8 Assignment . . . 295

9 Transfer Learning **297**
9.1 Part 9.1: Introduction to Keras Transfer Learning . . . 297
9.1.1 Transfer Learning Example . . . 297
9.1.2 Module 9 Assignment . . . 304
9.2 Part 9.2: Popular Pretrained Neural Networks for Keras . . . 304
9.2.1 DenseNet . . . 304
9.2.2 InceptionResNetV2 and InceptionV3 . . . 304
9.2.3 MobileNet . . . 305
9.2.4 MobileNetV2 . . . 305
9.2.5 NASNet . . . 305
9.2.6 ResNet, ResNetV2, ResNeXt . . . 305
9.2.7 VGG16 and VGG19 . . . 306
9.2.8 Xception . . . 306
9.3 Part 9.3: Transfer Learning for Computer Vision and Keras . . . 306
9.3.1 Transfer . . . 310
9.4 Part 9.4: Transfer Learning for Languages and Keras . . . 316
9.5 Part 9.5: Transfer Learning for Keras Feature Engineering . . . 323

10 Time Series in Keras **327**
10.1 Part 10.1: Time Series Data Encoding . . . 327
10.1.1 Module 10 Assignment . . . 331
10.2 Part 10.2: Programming LSTM with Keras and TensorFlow . . . 331
10.2.1 Understanding LSTM . . . 333
10.2.2 Simple TensorFlow LSTM Example . . . 335
10.2.3 Sun Spots Example . . . 338
10.2.4 Further Reading for LSTM . . . 342
10.3 Part 10.3: Text Generation with LSTM . . . 343
10.3.1 Additional Information . . . 343
10.3.2 Character-Level Text Generation . . . 343
10.4 Part 10.4: Image Captioning with Keras and TensorFlow . . . 350
10.4.1 Needed Data . . . 352
10.4.2 Running Locally . . . 352
10.4.3 Clean/Build Dataset From Flickr8k . . . 352
10.4.4 Choosing a Computer Vision Neural Network to Transfer . . . 355
10.4.5 Creating the Training Set . . . 357
10.4.6 Using a Data Generator . . . 359
10.4.7 Loading Glove Embeddings . . . 361
10.4.8 Building the Neural Network . . . 361
10.4.9 Train the Neural Network . . . 363
10.4.10 Generating Captions . . . 364
10.4.11 Evaluate Performance on Test Data from Flicker8k . . . 364
10.4.12 Evaluate Performance on My Photos . . . 366
10.4.13 Module 10 Assignment . . . 368

10.5 Part 10.5: Temporal CNN in Keras and TensorFlow . . . 368
10.5.1 Sun Spots Example - CNN . . . 370

11 Natural Language Processing and Speech Recognition 375
11.1 Part 11.1: Getting Started with Spacy in Python . . . 375
11.1.1 Installing Spacy . . . 375
11.1.2 Tokenization . . . 376
11.1.3 Sentence Diagramming . . . 377
11.1.4 Stop Words . . . 379
11.2 Part 11.2: Word2Vec and Text Classification . . . 380
11.2.1 Suggested Software for Word2Vec . . . 380
11.3 Part 11.3: What are Embedding Layers in Keras . . . 383
11.3.1 Simple Embedding Layer Example . . . 383
11.3.2 Transferring An Embedding . . . 386
11.3.3 Training an Embedding . . . 387
11.4 Part 11.4: Natural Language Processing with Spacy and Keras . . . 391
11.4.1 Word-Level Text Generation . . . 391
11.5 Part 11.5: Learning English from Scratch with Keras and TensorFlow . . . 399
11.5.1 Imports and Utility Functions . . . 399
11.5.2 Getting the Data . . . 401
11.5.3 Building the Vocabulary . . . 403
11.5.4 Building the Training and Test Data . . . 404
11.5.5 Compile the Neural Network . . . 406
11.5.6 Train the Neural Network . . . 407
11.5.7 Evaluate Accuracy . . . 409
11.5.8 Adhoc Query . . . 410

12 Reinforcement Learning 413
12.1 Part 12.1: Introduction to the OpenAI Gym . . . 413
12.1.1 OpenAI Gym Leaderboard . . . 413
12.1.2 Looking at Gym Environments . . . 413
12.1.3 Render OpenAI Gym Environments from CoLab . . . 416
12.2 Part 12.2: Introduction to Q-Learning . . . 418
12.2.1 Introducing the Mountain Car . . . 419
12.2.2 Programmed Car . . . 422
12.2.3 Reinforcement Learning . . . 424
12.2.4 Running and Observing the Agent . . . 428
12.2.5 Inspecting the Q-Table . . . 429
12.3 Part 12.3: Keras Q-Learning in the OpenAI Gym . . . 430
12.3.1 DQN and the Cart-Pole Problem . . . 430
12.3.2 Hyperparameters . . . 432
12.3.3 Environment . . . 433
12.3.4 Agent . . . 436
12.3.5 Policies . . . 437
12.3.6 Metrics and Evaluation . . . 438
12.3.7 Replay Buffer . . . 439
12.3.8 Data Collection . . . 440
12.3.9 Training the agent . . . 441
12.3.10 Visualization . . . 442

12.3.11 Plots . . . 442
12.3.12 Videos . . . 443
12.4 Part 12.4: Atari Games with Keras Neural Networks . . . 444
12.4.1 Actual Atari 2600 Specs . . . 444
12.4.2 OpenAI Lab Atari Pong . . . 445
12.4.3 Hyperparameters . . . 446
12.4.4 Atari Environment's . . . 447
12.4.5 Agent . . . 448
12.4.6 Metrics and Evaluation . . . 450
12.4.7 Replay Buffer . . . 451
12.4.8 Random Collection . . . 451
12.4.9 Training the agent . . . 452
12.4.10 Visualization . . . 453
12.4.11 Videos . . . 454
12.5 Part 12.5: Application of Reinforcement Learning . . . 455
12.5.1 Create an Environment of your Own . . . 456
12.5.2 Testing the Environment . . . 464
12.5.3 Hyperparameters . . . 466
12.5.4 Instanciate the Environment . . . 466
12.5.5 Metrics and Evaluation . . . 469
12.5.6 Data Collection . . . 470
12.5.7 Training the agent . . . 471
12.5.8 Visualization . . . 472
12.5.9 Plots . . . 472
12.5.10 Videos . . . 473

13 Advanced/Other Topics 475
13.1 Part 13.1: Flask and Deep Learning Web Services . . . 475
13.1.1 Flask Hello World . . . 475
13.1.2 MPG Flask . . . 476
13.1.3 Flask MPG Client . . . 480
13.1.4 Images and Web Services . . . 481
13.2 Part 13.2: Deploying a Model to AWS . . . 483
13.2.1 Train Model (optionally, outside of AWS) . . . 483
13.2.2 Next Step: Convert Model (must use AWS SageMaker Notebook) . . . 484
13.2.3 Step 1. Set up . . . 484
13.2.4 Step 2. Load the Keras model using the JSON and weights file . . . 485
13.2.5 Step 3. Export the Keras model to the TensorFlow ProtoBuf format (must use AWS SageMaker Notebook) . . . 486
13.2.6 Step 4. Convert TensorFlow model to a SageMaker readable format (must use AWS SageMaker Notebook) . . . 487
13.2.7 Tar the entire directory and upload to S3 . . . 487
13.2.8 Step 5. Deploy the trained model (must use AWS SageMaker Notebook) . . . 488
13.2.9 Part 3: Test Model Deployment (optionally, outside of AWS) . . . 489
13.2.10 Call the end point . . . 489
13.2.11 Additional Reading . . . 490
13.3 Part 13.3: Using a Keras Deep Neural Network with a Web Application . . . 490
13.4 Part 13.4: When to Retrain Your Neural Network . . . 491
13.4.1 Preprocessing the Sberbank Russian Housing Market Data . . . 493

13.4.2 KS-Statistic . . . 494
13.4.3 Detecting Drift between Training and Testing Datasets by Training . . . 496
13.5 Part 13.5: Using a Keras Deep Neural Network with a Web Application . . . 497
13.5.1 Converting Keras to CoreML . . . 498
13.5.2 Creating an IOS CoreML Application . . . 500
13.5.3 More Reading . . . 500

14 Other Neural Network Techniques 503
14.1 Part 14.1: What is AutoML . . . 503
14.1.1 AutoML from your Local Computer . . . 503
14.1.2 AutoML from Google Cloud . . . 503
14.1.3 A Simple AutoML System . . . 503
14.1.4 Running My Sample AutoML Program . . . 514
14.2 Part 14.2: Using Denoising AutoEncoders in Keras . . . 516
14.2.1 Function Approximation . . . 516
14.2.2 Multi-Output Regression . . . 518
14.2.3 Simple Autoencoder . . . 520
14.2.4 Autoencode (single image) . . . 521
14.2.5 Standardize Images . . . 522
14.2.6 Image Autoencoder (multi-image) . . . 525
14.2.7 Adding Noise to an Image . . . 527
14.2.8 Denoising Autoencoder . . . 529
14.3 Part 14.3: Anomaly Detection in Keras . . . 538
14.3.1 Read in KDD99 Data Set . . . 538
14.3.2 Preprocessing . . . 540
14.3.3 Training the Autoencoder . . . 543
14.3.4 Detecting an Anomaly . . . 545
14.4 Part 14.4: Training an Intrusion Detection System with KDD99 . . . 545
14.4.1 Read in Raw KDD-99 Dataset . . . 545
14.4.2 Analyzing a Dataset . . . 547
14.4.3 Encode the feature vector . . . 549
14.4.4 Train the Neural Network . . . 551
14.5 Part 14.5: New Technologies . . . 553
14.5.1 Neural Structured Learning (NSL) . . . 553
14.5.2 Bert, AlBert, and Other NLP Technologies . . . 554
14.5.3 Explainability Frameworks . . . 557

0.1 Introduction

Deep learning is a group of exciting new technologies for neural networks. Through a combination of advanced training techniques and neural network architectural components, it is now possible to create neural networks that can handle tabular data, images, text, and audio as both input and output. Deep learning allows a neural network to learn hierarchies of information in a way that is like the function of the human brain. This course will introduce the student to classic neural network structures, Convolution Neural Networks (CNN), Long Short-Term Memory (LSTM), Gated Recurrent Neural Networks (GRU), General Adversarial Networks (GAN) and reinforcement learning. Application of these architectures to computer vision, time series, security, natural language processing (NLP), and data generation will be covered. High Performance Computing (HPC) aspects will demonstrate how deep learning can be leveraged both on graphical processing units (GPUs), as well as grids. Focus is primarily upon the application of deep learning to problems, with some introduction to mathematical foundations. Students will use the Python programming language to implement deep learning using Google TensorFlow and Keras. It is not necessary to know Python prior to this course; however, familiarity of at least one programming language is assumed. This course will be delivered in a hybrid format that includes both classroom and online instruction.

Chapter 1

Python Preliminaries

1.1 Part 1.1: Course Overview

Deep learning is a group of exciting new technologies for neural networks.[25]By using a combination of advanced training techniques neural network architectural components, it is now possible to train neural networks of much greater complexity. This course introduces the student to deep belief neural networks, regularization units (ReLU), convolution neural networks, and recurrent neural networks. High-performance computing (HPC) aspects demonstrate how deep learning can be leveraged both on graphical processing units (GPUs), as well as grids. Deep learning allows a model to learn hierarchies of information in a way that is similar to the function of the human brain. The focus is primarily upon the application of deep learning, with some introduction to the mathematical foundations of deep learning. Students make use of the Python programming language to architect a deep learning model for several real-world data sets and interpret the results of these networks.[11]

1.1.1 Assignments

Your grade is calculated according to the following assignments:

Assignment	Weight	Description
Ice Breaker	5%	Post a short get to know you discussion topic (individual)
Group Selection	5%	Choose groups for Kaggle final projects
Class Assignments	50%	10 small programming assignments (5% each, individual)
Kaggle Project	20%	"Kaggle In-Class" competition (team)
Final Project	20%	Deep Learning Implementation Report (team)

The 10 class assignments correspond with each of the first 10 modules. Generally, each module assignment is due just before the following module date. Refer to the syllabus for exact due dates. The 10 class assignments are submitted using the Python submission script. Refer to assignment 1 for details.

The Kaggle and Final Projects are completed in teams. The same teams will complete each of these.

- **Module 1 Assignment**: How to Submit an Assignment
- **Module 2 Assignment**: Creating Columns in Pandas
- **Module 3 Assignment**: Data Preparation in Pandas
- **Module 4 Assignment**: Classification and Regression Neural Networks
- **Module 5 Assignment**: Predict Home Price
- **Module 6 Assignment**: Image Processing
- **Module 7 Assignment**: Computer Vision

- **Module 8 Assignment**: Feature Engineering
- **Module 9 Assignment**: Transfer Learning
- **Module 10 Assignment**: Time Series Neural Network

1.1.2 Your Instructor: Jeff Heaton

Jeff Heaton, pictured in Figure 1.1 is the author of this book and developer of this course. A brief summary of his credentials is given here:

- Master of Information Management (MIM), Washington University in St. Louis, MO
- PhD in Computer Science, Nova Southeastern University in Ft. Lauderdale, FL
- Vice President and Data Scientist, Reinsurance Group of America (RGA)
- Senior Member, IEEE
- jtheaton at domain name of this university
- Other industry certifications: FLMI, ARA, ACS

Social media:

- Homepage - My home page. Includes my research interests and publications.
- YouTube Channel - My YouTube Channel. Subscribe for my videos on AI and updates to this class.
- Discord Server - To discuss this class and AI topics.
- GitHub - My GitHub repositories.
- LinkedIn - My Linked In profile.
- Twitter - My Twitter feed.
- Google Scholar - My citations on Google Scholar.
- Research Gate - My profile/research at Research Gate.
- Orcid
- Others - About me and other social media sites that I am a member of.

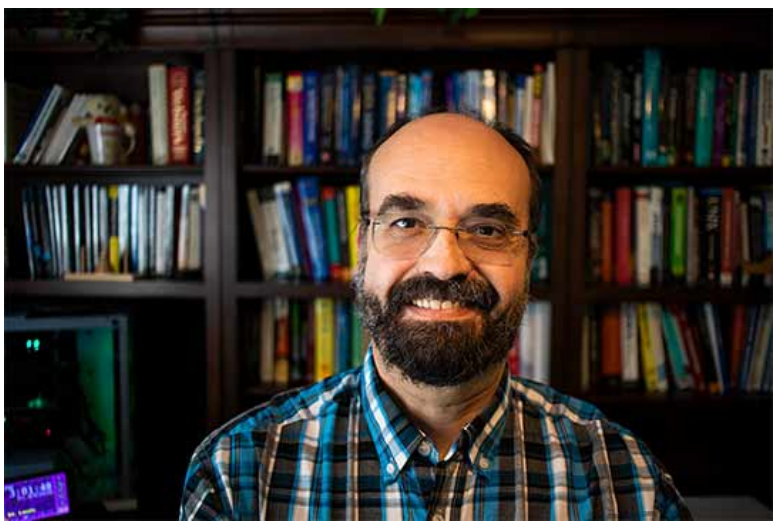


Figure 1.1: Jeff Heaton Recording a Video

My PhD dissertation explored new methods for feature engineering in deep learning.[16]I also created the Encog Machine Learning Framework in Java and C#.[13]I've also conducted research on artificial life[14].

1.1.3 Course Resources

- Google CoLab - Free web-based platform that includes Python, Juypter Notebooks, and TensorFlow[1]. No setup needed.
- Python Anaconda - Python distribution that includes many data science packages, such as Numpy, Scipy, Scikit-Learn, Pandas, and much more.
- Juypter Notebooks - Easy to use environment that combines Python, Graphics and Text.
- TensorFlow - Google's mathematics package for deep learning.
- Kaggle - Competitive data science. Good source of sample data.
- Course GitHub Repository - All of the course notebooks will be published here.

1.1.4 What is Deep Learning

The focus of this class is deep learning, which is a prevalent type of machine learning that builds upon the original neural networks popularized in the 1980s. There is very little difference between how a deep neural network is calculated compared with the first neural network. We've always been able to create and calculate deep neural networks. A deep neural network is nothing more than a neural network with many layers. While we've always been able to create/calculate deep neural networks, we've lacked an effective means of training them. Deep learning provides an efficient means to train deep neural networks.

1.1.5 What is Machine Learning

If deep learning is a type of machine learning, this begs the question, "What is machine learning?" Figure 1.2 illustrates how machine learning differs from traditional software development.

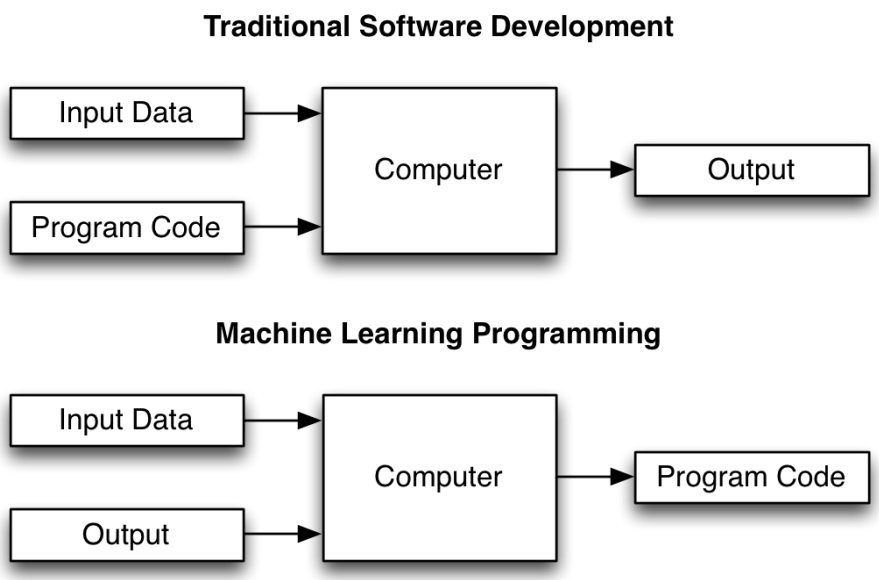


Figure 1.2: ML vs Traditional Software Development

- **Traditional Software Development** - Programmers create programs that specify how to transform input into the desired output.

- **Machine Learning** - Programmers create models that can learn to produce the desired output for given input. This learning fills the traditional role of the computer program.

Researchers have applied machine learning to many different areas. This class explores three specific domains for the application of deep neural networks, as illustrated in Figure 1.3.

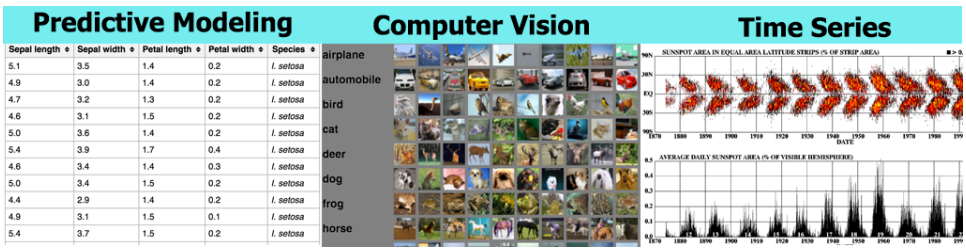


Figure 1.3: Application of Machine Learning

- **Predictive Modeling** - Several named input values allow the neural network to predict another named value that becomes the output. For example, using four measurements of iris flowers to predict the species. This type of data is often called tabular data.
- **Computer Vision** - The use of machine learning to detect patterns in visual data. For example, is an image a picture of a cat or a dog.
- **Time Series** - The use of machine learning to detect patterns in time. Typical applications of time series are financial applications, speech recognition, and even natural language processing (NLP).

1.1.6 Regression

Regression is when a model, such as a neural network, accepts input, and produces a numeric output. Consider if you must to write a program that predicted how many miles per gallon (MPG) a car could achieve. For the inputs, you would probably want such features as the weight of the car, the horsepower, how large the engine is, and other values. Your program would be a combination of math and if-statements.

Machine learning lets the computer learn the "formula" for calculating the MPG of a car, using data. Consider this dataset. We can use regression machine learning models to study this data and learn how to predict the MPG for a car.

1.1.7 Classification

The output of a classification model is what class the input most closely resembles. For example, consider using four measurements of an iris flower to determine the likely species that the flower. The iris dataset is a typical dataset for machine learning examples.

1.1.8 Beyond Classification and Regression

One of the most potent aspects of neural networks is that a neural network can be both regression and classification at the same time. The output from a neural network could be any number of the following:

- An image
- A series of numbers that could be interpreted as text, audio, or another time series
- A regression number
- A classification class

1.1.9 What are Neural Networks

Neural networks are one of the earliest examples of a machine learning model. Neural networks were initially introduced in the 1940s and have risen and fallen several times from popularity. The current generation of deep learning began in 2006 with an improved training algorithm by Geoffrey Hinton.[17]This technique finally allowed neural networks with many layers (deep neural networks) to be efficiently trained. Four researchers have contributed significantly to the development of neural networks. They have consistently pushed neural network research, both through the ups and downs. These four luminaries are shown in Figure 1.4.

Figure 1.4: Neural Network Luminaries

The current luminaries of artificial neural network (ANN) research and ultimately deep learning, in order as appearing in the above picture:

- Yann LeCun, Facebook and New York University - Optical character recognition and computer vision using convolutional neural networks (CNN). The founding father of convolutional nets.
- Geoffrey Hinton, Google and University of Toronto. Extensive work on neural networks. Creator of deep learning and early adapter/creator of backpropagation for neural networks.
- Yoshua Bengio, University of Montreal. Extensive research into deep learning, neural networks, and machine learning. He has so far remained entirely in academia.
- Andrew Ng, Badiu and Stanford University. Extensive research into deep learning, neural networks, and application to robotics.

Geoffrey Hinton, Yann LeCun, and Yoshua Bengio won the Turing Award for their contributions to deep learning.

1.1.10 Why Deep Learning?

For predictive modeling, neural networks are not that different than other models, such as:

- Support Vector Machines

- Random Forests
- Gradient Boosted Machines

Like these other models, neural networks can perform both **classification** and **regression**. When applied to relatively low-dimensional predictive modeling tasks, deep neural networks do not necessarily add significant accuracy over other model types. Andrew Ng describes the advantage of deep neural networks over traditional model types as illustrated by Figure 1.5.

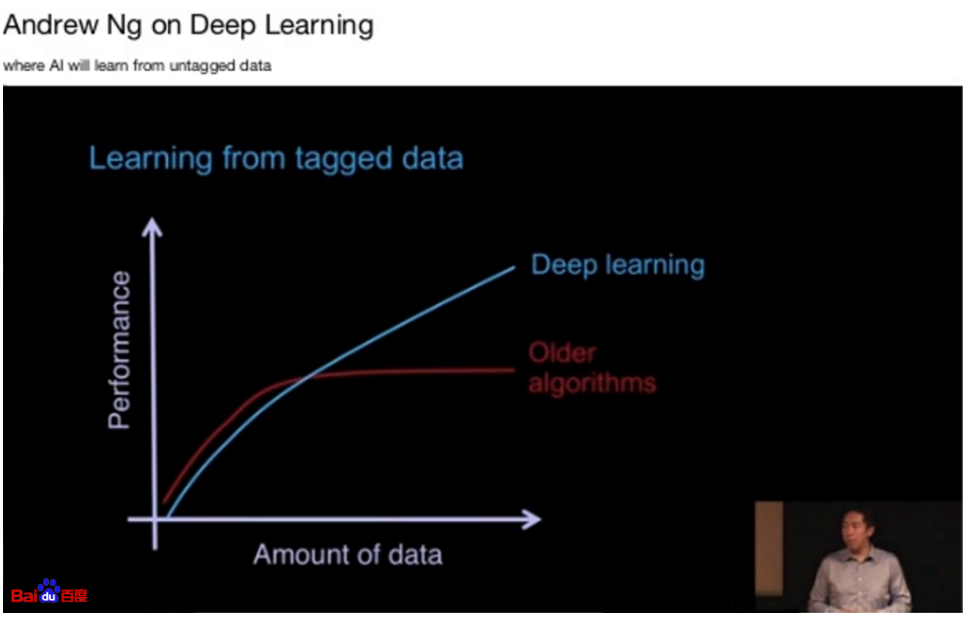


Figure 1.5: Why Deep Learning?

Neural networks also have two additional significant advantages over other machine learning models:

- **Convolutional Neural Networks** - Can scan an image for patterns within the image.
- **Recurrent Neural Networks** - Can find patterns across several inputs, not just within a single input.

Neural networks are also very flexible in the types of data that are compatible with the input and output layers. A neural network can take tabular data, images, audio sequences, time series tabular data, and text as its input or output.

1.1.11 Python for Deep Learning

Python 3.x is the programming language that will be used for this class. Python, as a programming language, has the widest support for deep learning. The three most popular frameworks for deep learning in Python are:

- TensorFlow (Google)
- MXNet (Amazon)
- CNTK (Microsoft)

Some references on popular programming languages for AI/Data Science:

- Popular Programming Languages for AI
- Popular Programming Languages for Data Science

1.1.12 Software Installation

This class is technically oriented. A successful student needs to be able to compile and execute Python code that makes use of TensorFlow for deep learning. There are two options for you to accomplish this:

- Install Python, TensorFlow and some IDE (Jupyter, TensorFlow, and others)
- Use Google CoLab in the cloud

Near the top of this document, there are links to videos that describe how to use Google CoLab. There are also videos explaining how to install Python on your local computer. The following sections take you through the process of installing Python on your local computer. This process is essentially the same on Windows, Linux, or Mac. For specific OS instructions, refer to one of the tutorial YouTube videos earlier in this document.

To install Python on your computer complete the following instructions:

- Installing Python and TensorFlow

1.1.13 Python Introduction

- Anaconda v3.6 Scientific Python Distribution, including: Scikit-Learn, Pandas, and others: csv, json, numpy, scipy
- Jupyter Notebooks
- PyCharm IDE
- Cx_Oracle
- MatPlotLib

1.1.14 Jupyter Notebooks

Space matters in Python, indent code to define blocks

Jupyter Notebooks Allow Python and Markdown to coexist.

Even LaTeX math:

$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$.

1.1.15 Python Versions

- If you see `xrange` instead of `range`, you are dealing with Python 2
- If you see `print x` instead of `print(x)`, you are dealing with Python 2
- This class uses Python 3.6!

Code

```
# What version of Python do you have?
import sys

import tensorflow.keras
import pandas as pd
import sklearn as sk
```

```
import tensorflow as tf

print(f"Tensor Flow Version: {tf.__version__}")
print(f"Keras Version: {tensorflow.keras.__version__}")
print()
print(f"Python {sys.version}")
print(f"Pandas {pd.__version__}")
print(f"Scikit-Learn {sk.__version__}")
print("GPU is", "available" if tf.test.is_gpu_available() \
      else "NOT AVAILABLE")
```

Output

```
Tensor Flow Version: 2.1.0
Keras Version: 2.2.4-tf
Python 3.7.7 (default, May  6 2020, 11:45:54) [MSC v.1916 64 bit
(AMD64)]
Pandas 1.0.5
Scikit-Learn 0.23.1
WARNING:tensorflow:From <ipython-input-2-796d1d9091ed>:15:
is_gpu_available (from tensorflow.python.framework.test_util) is
deprecated and will be removed in a future version.
Instructions for updating:
Use `tf.config.list_physical_devices('GPU')` instead.
GPU is available
```

Software used in this class:

- **Python** - The programming language.
- **TensorFlow** - Googles deep learning framework, must have the version specified above.
- **Keras** - Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.[7]
- **Pandas** - Allows for data preprocessing. Tutorial here
- **Scikit-Learn** - Machine learning framework for Python. Tutorial here.

1.1.16 Module 1 Assignment

You can find the first assignment here: assignment 1

1.2 Part 1.2: Introduction to Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python has become a common language for machine learning research and is the primary language for TensorFlow.

Python 3.0, released in 2008, was a significant revision of the language that is not entirely backward-compatible, and much Python 2 code does not run unmodified on Python 3. This course makes use of

Python 3. Furthermore, TensorFlow is not compatible with versions of Python earlier than 3. A non-profit organization, the Python Software Foundation (PSF), manages and directs resources for Python development. On January 1, 2020, the PSF discontinued the Python 2 language and no longer provides security patches and other improvements. Python interpreters are available for many operating systems.

The first two modules of this course provide an introduction to some aspects of the Python programming language. However, entire books focus on Python. Two modules will not cover every detail of this language. The reader is encouraged to consult additional sources on the Python language.

Like most tutorials, we will begin by printing Hello World.

Code

```
print("Hello World")
```

Output

```
Hello World
```

The above code passes a constant string, containing the text "hello world" to a function that is named print.

You can also leave comments in your code to explain what you are doing. Comments can begin anywhere in a line.

Code

```
# Single line comment (this has no effect on your program)
print("Hello World") # Say hello
```

Output

```
Hello World
```

Strings are very versatile and allow your program to process textual information. Constant string, enclosed in quotes, define literal string values inside your program. Sometimes you may wish to define a larger amount of literal text inside of your program. This text might consist of multiple lines. The triple quote allows for multiple lines of text.

Code

```
print("""Print
Multiple
Lines
""")
```

Output

```
Print
Multiple
Lines
```

Like many languages uses single (') and double (") quotes interchangeably to denote literal string constants. The general convention is that double quotes should enclose actual text, such as words or sentences. Single quotes should enclose symbolic text, such as error codes. An example of an error code might be 'HTTP404'.

However, there is no difference between single and double quotes in Python, and you may use whichever you like. The following code makes use of a single quote.

Code

```
print('Hello World')
```

Output

```
Hello World
```

In addition to strings, Python allows numbers as literal constants in programs. Python includes support for floating-point, integer, complex, and other types of numbers. This course will not make use of complex numbers. Unlike strings, quotes do not enclose numbers.

The presence of a decimal point differentiates floating-point and integer numbers. For example, the value 42 is an integer. Similarly, 42.5 is a floating-point number. If you wish to have a floating-point number, without a fraction part, you should specify a zero fraction. The value 42.0 is a floating-point number, although it has no fractional part. As an example, the following code prints two numbers.

Code

```
print(42)
print(42.5)
```

Output

```
42
42.5
```

So far, we have only seen how to define literal numeric and string values. These literal values are constant and do not change as your program runs. Variables allow your program to hold values that can change as the program runs. Variables have names that allow you to reference their values. The following code assigns an integer value to a variable named "a" and a string value to a variable named "b."

Code

```
a = 10
b = "ten"
print(a)
print(b)
```

Output

```
10
```

```
ten
```

The key feature of variables is that they can change. The following code demonstrates how to change the values held by variables.

Code

```
a = 10
print(a)
a = a + 1
print(a)
```

Output

```
10
11
```

You can mix strings and variables for printing. This technique is called a formatted or interpolated string. The variables must be inside of the curly braces. In Python, this type of string is generally called an f-string. The f-string is denoted by placing an "f" just in front of the opening single or double quote that begins the string. The following code demonstrates the use of an f-string to mix several variables with a literal string.

Code

```
a = 10
print(f'The␣value␣of␣a␣is␣{a}')
```

Output

```
The value of a is 10
```

You can also use f-strings with math (called an expression). Curly braces can enclose any valid Python expression for printing. The following code demonstrates the use of an expression inside of the curly braces of an f-string.

Code

```
a = 10
print(f'The␣value␣of␣a␣plus␣5␣is␣{a+5}')
```

Output

```
The value of a plus 5 is 15
```

Python has many ways to print numbers; these are all correct. However, for this course, we will use f-strings. The following code demonstrates some of the varied methods of printing numbers in Python.

Code

```
a = 5

print(f'a is {a}') # Preferred method for this course.
print('a is {}'.format(a))
print('a is ' + str(a))
print('a is %d' % (a))
```

Output

```
a is 5
a is 5
a is 5
a is 5
```

You can use if-statements to perform logic. Notice the indents? These if-statements are how Python defines blocks of code to execute together. A block usually begins after a colon and includes any lines at the same level of indent. Unlike many other programming languages, Python uses whitespace to define blocks of code. The fact that whitespace is significant to the meaning of program code is a frequent source of annoyance for new programmers of Python. Tabs and spaces are both used to define the scope in a Python program. Mixing both spaces and tabs in the same program is not recommended.

Code

```
a = 5
if a>5:
    print('The variable a is greater than 5.')
else:
    print('The variable a is not greater than 5')
```

Output

```
The variable a is not greater than 5
```

The following if-statement has multiple levels. It can be easy to indent these levels improperly, so be careful. This code contains a nested if-statement under the first "a==5" if-statement. Only if a is equal to 5 will the nested "b==6" if-statement be executed. Also, not that the "elif" command means "else if."

Code

```
a = 5
b = 6

if a==5:
    print('The variable a is 5')
    if b==6:
        print('The variable b is also 6')
```

```
elif a==6:
    print('The variable a is 6')
```

Output

```
The variable a is 5
The variable b is also 6
```

It is also important to note that the double equal ("==") operator is used to test the equality of two expressions. The single equal ("=") operator is only used to assign values to variables in Python. The greater than (">"), less than ("<"), greater than or equal (">="), less than or equal ("<=") all perform as would generally be accepted. Testing for inequality is performed with the not equal ("!=") operator.

It is common in programming languages to loop over a range of numbers. Python accomplishes this through the use of the **range** operation. Here you can see a **for** loop and a **range** operation that causes the program to loop between 1 and 9.

Code

```
for x in range(1, 10):  # If you ever see xrange, you are in Python 2
    print(x)
# If you ever see print x (no parenthesis), you are in Python 2
```

Output

```
1
2
3
4
5
6
7
8
9
```

This code illustrates some incompatibilities between Python 2 and Python 3. Before Python 3, it was acceptable to leave the parentheses off of a *print* function call. This method of invoking the *print* command is no longer allowed in Python 3. Similarly, it used to be a performance improvement to use the *xrange* command in place of range command at times. Python 3 incorporated all of the functionality of the *xrange* Python 2 command into the normal *range* command. As a result, the programmer should not use the *xrange* command in Python 3. If you see either of these constructs used in example code, then you are looking at an older Python 2 era example.

The *range* command is used in conjunction with loops to pass over a specific range of numbers. Cases, where you must loop over specific number ranges, are somewhat uncommon. Generally, programmers use loops on collections of items, rather than hard-coding numeric values into your code. Collections, as well as the operations that loops can perform on them, is covered later in this module.

The following is a further example of a looped printing of strings and numbers.

Code

```
acc = 0
for x in range(1, 10):
    acc += x
    print(f"Adding {x}, sum so far is {acc}")

print(f"Final sum: {acc}")
```

Output

```
Adding 1, sum so far is 1
Adding 2, sum so far is 3
Adding 3, sum so far is 6
Adding 4, sum so far is 10
Adding 5, sum so far is 15
Adding 6, sum so far is 21
Adding 7, sum so far is 28
Adding 8, sum so far is 36
Adding 9, sum so far is 45
Final sum: 45
```

1.3 Part 1.3: Python Lists, Dictionaries, Sets and JSON

Like most modern programming languages, Python includes Lists, Sets, Dictionaries, and other data structures as built-in types. The syntax appearance of both of these is similar to JSON. Python and JSON compatibility is discussed later in this module. This course will focus primarily on Lists, Sets, and Dictionaries. It is essential to understand the differences between these three fundamental collection types.

- **Dictionary** - A dictionary is a mutable unordered collection that Python indexes with name and value pairs.
- **List** - A list is a mutable ordered collection that allows duplicate elements.
- **Set** - A set is a mutable unordered collection with no duplicate elements.
- **Tuple** - A tuple is an immutable ordered collection that allows duplicate elements.

Most Python collections are mutable, which means that the program can add and remove elements after definition. An immutable collection cannot add or remove items after definition. It is also essential to understand that an ordered collection means that items maintain their order as the program adds them to a collection. This order might not be any specific ordering, such as alphabetic or numeric.

Lists and tuples are very similar in Python and are often confused. The significant difference is that a list is mutable, but a tuple isn't. So, we include a list when we want to contain similar items, and include a tuple when we know what information goes into it ahead of time.

Many programming languages contain a data collection called an array. The array type is noticeably absent in Python. Generally, the programmer will use a list in place of an array in Python. Arrays in most programming languages were fixed-length, requiring the program to know the maximum number of elements needed ahead of time. This restriction leads to the infamous array-overrun bugs and security issues. The Python list is much more flexible in that the program can dynamically change the size of a list.

The next sections will look at each of these collection types in more detail.

1.3.1 Lists and Tuples

For a Python program, lists and tuples are very similar. It is possible to get by as a programmer using only lists and ignoring tuples. Both lists and tuples hold an ordered collection of items.

The primary difference that you will see syntactically is that a list is enclosed by square braces [] and a list is enclosed by parenthesis (). The following code defines both list and tuple.

Code

```
l = ['a', 'b', 'c', 'd']
t = ('a', 'b', 'c', 'd')

print(l)
print(t)
```

Output

```
['a', 'b', 'c', 'd']
('a', 'b', 'c', 'd')
```

The primary difference that you will see programmatically is that a list is mutable, which means the program can change it. A tuple is immutable, which means the program cannot change it. The following code demonstrates that the program can change a list. This code also illustrates that Python indexes lists starting at element 0. Accessing element one modifies the second element in the collection. One advantage of tuples over lists is that tuples are generally slightly faster to iterate over than lists. One advantage of tuples over lists is that tuples are generally slightly faster to iterate over lists.

Code

```
l[1] = 'changed'
#t[1] = 'changed' # This would result in an error

print(l)
```

Output

```
['a', 'changed', 'c', 'd']
```

Like many languages, Python has a for-each statement. This statement allows you to loop over every element in a collection, such as a list or a tuple.

Code

```
# Iterate over a collection.
for s in l:
    print(s)
```

Output

```
a
changed
c
d
```

The **enumerate** function is useful for enumerating over a collection and having access to the index of the element that we are currently on.

Code

```
# Iterate over a collection, and know where your index.
(Python is zero-based!)
for i,l in enumerate(l):
    print(f"{i}:{l}")
```

Output

```
0:a
1:changed
2:c
3:d
```

A **list** can have multiple objects added to it, such as strings. Duplicate values are allowed. **Tuples** do not allow the program to add additional objects after definition.

Code

```
# Manually add items, lists allow duplicates
c = []
c.append('a')
c.append('b')
c.append('c')
c.append('c')
print(c)
```

Output

```
['a', 'b', 'c', 'c']
```

Ordered collections, such as lists and tuples, allow you to access an element by its index number, such as is done in the following code. Unordered collections, such as dictionaries and sets, do not allow the program to access them in this way.

Code

```
print(c[1])
```

Output

```
b
```

A **list** can have multiple objects added to it, such as strings. Duplicate values are allowed. Tuples do not allow the program to add additional objects after definition. For the insert function, an index, the programmer must specify an index. These operations are not allowed for tuples because they would result in a change.

Code

```
# Insert
c = ['a', 'b', 'c']
c.insert(0, 'a0')
print(c)
# Remove
c.remove('b')
print(c)
# Remove at index
del c[0]
print(c)
```

Output

```
['a0', 'a', 'b', 'c']
['a0', 'a', 'c']
['a', 'c']
```

1.3.2 Sets

A Python **set** holds an unordered collection of objects, but sets do *not* allow duplicates. If a program adds a duplicate item to a set, only one copy of each item remains in the collection. Adding a duplicate item to a set does not result in an error. Any of the following techniques will define a set.

Code

```
s = set()
s = { 'a', 'b', 'c'}
s = set(['a', 'b', 'c'])
print(s)
```

Output

```
{'c', 'a', 'b'}
```

A **list** is always enclosed in square braces [], a **tuple** in parenthesis (), and now we see that the programmer

encloses a **set** in curly braces. Programs can add items to a **set** as they run. Programs can dynamically add items to a **set** with the **add** function. It is important to note that the **append** function adds items to lists and tuples, whereas the **add** function adds items to a **set**.

Code

```
# Manually add items, sets do not allow duplicates
# Sets add, lists append.  I find this annoying.
c = set()
c.add('a')
c.add('b')
c.add('c')
c.add('c')
print(c)
```

Output

```
{'c', 'a', 'b'}
```

1.3.3 Maps/Dictionaries/Hash Tables

Many programming languages include the concept of a map, dictionary, or hash table. These are all very related concepts. Python provides a dictionary, that is essentially a collection of name-value pairs. Programs define dictionaries using curly-braces, as seen here.

Code

```
d = {'name': "Jeff", 'address':"123 Main"}
print(d)
print(d['name'])

if 'name' in d:
    print("Name is defined")

if 'age' in d:
    print("age defined")
else:
    print("age undefined")
```

Output

```
{'name': 'Jeff', 'address': '123 Main'}
Jeff
Name is defined
age undefined
```

Be careful that you do not attempt to access an undefined key, as this will result in an error. You can

check to see if a key is defined, as demonstrated above. You can also access the directory and provide a default value, as the following code demonstrates.

Code

```
d.get('unknown_key', 'default')
```

Output

```
'default'
```

You can also access the individual keys and values of a dictionary.

Code

```
d = {'name': "Jeff", 'address':"123 Main"}
# All of the keys
print(f"Key: {d.keys()}")

# All of the values
print(f"Values: {d.values()}")
```

Output

```
Key: dict_keys(['name', 'address'])
Values: dict_values(['Jeff', '123 Main'])
```

Dictionaries and lists can be combined. This syntax is closely related to JSON. Dictionaries and lists together are a good way to build very complex data structures. While Python allows quotes (") and apostrophe (') for strings, JSON only allows double-quotes ("). We will cover JSON in much greater detail later in this module.

The following code shows a hybrid usage of dictionaries and lists.

Code

```
# Python list & map structures
customers = [
    {"name": "Jeff & Tracy Heaton", "pets": ["Wynton", "Cricket",
        "Hickory"]},
    {"name": "John Smith", "pets": ["rover"]},
    {"name": "Jane Doe"}
]

print(customers)

for customer in customers:
    print(f"{customer['name']}:{customer.get('pets', 'no pets')}")
```

Output

```
[{'name': 'Jeff & Tracy Heaton', 'pets': ['Wynton', 'Cricket',
'Hickory']}, {'name': 'John Smith', 'pets': ['rover']}, {'name': 'Jane
Doe'}]
Jeff & Tracy Heaton:['Wynton', 'Cricket', 'Hickory']
John Smith:['rover']
Jane Doe:no pets
```

The variable **customers** is a list that holds three dictionaries that represent customers. You can think of these dictionaries as records in a table. The fields in these individual records are the keys of the dictionary. Here the keys **name** and **pets** are fields. However, the field **pets** holds a list of pet names. There is no limit to how deep you might choose to nest lists and maps. It is also possible to nest a map inside of a map or a list inside of another list.

1.3.4 More Advanced Lists

Several advanced features are available for lists that this section introduces. One such function is **zip**. Two lists can be combined into a single list by the **zip** command. The following code demonstrates the **zip** command.

Code

```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

print(zip(a,b))
```

Output

```
<zip object at 0x000001802A7A2E08>
```

To see the results of the **zip** function, we convert the returned zip object into a list. As you can see, the **zip** function returns a list of tuples. Each tuple represents a pair of items that the function zipped together. The order in the two lists was maintained.

Code

```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

print(list(zip(a,b)))
```

Output

```
[(1, 5), (2, 4), (3, 3), (4, 2), (5, 1)]
```

The usual method for using the zip command is inside of a for-loop. The following code shows how a for-loop can assign a variable to each collection that the program is iterating.

Code

```
a = [1,2,3,4,5]
b = [5,4,3,2,1]

for x,y in zip(a,b):
    print(f'{x} - {y}')
```

Output

```
1 - 5
2 - 4
3 - 3
4 - 2
5 - 1
```

Usually, both collections will be of the same length when passed to the zip command. It is not an error to have collections of different lengths. As the following code illustrates, the zip command will only process elements up to the length of the smaller collection.

Code

```
a = [1,2,3,4,5]
b = [5,4,3]

print(list(zip(a,b)))
```

Output

```
[(1, 5), (2, 4), (3, 3)]
```

Sometimes you may wish to know the current numeric index when a for-loop is iterating through an ordered collection. Use the **enumerate** command to track the index location for a collection element. Because the **enumerate** command deals with numeric indexes of the collection, the zip command will assign arbitrary indexes to elements from unordered collections.

Consider how you might construct a Python program to change every element greater than 5 to the value of 5. The following program performs this transformation. The enumerate command allows the loop to know which element index it is currently on, thus allowing the program to be able to change the value of the current element of the collection.

Code

```
a = [2, 10, 3, 11, 10, 3, 2, 1]
for i, x in enumerate(a):
    if x>5:
        a[i] = 5
```

```
print(a)
```

Output

```
[2, 5, 3, 5, 5, 3, 2, 1]
```

The comprehension command can dynamically build up a list. The comprehension below counts from 0 to 9 and adds each value (multiplied by 10) to a list.

Code

```
lst = [x*10 for x in range(10)]
print(lst)
```

Output

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

A dictionary can also be a comprehension. The general format for this is:

```
dict_variable = {key:value for (key,value) in dictonary.items()}
```

A common use for this is to build up an index to symbolic column names.

Code

```
text = ['col-zero','col-one', 'col-two', 'col-three']
lookup = {key:value for (value,key) in enumerate(text)}
print(lookup)
```

Output

```
{'col-zero': 0, 'col-one': 1, 'col-two': 2, 'col-three': 3}
```

This can be used to easily find the index of a column by name.

Code

```
print(f'The␣index␣of␣"col-two"␣is␣{lookup["col-two"]}')
```

Output

```
The index of "col-two" is 2
```

1.3.5 An Introduction to JSON

Data stored in a CSV file must be flat; that is, it must fit into rows and columns. Most people refer to this type of data as structured or tabular. This data is tabular because the number of columns is the same for every row. Individual rows may be missing a value for a column; however, these rows still have the same columns.

This sort of data is convenient for machine learning because most models, such as neural networks, also expect incoming data to be of fixed dimensions. Real-world information is not always so tabular. Consider if the rows represent customers. These people might have multiple phone numbers and addresses. How would you describe such data using a fixed number of columns? It would be useful to have a list of these courses in each row that can be of a variable length for each row, or student.

JavaScript Object Notation (JSON) is a standard file format that stores data in a hierarchical format similar to eXtensible Markup Language (XML). JSON is nothing more than a hierarchy of lists and dictionaries. Programmers refer to this sort of data as semi-structured data or hierarchical data. The following is a sample JSON file.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

The above file may look somewhat like Python code. You can see curly braces that define dictionaries and square brackets that define lists. JSON does require there to be a single root element. A list or dictionary can fulfill this role. JSON requires double-quotes to enclose strings and names. Single quotes are not allowed in JSON.

JSON files are always legal JavaScript syntax. JSON is also generally valid as Python code, as demonstrated by the following Python program.

Code

```
jsonHardCoded = {
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": True,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": None
}
```

Generally, it is better to read JSON from files, strings, or the Internet than hard coding, as demonstrated here. However, for internal data structures, sometimes such hard-coding can be useful.

Python contains support for JSON. When a Python program loads a JSON the root list or dictionary is returned, as demonstrated by the following code.

Code

```
import json

json_string = '{"first":"Jeff","last":"Heaton"}'
obj = json.loads(json_string)
print(f"First name: {obj['first']}")
print(f"Last name: {obj['last']}")
```

Output

```
First name: Jeff
Last name: Heaton
```

Python programs can also load JSON from a file or URL.

Code

```
import requests

r = requests.get("https://raw.githubusercontent.com/jeffheaton/"
                 +"t81_558_deep_learning/master/person.json")
print(r.json())
```

Output

```
{'firstName': 'John', 'lastName': 'Smith', 'isAlive': True, 'age': 27,
'address': {'streetAddress': '21 2nd Street', 'city': 'New York',
'state': 'NY', 'postalCode': '10021-3100'}, 'phoneNumbers': [{'type':
'home', 'number': '212 555-1234'}, {'type': 'office', 'number': '646
555-4567'}, {'type': 'mobile', 'number': '123 456-7890'}], 'children':
[], 'spouse': None}
```

Python programs can easily generate JSON strings from Python objects of dictionaries and lists.

Code

```
python_obj = {"first":"Jeff","last":"Heaton"}
print(json.dumps(python_obj))
```

Output

```
{"first": "Jeff", "last": "Heaton"}
```

A data scientist will generally encounter JSON when they access web services to get their data. A data scientist might use the techniques presented in this section to convert the semi-structured JSON data into tabular data for the program to use with a model such as a neural network.

1.4 Part 1.4: File Handling

Files often contain the data that you use to train your AI programs. Once trained, your models may use real-time data to form predictions. These predictions might be made on files too. Regardless of if you are predicting or training, file processing is a vital skill for the AI practitioner.

There are many different types of files that you must process as an AI practitioner. Some of these file types are listed here:

- **CSV files** (generally have the .csv extension) hold tabular data that resembles spreadsheet data.
- **Image files** (generally with the .png or .jpg extension) hold images for computer vision.

- **Text files** (often have the .txt extension) hold unstructured text and are essential for natural language processing.
- **JSON** (often have the .json extension) contain semi-structured textual data in a human-readable text-based format.
- **H5** (can have a wide array of extensions) contain semi-structured textual data in a human-readable text-based format. Keras and TensorFlow store neural networks as H5 files.
- **Audio Files** (often have an extension such as .au or .wav) contain recorded sound.

Data can come from a variety of sources. In this class, we obtain data from three primary locations:

- **Your Hard Drive** - This type of data is stored locally, and Python accesses it from a path that looks something like: **c:\data\myfile.csv or /Users/jheaton/data/myfile.csv**.
- **The Internet** - This type of data resides in the cloud, and Python accesses it from a URL that looks something like:

https://data.heatonresearch.com/data/t81-558/iris.csv.

- **Google Drive (cloud)** - If you code in Google CoLab, you make use of GoogleDrive to save and load some data files. CoLab mounts your GoogleDrive into a path similar to the following: **/content/drive/My Drive/myfile.csv**.

1.4.1 Read a CSV File

Python programs can read CSV files with Pandas. We will see more about Pandas in the next section, but for now, its general format is:

Code

```
import pandas as pd

df = pd.read_csv("https://data.heatonresearch.com/data/t81-558/iris.csv")
```

The above command loads Fisher's Iris data set from the Internet. It might take a few seconds to load, so it is good to keep the loading code in a separate Jupyter notebook cell so that you do not have to reload it as you test your program. You can load Internet data, local hard drive, and Google Drive data this way.

Now that the data is loaded, you can display the first five rows with this command.

Code

```
display(df[0:5])
```

Output

	sepal_l	sepal_w	petal_l	petal_w	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

1.4.2 Read (stream) a Large CSV File

Pandas will read the entire CSV file into memory. Usually, this is fine. However, at times you may wish to "stream" a huge file. Streaming allows you to process this file one record at a time. Because the program does not load all of the data into memory, you can handle huge files. The following code loads the Iris dataset and calculates averages, one row at a time. This technique would work for large files.

Code

```
import csv
import urllib.request
import codecs
import numpy as np

url = "https://data.heatonresearch.com/data/t81-558/iris.csv"
urlstream = urllib.request.urlopen(url)
csvfile = csv.reader(codecs.iterdecode(urlstream, 'utf-8'))
next(csvfile) # Skip header row
sum = np.zeros(4)
count = 0

for line in csvfile:
    # Convert each row to Numpy array
    line2 = np.array(line)[0:4].astype(float)

    # If the line is of the right length (skip empty lines), then add
    if len(line2) == 4:
        sum += line2
        count += 1

# Calculate the average, and print the average of the 4 iris
# measurements (features)
print(sum/count)
```

Output

```
[5.84333333 3.05733333 3.758      1.19933333]
```

1.4.3 Read a Text File

The following code reads the USA Declaration of Independence as a text file. This code streams the document and reads it line-by-line. This code could handle a huge file.

Code

```
import urllib.request

url = "https://data.heatonresearch.com/data/t81-558/datasets/sonnet_18.txt"
with urllib.request.urlopen(url) as urlstream:
```

```
    for line in codecs.iterdecode(urlstream, 'utf-8'):
        print(line.rstrip())
```

Output

```
Sonnet 18 original text
William Shakespeare
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.
```

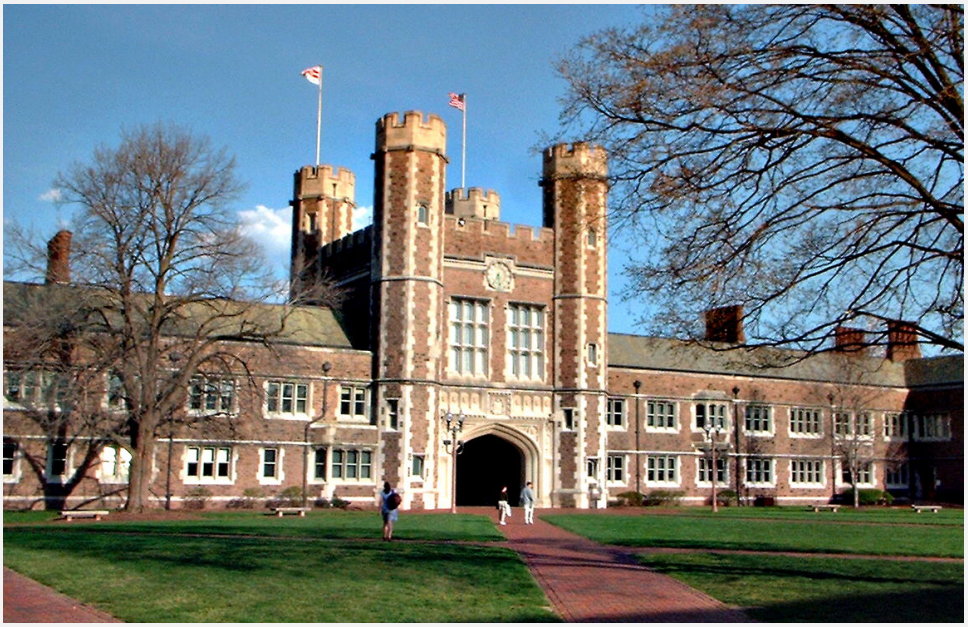
1.4.4 Read an Image

Computer vision is one of the areas that neural networks outshine other models. To support computer vision, the Python programmer needs to understand how to process images. For this course, we will use the Python PIL package for image processing. The following code demonstrates how to load an image from a URL and display it.

Code

```
%matplotlib inline
from PIL import Image
import requests
from io import BytesIO

url = "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))

img
```

Output



1.5 Part 1.5: Functions, Lambdas, and Map/Reduce

Functions, **lambdas**, and **map/reduce** can allow you to process your data in advanced ways. We will introduce these techniques here and expand on them in the next module, which will discuss Pandas.

Function parameters can be named or unnamed in Python. Default values can also be used. Consider the following function.

Code

```
def say_hello(speaker, person_to_greet, greeting = "Hello"):
    print(f'{greeting} {person_to_greet}, this is {speaker}.')

say_hello('Jeff', "John")
say_hello('Jeff', "John", "Goodbye")
say_hello(speaker='Jeff', person_to_greet="John", greeting = "Goodbye")
```

Output

```
Hello John, this is Jeff.
Goodbye John, this is Jeff.
Goodbye John, this is Jeff.
```

A function is a way to capture code that is commonly executed. Consider the following function that can be used to trim white space from a string capitalize the first letter.

Code

```
def process_string(str):
    t = str.strip()
    return t[0].upper()+t[1:]
```

This function can now be called quite easily.

Code

```
str = process_string("␣␣hello␣␣")
print(f'"{str}"')
```

Output

```
"Hello"
```

Python's **map** is a very useful function that is provided in many different programming languages. The **map** function takes a **list** and applies a function to each member of the **list** and returns a second **list** that is the same size as the first.

Code

```
l = ['␣␣␣apple␣␣', 'pear␣', 'orange', 'pine␣apple␣␣']
list(map(process_string, l))
```

Output

```
['Apple', 'Pear', 'Orange', 'Pine apple']
```

1.5.1 Map

The **map** function is very similar to the Python **comprehension** that we previously explored. The following **comprehension** accomplishes the same task as the previous call to **map**.

Code

```
l = ['␣␣␣apple␣␣', 'pear␣', 'orange', 'pine␣apple␣␣']
l2 = [process_string(x) for x in l]
print(l2)
```

Output

```
['Apple', 'Pear', 'Orange', 'Pine apple']
```

The choice of using a **map** function or **comprehension** is up to the programmer. I tend to prefer **map** since it is so common in other programming languages.

1.5.2 Filter

While a **map function** always creates a new **list** of the same size as the original, the **filter** function creates a potentially smaller **list**.

Code

```
def greater_than_five(x):
    return x>5

l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(greater_than_five, l))
print(l2)
```

Output

```
[10, 20]
```

1.5.3 Lambda

It might seem somewhat tedious to have to create an entire function just to check to see if a value is greater than 5. A **lambda** saves you this effort. A lambda is essentially an unnamed function.

Code

```
l = [ 1, 10, 20, 3, -2, 0]
l2 = list(filter(lambda x: x>5, l))
print(l2)
```

Output

```
[10, 20]
```

1.5.4 Reduce

Finally, we will make use of **reduce**. Like **filter** and **map** the **reduce** function also works on a **list**. However, the result of the **reduce** is a single value. Consider if you wanted to sum the **values** of a **list**. The sum is implemented by a **lambda**.

Code

```
from functools import reduce

l = [ 1, 10, 20, 3, -2, 0]
result = reduce(lambda x,y: x+y,l)
print(result)
```

Output

```
32
```

Chapter 2

Python for Machine Learning

2.1 Part 2.1: Introduction to Pandas

Pandas is an open-source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. It is based on the dataframe concept found in the R programming language. For this class, Pandas will be the primary means by which we manipulate data to be processed by neural networks.

The data frame is a crucial component of Pandas. We will use it to access the auto-mpg dataset. You can find this dataset on the UCI machine learning repository. For this class, we will use a version of the Auto MPG dataset, where I added column headers. You can find my version here.

UCI took this dataset from the StatLib library, which Carnegie Mellon University maintains. The dataset was used in the 1983 American Statistical Association Exposition. It contains data for 398 cars, including mpg, cylinders, displacement, horsepower , weight, acceleration, model year, origin and the car's name.

The following code loads the MPG dataset into a data frame:

Code

```
# Simple dataframe
import os
import pandas as pd

df = pd.read_csv("https://data.heatonresearch.com/data/t81-558/auto-mpg.csv")
print(df[0:5])
```

Output

```
mpg  cylinders  displacement horsepower  weight  acceleration
year  \
0  18.0          8          307.0        130     3504          12.0
70
1  15.0          8          350.0        165     3693          11.5
70
2  18.0          8          318.0        150     3436          11.0
70
3  16.0          8          304.0        150     3433          12.0
```

```
70
4  17.0          8          302.0         140     3449          10.5
70
   origin                        name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

The **display** function provides a cleaner display than merely printing the data frame. Specifying the maximum rows and columns allows you to achieve greater control over the display.

Code

```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

It is possible to generate a second data frame to display statistical information about the first data frame.

Code

```
# Strip non-numerics
df = df.select_dtypes(include=['int', 'float'])

headers = list(df.columns.values)
fields = []

for field in headers:
    fields.append({
        'name' : field,
        'mean': df[field].mean(),
        'var': df[field].var(),
        'sdev': df[field].std()
    })

for field in fields:
    print(field)
```

Output

```
{'name': 'mpg', 'mean': 23.514572864321607, 'var': 61.089610774274405,
'sdev': 7.815984312565782}
{'name': 'cylinders', 'mean': 5.454773869346734, 'var':
2.893415439920003, 'sdev': 1.7010042445332119}
{'name': 'displacement', 'mean': 193.42587939698493, 'var':
10872.199152247384, 'sdev': 104.26983817119591}
{'name': 'weight', 'mean': 2970.424623115578, 'var':
717140.9905256763, 'sdev': 846.8417741973268}
{'name': 'acceleration', 'mean': 15.568090452261307, 'var':
7.604848233611383, 'sdev': 2.757688929812676}
{'name': 'year', 'mean': 76.01005025125629, 'var': 13.672442818627143,
'sdev': 3.697626646732623}
{'name': 'origin', 'mean': 1.5728643216080402, 'var':
0.6432920268850549, 'sdev': 0.8020548777266148}
```

This code outputs a list of dictionaries that hold this statistical information. This information looks similar to the JSON code seen in Module 1. To as proper JSON, the program should add these records to a list and call the Python JSON library's **dumps** command called.

The Python program can convert this JSON-like information to a data frame for better display.

Code

```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)
df2 = pd.DataFrame(fields)
display(df2)
```

Output

	name	mean	var	sdev
0	mpg	23.514573	61.089611	7.815984
1	cylinders	5.454774	2.893415	1.701004
2	displacement	193.425879	10872.199152	104.269838
3	weight	2970.424623	717140.990526	846.841774
4	acceleration	15.568090	7.604848	2.757689
5	year	76.010050	13.672443	3.697627
6	origin	1.572864	0.643292	0.802055

2.1.1 Missing Values

Missing values are a reality of machine learning. Ideally, every row of data will have values for all columns. However, this is rarely the case. Most of the values are present in the MPG database. However, there are missing values in the horsepower column. A common practice is to replace missing values with the median value for that column. The program calculates the median as described here. The following code replaces any NA values in horsepower with the median:

Code

```
import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])
print(f"horsepower has na? {pd.isnull(df['horsepower']).values.any()}")

print("Filling missing values...")
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)
# df = df.dropna() # you can also simply drop NA values

print(f"horsepower has na? {pd.isnull(df['horsepower']).values.any()}")
```

Output

```
horsepower has na? True
Filling missing values...
horsepower has na? False
```

2.1.2 Dealing with Outliers

Outliers are values that are unusually high or low. Sometimes outliers are simply errors; this is a result of observation error. Outliers can also be truly large or small values that may be difficult to address. We typically consider outliers to be a value that is several standard deviations from the mean. The following function can remove such values.

Code

```
# Remove all rows where the specified column is +/- sd standard deviations
def remove_outliers(df, name, sd):
    drop_rows = df.index[(np.abs(df[name] - df[name].mean())
                          >= (sd * df[name].std()))]
    df.drop(drop_rows, axis=0, inplace=True)
```

The code below will drop every row from the Auto MPG dataset where the horsepower is more than two standard deviations above or below the mean.

Code

```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
```

```
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

# create feature vector
med = df['horsepower'].median()
df['horsepower'] = df['horsepower'].fillna(med)

# Drop the name column
df.drop('name',1,inplace=True)

# Drop outliers in horsepower
print("Length before MPG outliers dropped: {}".format(len(df)))
remove_outliers(df,'mpg',2)
print("Length after MPG outliers dropped: {}".format(len(df)))

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)
display(df)
```

Output

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin
0	18.0	8	307.0	130.0	3504	12.0	70	1
1	15.0	8	350.0	165.0	3693	11.5	70	1
...	...	...	...	...	...	...	...	...
396	28.0	4	120.0	79.0	2625	18.6	82	1
397	31.0	4	119.0	82.0	2720	19.4	82	1

```
Length before MPG outliers dropped: 398
Length after MPG outliers dropped: 388
```

2.1.3 Dropping Fields

Some fields are of no value to the neural network should be dropped. The following code removes the name column from the MPG dataset.

Code

```
import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

print(f"Before drop: {list(df.columns)}")
```

```
df.drop('name', 1, inplace=True)
print(f"After drop: {list(df.columns)}")
```

Output

```
Before drop: ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
After drop: ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin']
```

2.1.4 Concatenating Rows and Columns

Python can concatenate rows and columns together to form new data frames. The code below creates a new data frame from the **name** and **horsepower** columns from the Auto MPG dataset. The program does this by concatenating two columns together.

Code

```
# Create a new dataframe from name and horsepower

import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

col_horsepower = df['horsepower']
col_name = df['name']
result = pd.concat([col_name, col_horsepower], axis=1)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)
display(result)
```

Output

	name	horsepower
0	chevrolet chevelle malibu	130.0
1	buick skylark 320	165.0
...	...	...
396	ford ranger	79.0
397	chevy s-10	82.0

The **concat** function can also concatenate two rows together. This code concatenates the first two rows and the last two rows of the Auto MPG dataset.

Code

```
# Create a new dataframe from first 2 rows and last 2 rows

import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

result = pd.concat([df[0:2],df[-2:]], axis=0)

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 0)
display(result)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

2.1.5 Training and Validation

We must evaluate a machine learning model based on its ability to predict data that it has never seen before. Because of this, we often divide the training data into a validation and training set. The machine learning model will learn from the training data, but ultimately be evaluated based on the validation data.

- **Training Data** - **In Sample Data** - The data that the neural network used to train.
- **Validation Data** - **Out of Sample Data** - The data that the machine learning model is evaluated upon after it is fit to the training data.

There are two effective means of dealing with training and validation data:

- **Training/Validation Split** - The program splits the data according to some ratio between a training and validation (hold-out) set. Typical rates are 80% training and 20% validation.
- **K-Fold Cross Validation** - The program splits the data into several folds and models. Because the program creates the same number of models as folds, the program can generate out-of-sample predictions for the entire dataset.

The code below performs a split of the MPG data into a training and validation set. The training set uses 80% of the data, and the validation set uses 20%. Figure 2.1 shows how a model is trained on 80% of the data and then validated against the remaining 20%.

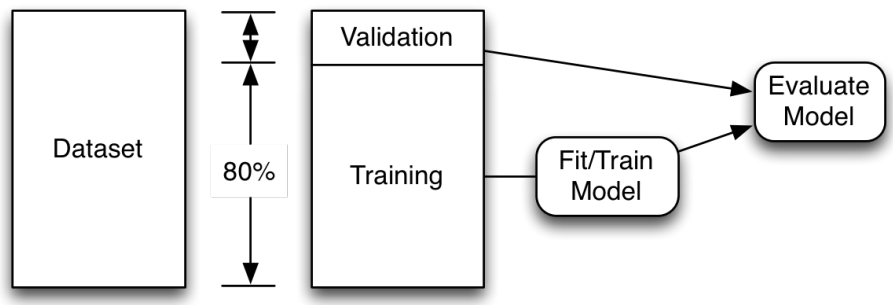


Figure 2.1: Training and Validation

Code

```
import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

# Usually a good idea to shuffle
df = df.reindex(np.random.permutation(df.index))

mask = np.random.rand(len(df)) < 0.8
trainDF = pd.DataFrame(df[mask])
validationDF = pd.DataFrame(df[~mask])

print(f"Training DF: {len(trainDF)}")
print(f"Validation DF: {len(validationDF)}")
```

Output

```
Training DF: 333
Validation DF: 65
```

2.1.6 Converting a Dataframe to a Matrix

Neural networks do not directly operate on Python data frames. A neural network requires a numeric matrix. The program uses the **values** property of a data frame to convert the data to a matrix.

Code

```
df.values
```

Output

```
array([[20.2, 6, 232.0, ..., 79, 1, 'amc concord dl 6'],
       [14.0, 8, 304.0, ..., 74, 1, 'amc matador (sw)'],
       [14.0, 8, 351.0, ..., 71, 1, 'ford galaxie 500'],
       ...,
       [20.2, 6, 200.0, ..., 78, 1, 'ford fairmont (auto)'],
       [26.0, 4, 97.0, ..., 70, 2, 'volkswagen 1131 deluxe sedan'],
       [19.4, 6, 232.0, ..., 78, 1, 'amc concord']], dtype=object)
```

You might wish only to convert some of the columns, to leave out the name column, use the following code.

Code

```
df[['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
```

Output

```
array([[ 20.2,   6. , 232. , ...,  18.2,  79. ,   1. ],
       [ 14. ,   8. , 304. , ...,  15.5,  74. ,   1. ],
       [ 14. ,   8. , 351. , ...,  13.5,  71. ,   1. ],
       ...,
       [ 20.2,   6. , 200. , ...,  15.8,  78. ,   1. ],
       [ 26. ,   4. ,  97. , ...,  20.5,  70. ,   2. ],
       [ 19.4,   6. , 232. , ...,  17.2,  78. ,   1. ]])
```

2.1.7 Saving a Dataframe to CSV

Many of the assignments in this course will require that you save a data frame to submit to the instructor. The following code performs a shuffle and then saves a new copy.

Code

```
import os
import pandas as pd
import numpy as np

path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
```

```
    na_values=['NA','?'])

filename_write = os.path.join(path, "auto-mpg-shuffle.csv")
df = df.reindex(np.random.permutation(df.index))
# Specify index = false to not write row numbers
df.to_csv(filename_write, index=False)
print("Done")
```

Output

```
Done
```

2.1.8 Saving a Dataframe to Pickle

A variety of software programs can make use of text files stored as CSV. However, they do take longer to generate and can sometimes lose small amounts of precision in the conversion. Another format is Pickle. Generally, you will output to CSV because it is very compatible, even outside of Python. The code below stores the Dataframe to Pickle.

Code

```
import os
import pandas as pd
import numpy as np
import pickle

path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

filename_write = os.path.join(path, "auto-mpg-shuffle.pkl")
df = df.reindex(np.random.permutation(df.index))

with open(filename_write,"wb") as fp:
    pickle.dump(df, fp)
```

Loading the pickle file back into memory is accomplished by the following lines of code. Notice that the index numbers are still jumbled from the previous shuffle? Loading the CSV rebuilt (in the last step) did not preserve these values.

Code

```
import os
import pandas as pd
import numpy as np
import pickle
```

```
path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

filename_read = os.path.join(path, "auto-mpg-shuffle.pkl")

with open(filename_write,"rb") as fp:
    df = pickle.load(fp)

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
387	38.0	6	262.0	...	82	1	oldsmobile cutlass ciera (diesel)
361	25.4	6	168.0	...	81	3	toyota cressida
...	...	...	...	...	...	...	...
358	31.6	4	120.0	...	81	3	mazda 626
237	30.5	4	98.0	...	77	1	chevrolet chevette

2.1.9 Module 2 Assignment

You can find the first assignment here: assignment 2

2.2 Part 2.2: Categorical and Continuous Values

Neural networks require their input to be a fixed number of columns. This input format is very similar to spreadsheet data. This input must be entirely numeric.

It is essential to represent the data in a way that the neural network can train from it. In class 6, we will see even more ways to preprocess data. For now, we will look at several of the most basic ways to transform data for a neural network.

Before we look at specific ways to preprocess data, it is important to consider four basic types of data, as defined by[**?**]. Statisticians commonly refer to as the levels of measure:

- Character Data (strings)
  - **Nominal** - Individual discrete items, no order. For example, color, zip code, shape.
  - **Ordinal** - Individual distinct items have an implied order. For example grade level, job title, Starbucks(tm) coffee size (tall, vente, grande)
- Numeric Data
  - **Interval** - Numeric values, no defined start. For example, temperature. You would never say, "yesterday was twice as hot as today."

- **Ratio** - Numeric values, clearly defined start. For example, speed. You would say that "The first car is going twice as fast as the second."

2.2.1 Encoding Continuous Values

One common transformation is to normalize the inputs. It is sometimes valuable to normalization numeric inputs to be put in a standard form so that the program can easily compare these two values. Consider if a friend told you that he received a 10 dollar discount. Is this a good deal? Maybe. But the cost is not normalized. If your friend purchased a car, then the discount is not that good. If your friend bought dinner, this is an excellent discount!

Percentages are a prevalent form of normalization. If your friend tells you they got 10% off, we know that this is a better discount than 5%. It does not matter how much the purchase price was. One widespread machine learning normalization is the Z-Score:

$$z = \frac{x - \mu}{\sigma}$$

To calculate the Z-Score you need to also calculate the mean(μ) and the standard deviation (σ). The mean is calculated as follows:

$$\mu = \bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

The standard deviation is calculated as follows:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \text{ where } \mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

The following Python code replaces the mpg with a z-score. Cars with average MPG will be near zero, above zero is above average, and below zero is below average. Z-Scores above/below -3/3 are very rare, these are outliers.

Code

```
import os
import pandas as pd
from scipy.stats import zscore

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df['mpg'] = zscore(df['mpg'])
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
0	-0.706439	8	307.0	...	70	1	chevrolet chevelle malibu
1	-1.090751	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	0.574601	4	120.0	...	82	1	ford ranger
397	0.958913	4	119.0	...	82	1	chevy s-10

2.2.2 Encoding Categorical Values as Dummies

The traditional means of encoding categorical values is to make them dummy variables. This technique is also called one-hot-encoding. Consider the following data set.

Code

```
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

display(df)
```

Output

	id	job	area	...	retail_dense	crime	product
0	1	vv	c	...	0.492126	0.071100	b
1	2	kd	c	...	0.342520	0.400809	c
...	...	...	...	...	...	...	...
1998	1999	qp	c	...	0.598425	0.117803	c
1999	2000	pe	c	...	0.539370	0.451973	c

Code

```
areas = list(df['area'].unique())
print(f'Number of areas: {len(areas)}')
print(f'Areas: {areas}')
```

Output

```
Number of areas: 4
Areas: ['c', 'd', 'a', 'b']
```

There are four unique values in the areas column. To encode these to dummy variables, we would use

four columns, each of which would represent one of the areas. For each row, one column would have a value of one, the rest zeros. For this reason, this type of encoding is sometimes called one-hot encoding. The following code shows how you might encode the values "a" through "d." The value A becomes [1,0,0,0] and the value B becomes [0,1,0,0].

Code

```
dummies = pd.get_dummies(['a','b','c','d'],prefix='area')
print(dummies)
```

Output

```
area_a  area_b  area_c  area_d
0       1       0       0       0
1       0       1       0       0
2       0       0       1       0
3       0       0       0       1
```

Code

```
dummies = pd.get_dummies(df['area'],prefix='area')
print(dummies[0:10]) # Just show the first 10
```

Output

```
area_a  area_b  area_c  area_d
0        0       0       1       0
1        0       0       1       0
..     ...     ...     ...     ...
8        0       0       1       0
9        1       0       0       0
[10 rows x 4 columns]
```

Code

```
df = pd.concat([df,dummies],axis=1)
```

To encode the "area" column, we use the following. Note that it is necessary to merge these dummies back into the data frame.

Code

```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df[['id','job','area','income','area_a',
                  'area_b','area_c','area_d']])
```

Output

	id	job	area	income	area_a	area_b	area_c	area_d
0	1	vv	c	50876.0	0	0	1	0
1	2	kd	c	60369.0	0	0	1	0
2	3	pe	c	55126.0	0	0	1	0
3	4	11	c	51690.0	0	0	1	0
4	5	kl	d	28347.0	0	0	0	1
...	...	...	...	...	...	...	...	...
1995	1996	vv	c	51017.0	0	0	1	0
1996	1997	kl	d	26576.0	0	0	0	1
1997	1998	kl	d	28595.0	0	0	0	1
1998	1999	qp	c	67949.0	0	0	1	0
1999	2000	pe	c	61467.0	0	0	1	0

Usually, you will remove the original column ('area'), because it is the goal to get the data frame to be entirely numeric for the neural network.

Code

```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)

df.drop('area', axis=1, inplace=True)
display(df[['id','job','income','area_a',
                    'area_b','area_c','area_d']])
```

Output

	id	job	income	area_a	area_b	area_c	area_d
0	1	vv	50876.0	0	0	1	0
1	2	kd	60369.0	0	0	1	0
...	...	...	...	...	...	...	...
1998	1999	qp	67949.0	0	0	1	0
1999	2000	pe	61467.0	0	0	1	0

2.2.3 Target Encoding for Categoricals

Target encoding can sometimes increase the predictive power of a machine learning model. However, it also dramatically increases the risk of overfitting. Because of this risk, you must take care if you are using this method. Target encoding is a popular technique for Kaggle competitions.

Generally, target encoding can only be used on a categorical feature when the output of the machine learning model is numeric (regression).

The concept of target encoding is straightforward. For each category, we calculate the average target value for that category. Then to encode, we substitute the percent that corresponds to the category that the categorical value has. Unlike dummy variables, where you have a column for each category, with target encoding, the program only needs a single column. In this way, target coding is more efficient than dummy variables

Code

```
# Create a small sample dataset
import pandas as pd
import numpy as np

np.random.seed(43)
df = pd.DataFrame({
    'cont_9': np.random.rand(10)*100,
    'cat_0': ['dog'] * 5 + ['cat'] * 5,
    'cat_1': ['wolf'] * 9 + ['tiger'] * 1,
    'y': [1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
})

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)
display(df)
```

Output

	cont_9	cat_0	cat_1	y
0	11.505457	dog	wolf	1
1	60.906654	dog	wolf	0
2	13.339096	dog	wolf	1
3	24.058962	dog	wolf	1
4	32.713906	dog	wolf	1
5	85.913749	cat	wolf	1
6	66.609021	cat	wolf	0
7	54.116221	cat	wolf	0
8	2.901382	cat	wolf	0
9	73.374830	cat	tiger	0

Rather than creating dummy variables for "dog" and "cat," we would like to change it to a number. We could use 0 for cat, 1 for dog. However, we can encode more information than just that. The simple 0 or 1 would also only work for one animal. Consider what the mean target value is for cat and dog.

Code

```
means0 = df.groupby('cat_0')['y'].mean().to_dict()
means0
```

Output

```
{'cat': 0.2, 'dog': 0.8}
```

The danger is that we are now using the target value for training. This technique will potentially lead to overfitting. The possibility of overfitting is even greater if there are a small number of a particular category. To prevent this from happening, we use a weighting factor. The stronger the weight, the more than categories

with a small number of values will tend towards the overall average of y. You can perform this calculation as follows.

Code

```
df['y'].mean()
```

Output

```
0.5
```

You can implement target encoding as follows. For more information on Target Encoding, refer to the article "Target Encoding Done the Right Way", that I based this code upon.

Code

```
def calc_smooth_mean(df1, df2, cat_name, target, weight):
    # Compute the global mean
    mean = df[target].mean()

    # Compute the number of values and the mean of each group
    agg = df.groupby(cat_name)[target].agg(['count', 'mean'])
    counts = agg['count']
    means = agg['mean']

    # Compute the "smoothed" means
    smooth = (counts * means + weight * mean) / (counts + weight)

    # Replace each value by the according smoothed mean
    if df2 is None:
        return df1[cat_name].map(smooth)
    else:
        return df1[cat_name].map(smooth),df2[cat_name].map(smooth.to_dict())
```

The following code encodes these two categories.

Code

```
WEIGHT = 5
df['cat_0_enc'] = calc_smooth_mean(df1=df, df2=None,
    cat_name='cat_0', target='y', weight=WEIGHT)
df['cat_1_enc'] = calc_smooth_mean(df1=df, df2=None,
    cat_name='cat_1', target='y', weight=WEIGHT)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)

display(df)
```

Output

	cont_9	cat_0	cat_1	y	cat_0_enc	cat_1_enc
0	11.505457	dog	wolf	1	0.65	0.535714
1	60.906654	dog	wolf	0	0.65	0.535714
2	13.339096	dog	wolf	1	0.65	0.535714
3	24.058962	dog	wolf	1	0.65	0.535714
4	32.713906	dog	wolf	1	0.65	0.535714
5	85.913749	cat	wolf	1	0.35	0.535714
6	66.609021	cat	wolf	0	0.35	0.535714
7	54.116221	cat	wolf	0	0.35	0.535714
8	2.901382	cat	wolf	0	0.35	0.535714
9	73.374830	cat	tiger	0	0.35	0.416667

2.2.4 Encoding Categorical Values as Ordinal

Typically categoricals will be encoded as dummy variables. However, there might be other techniques to convert categoricals to numeric. Any time there is an order to the categoricals, a number should be used. Consider if you had a categorical that described the current education level of an individual.

- Kindergarten (0)
- First Grade (1)
- Second Grade (2)
- Third Grade (3)
- Fourth Grade (4)
- Fifth Grade (5)
- Sixth Grade (6)
- Seventh Grade (7)
- Eighth Grade (8)
- High School Freshman (9)
- High School Sophomore (10)
- High School Junior (11)
- High School Senior (12)
- College Freshman (13)
- College Sophomore (14)
- College Junior (15)
- College Senior (16)
- Graduate Student (17)
- PhD Candidate (18)
- Doctorate (19)
- Post Doctorate (20)

The above list has 21 levels. This would take 21 dummy variables. However, simply encoding this to dummies would lose the order information. Perhaps the easiest approach would be to assign simply number them and assign the category a single number that is equal to the value in parenthesis above. However, we might be able to do even better. Graduate student is likely more than a year, so you might increase more than just one value.

2.3 Part 2.3: Grouping, Sorting, and Shuffling

Now we will take a look at a few ways to affect an entire Pandas data frame. These techniques will allow us to group, sort, and shuffle data sets. These are all essential operations for both data preprocessing and evaluation.

2.3.1 Shuffling a Dataset

There may be information lurking in the order of the rows of your dataset. Unless you are dealing with time-series data, the order of the rows should not be significant. Consider if your training set included employees in a company. Perhaps this dataset is ordered by the number of years that the employees were with the company. It is okay to have an individual column that specifies years of service. However, having the data in this order might be problematic.

Consider if you were to split the data into training and validation. You could end up with your validation set having only the newer employees and the training set longer-term employees. Separating the data into a k-fold cross validation could have similar problems. Because of these issues, it is important to shuffle the data set.

Often shuffling and reindexing are both performed together. Shuffling randomizes the order of the data set. However, it does not change the Pandas row numbers. The following code demonstrates a reshuffle. Notice that the first column, the row indexes, has not been reset. Generally, this will not cause any issues and allows trace back to the original order of the data. However, I usually prefer to reset this index. I reason that I typically do not care about the initial position, and there are a few instances where this unordered index can cause issues.

Code

```
import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

#np.random.seed(42) # Uncomment this line to get the same shuffle each time
df = df.reindex(np.random.permutation(df.index))

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
117	29.0	4	68.0	...	73	2	fiat 128
245	36.1	4	98.0	...	78	1	ford fiesta
...	...	...	...	...	...	...	...
88	14.0	8	302.0	...	73	1	ford gran torino
26	10.0	8	307.0	...	70	1	chevy c20

The following code demonstrates a reindex. Notice how the reindex orders the row indexes.

Code

```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df.reset_index(inplace=True, drop=True)
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
0	29.0	4	68.0	...	73	2	fiat 128
1	36.1	4	98.0	...	78	1	ford fiesta
...	...	...	...	...	...	...	...
396	14.0	8	302.0	...	73	1	ford gran torino
397	10.0	8	307.0	...	70	1	chevy c20

2.3.2 Sorting a Data Set

While it is always a good idea to shuffle a data set before training, during training and preprocessing, you may also wish to sort the data set. Sorting the data set allows you to order the rows in either ascending or descending order for one or more columns. The following code sorts the MPG dataset by name and displays the first car.

Code

```
import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

df = df.sort_values(by='name', ascending=True)
print(f"The first car is: {df['name'].iloc[0]}")

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
96	13.0	8	360.0	...	73	1	amc ambassador brougham
9	15.0	8	390.0	...	70	1	amc ambassador dpl
...	...	...	...	...	...	...	...
325	44.3	4	90.0	...	80	2	vw rabbit c (diesel)
293	31.9	4	89.0	...	79	2	vw rabbit custom

```
The first car is: amc ambassador brougham
```

2.3.3 Grouping a Data Set

Grouping is a typical operation on data sets. Structured Query Language (SQL) calls this operation a "GROUP BY." Programmers use grouping to summarize data. Because of this, the summarization row count will usually shrink, and you cannot undo the grouping. Because of this loss of information, it is essential to keep your original data before the grouping.

The Auto MPG dataset is used to demonstrate grouping.

Code

```
import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

The above data set can be used with the group to perform summaries. For example, the following code will group cylinders by the average (mean). This code will provide the grouping. In addition to **mean**, you can use other aggregating functions, such as **sum** or **count**.

Code

```
g = df.groupby('cylinders')['mpg'].mean()
g
```

Output

```
cylinders
3     20.550000
4     29.286765
5     27.366667
6     19.985714
8     14.963107
Name: mpg, dtype: float64
```

It might be useful to have these **mean** values as a dictionary.

Code

```
d = g.to_dict()
d
```

Output

```
{3: 20.55,
 4: 29.28676470588236,
 5: 27.366666666666664,
 6: 19.985714285714284,
 8: 14.963106796116508}
```

A dictionary allows you to access an individual element quickly. For example, you could quickly look up the mean for six-cylinder cars. You will see that target encoding, introduced later in this module, makes use of this technique.

Code

```
d[6]
```

Output

```
19.985714285714284
```

The code below shows how to count the number of rows that match each cylinder count.

Code

```
df.groupby('cylinders')['mpg'].count().to_dict()
```

Output

```
{3: 4, 4: 204, 5: 3, 6: 84, 8: 103}
```

2.4 Part 2.4: Apply and Map

If you've ever worked with Big Data or functional programming languages before, you've likely heard of map/reduce. Map and reduce are two functions that apply a task that you create to a data frame. Pandas supports functional programming techniques that allow you to use functions across en entire data frame. In addition to functions that you write, Pandas also provides several standard functions for use with data frames.

2.4.1 Using Map with Dataframes

The map function allows you to transform a column by mapping certain values in that column to other values. Consider the Auto MPG data set that contains a field **origin_name** that holds a value between one and three that indicates the geographic origin of each car. We can see how to use the map function to transform this numeric origin into the textual name of each origin.

We will begin by loading the Auto MPG data set.

Code

```
import os
import pandas as pd
import numpy as np

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

display(df)
```

Output

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	4	120.0	...	82	1	ford ranger
397	31.0	4	119.0	...	82	1	chevy s-10

The **map** method in Pandas operates on a single column. You provide **map** with a dictionary of values to transform the target column. The map keys specify what values in the target column should be turned into values specified by those keys. The following code shows how the map function can transform the numeric values of 1, 2, and 3 into the string values of North America, Europe and Asia.

Code

```
# Apply the map
df['origin_name'] = df['origin'].map(
    {1: 'North America', 2: 'Europe', 3: 'Asia'})
```

```
# Shuffle the data, so that we hopefully see
# more regions.
df = df.reindex(np.random.permutation(df.index))

# Display
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
display(df)
```

Output

	mpg	cylinders	displacement	...	origin	name	origin_name
45	18.0	6	258.0	...	1	amc hornet sportabout (sw)	North America
290	15.5	8	351.0	...	1	ford country squire (sw)	North America
313	28.0	4	151.0	...	1	chevrolet citation	North America
82	23.0	4	120.0	...	3	toyouta corona mark ii (sw)	Asia
33	19.0	6	232.0	...	1	amc gremlin	North America
...	...	...	...	...	...	...	...
329	44.6	4	91.0	...	3	honda civic 1500 gl	Asia
326	43.4	4	90.0	...	2	vw dasher (diesel)	Europe
34	16.0	6	225.0	...	1	plymouth satellite custom	North America
118	24.0	4	116.0	...	2	opel manta	Europe
15	22.0	6	198.0	...	1	plymouth duster	North America

2.4.2 Using Apply with Dataframes

The **apply** function of the data frame can run a function over the entire data frame. You can use either be a traditional named function or a lambda function. Python will execute the provided function against each of the rows or columns in the data frame. The **axis** parameter specifies of the function is run across rows or columns. For axis = 1, rows are used. The following code calculates a series called **efficiency** that is the **displacement** divided by **horsepower**.

Code

```
efficiency = df.apply(lambda x: x['displacement']/x['horsepower'], axis=1)
display(efficiency[0:10])
```

Output

```
45     2.345455
290    2.471831
313    1.677778
82     1.237113
33     2.320000
249    2.363636
27     1.514286
```

```
7       2.046512
302     1.500000
179     1.234694
dtype: float64
```

You can now insert this series into the data frame, either as a new column or to replace an existing column. The following code inserts this new series into the data frame.

Code

```
df['efficiency'] = efficiency
```

2.4.3 Feature Engineering with Apply and Map

In this section, we will see how to calculate a complex feature using map, apply, and grouping. The data set is the following CSV:

- https://www.irs.gov/pub/irs-soi/16zpallagi.csv

This URL contains US Government public data for "SOI Tax Stats - Individual Income Tax Statistics." The entry point to the website is here:

- https://www.irs.gov/statistics/soi-tax-stats-individual-income-tax-statistics-2016-zip-code-data-soi

Documentation describing this data is at the above link.

For this feature, we will attempt to estimate the adjusted gross income (AGI) for each of the zip codes. The data file contains many columns; however, you will only use the following:

- **STATE** - The state (e.g., MO)
- **zipcode** - The zipcode (e.g. 63017)
- **agi_stub** - Six different brackets of annual income (1 through 6)
- **N1** - The number of tax returns for each of the agi_stubs

Note, the file will have six rows for each zip code, for each of the agi_stub brackets. You can skip zip codes with 0 or 99999.

We will create an output CSV with these columns; however, only one row per zip code. Calculate a weighted average of the income brackets. For example, the following six rows are present for 63017:

zipcode	agi_stub	N1
--	--	--
63017	1	4710
63017	2	2780
63017	3	2130
63017	4	2010
63017	5	5240
63017	6	3510

We must combine these six rows into one. For privacy reasons, AGI's are broken out into 6 buckets. We need to combine the buckets and estimate the actual AGI of a zipcode. To do this, consider the values for N1:

- 1 = 1 to 25,000
- 2 = 25,000 to 50,000
- 3 = 50,000 to 75,000
- 4 = 75,000 to 100,000
- 5 = 100,000 to 200,000
- 6 = 200,000 or more

The median of each of these ranges is approximately:

- 1 = 12,500
- 2 = 37,500
- 3 = 62,500
- 4 = 87,500
- 5 = 112,500
- 6 = 212,500

Using this you can estimate 63017's average AGI as:

```
>>> totalCount = 4710 + 2780 + 2130 + 2010 + 5240 + 3510
>>> totalAGI = 4710 * 12500 + 2780 * 37500 + 2130 * 62500
    + 2010 * 87500 + 5240 * 112500 + 3510 * 212500
>>> print(totalAGI / totalCount)

88689.89205103042
```

We begin by reading in the government data.

Code

```
import pandas as pd

df=pd.read_csv('https://www.irs.gov/pub/irs-soi/16zpallagi.csv')
```

First, we trim all zip codes that are either 0 or 99999. We also select the three fields that we need.

Code

```
df=df.loc[(df['zipcode']!=0) & (df['zipcode']!=99999),
          ['STATE','zipcode','agi_stub','N1']]

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

Output

	STATE	zipcode	agi_stub	N1
6	AL	35004	1	1510
7	AL	35004	2	1410
8	AL	35004	3	950
9	AL	35004	4	650
10	AL	35004	5	630
...	...	...	...	...
179785	WY	83414	2	40
179786	WY	83414	3	40
179787	WY	83414	4	0
179788	WY	83414	5	40
179789	WY	83414	6	30

We replace all of the **agi_stub** values with the correct median values with the **map** function.

Code

```
medians = {1:12500,2:37500,3:62500,4:87500,5:112500,6:212500}
df['agi_stub']=df.agi_stub.map(medians)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)
display(df)
```

Output

	STATE	zipcode	agi_stub	N1
6	AL	35004	12500	1510
7	AL	35004	37500	1410
8	AL	35004	62500	950
9	AL	35004	87500	650
10	AL	35004	112500	630
...	...	...	...	...
179785	WY	83414	37500	40
179786	WY	83414	62500	40
179787	WY	83414	87500	0
179788	WY	83414	112500	40
179789	WY	83414	212500	30

Next, we group the data frame by zip code.

Code

```
groups = df.groupby(by='zipcode')
```

The program applies a lambda is applied across the groups, and then calculates the AGI estimate.

Code

```
df = pd.DataFrame(groups.apply(
    lambda x:sum(x['N1']*x['agi_stub'])/sum(x['N1']))) \
    .reset_index()
```

Code

```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

Output

	zipcode	0
0	1001	52895.322940
1	1002	64528.451001
2	1003	15441.176471
3	1005	54694.092827
4	1007	63654.353562
...	...	...
29867	99921	48042.168675
29868	99922	32954.545455
29869	99925	45639.534884
29870	99926	41136.363636
29871	99929	45911.214953

We can now rename the new agi_estimate column.

Code

```
df.columns = ['zipcode','agi_estimate']
```

Code

```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df)
```

Output

	zipcode	agi_estimate
0	1001	52895.322940
1	1002	64528.451001
2	1003	15441.176471
3	1005	54694.092827
4	1007	63654.353562
...	...	...
29867	99921	48042.168675
29868	99922	32954.545455
29869	99925	45639.534884
29870	99926	41136.363636
29871	99929	45911.214953

Finally, we check to see that our zip code of 63017 got the correct value.

Code

```
df[ df['zipcode']==63017 ]
```

Output

2.5 Part 2.5: Feature Engineering

Feature engineering is an essential part of machine learning. For now, we will manually engineer features. However, later in this course, we will see some techniques for automatic feature engineering.

2.5.1 Calculated Fields

It is possible to add new fields to the data frame that your program calculates from the other fields. We can create a new column that gives the weight in kilograms. The equation to calculate a metric weight, given weight in pounds, is:

$m_{(kg)} = m_{(lb)} \times 0.45359237$

The following Python code performs this transformation:

Code

```
import os
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

df.insert(1, 'weight_kg', (df['weight'] * 0.45359237).astype(int))
pd.set_option('display.max_columns', 6)
pd.set_option('display.max_rows', 5)
df
```

Output

	mpg	weight_kg	cylinders	...	year	origin	name
0	18.0	1589	8	...	70	1	chevrolet chevelle malibu
1	15.0	1675	8	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	28.0	1190	4	...	82	1	ford ranger
397	31.0	1233	4	...	82	1	chevy s-10

2.5.2 Google API Keys

Sometimes you will use external API's to obtain data. The following examples show how to use the Google API keys to encode addresses for use with neural networks. To use these, you will need your own Google API key. The key I have below is not a real key; you need to put your own in there. Google will ask for a credit card, but unless you use a huge number of lookups, there will be no actual cost. YOU ARE NOT required to get a Google API key for this class; this only shows you how. If you would like to get a Google API key, visit this site and obtain one for **geocode**.

You can obtain your own key from this link: Google API Keys.

Code

```
GOOGLE_KEY = 'REPLACE WITH YOUR GOOGLE API KEY'
```

2.5.3 Other Examples: Dealing with Addresses

Addresses can be difficult to encode into a neural network. There are many different approaches, and you must consider how you can transform the address into something more meaningful. Map coordinates can be a good approach. latitude and longitude can be a useful encoding. Thanks to the power of the Internet, it is relatively easy to transform an address into its latitude and longitude values. The following code determines the coordinates of Washington University:

Code

```
import requests

address = "1 Brookings Dr, St. Louis, MO 63130"

response = requests.get(
    'https://maps.googleapis.com/maps/api/geocode/json?key={}&address={}' \
    .format(GOOGLE_KEY, address))

resp_json_payload = response.json()

if 'error_message' in resp_json_payload:
    print(resp_json_payload['error_message'])
else:
    print(resp_json_payload['results'][0]['geometry']['location'])
```

Output

```
{'lat': 38.648238, 'lng': -90.30487459999999}
```

If latitude and longitude are fed into the neural network as two features, they might not be overly helpful. These two values would allow your neural network to cluster locations on a map. Sometimes cluster locations on a map can be useful. Figure 2.2 shows the percentage of the population that smokes in the USA by state.

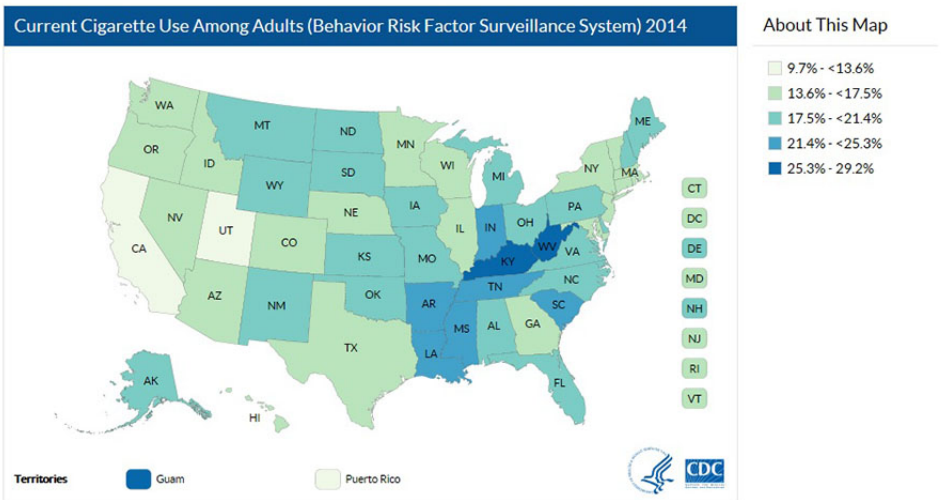


Figure 2.2: Smokers by State

The above map shows that certain behaviors, like smoking, can be clustered by the global region.

However, often you will want to transform the coordinates into distances. It is reasonably easy to estimate the distance between any two points on Earth by using the great circle distance between any two points on a sphere:

The following code implements this formula:

$$\Delta\sigma = \arccos\left(\sin\phi_1 \cdot \sin\phi_2 + \cos\phi_1 \cdot \cos\phi_2 \cdot \cos(\Delta\lambda)\right)$$

$$d = r, \Delta\sigma$$

Code

```
from math import sin, cos, sqrt, atan2, radians

URL='https://maps.googleapis.com' + \
    '/maps/api/geocode/json?key={}&address={}'

# Distance function
def distance_lat_lng(lat1,lng1,lat2,lng2):
    # approximate radius of earth in km
    R = 6373.0
```

```
    # degrees to radians (lat/lon are in degrees)
    lat1 = radians(lat1)
    lng1 = radians(lng1)
    lat2 = radians(lat2)
    lng2 = radians(lng2)

    dlng = lng2 - lng1
    dlat = lat2 - lat1

    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlng / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    return R * c

# Find lat lon for address
def lookup_lat_lng(address):
    response = requests.get( \
        URL) \
        .format(GOOGLE_KEY,address))
    json = response.json()
    if len(json['results']) == 0:
        print("Can't find: {}".format(address))
        return 0,0
    map = json['results'][0]['geometry']['location']
    return map['lat'],map['lng']


# Distance between two locations

import requests

address1 = "1 Brookings Dr, St. Louis, MO 63130"
address2 = "3301 College Ave, Fort Lauderdale, FL 33314"

lat1, lng1 = lookup_lat_lng(address1)
lat2, lng2 = lookup_lat_lng(address2)

print("Distance, St. Louis, MO to Ft. Lauderdale, FL: {} km".format(
        distance_lat_lng(lat1,lng1,lat2,lng2)))
```

Output

```
Distance, St. Louis, MO to Ft. Lauderdale, FL: 1685.530517973114 km
```

Distances can be a useful means to encode addresses. It would help if you considered what distance might be helpful for your dataset. Consider:

- Distance to a major metropolitan area
- Distance to a competitor
- Distance to a distribution center
- Distance to a retail outlet

The following code calculates the distance between 10 universities and Washington University in St. Louis:

Code

```
# Encoding other universities by their distance to Washington University

schools = [
    ["Princeton University, Princeton, NJ 08544", 'Princeton'],
    ["Massachusetts Hall, Cambridge, MA 02138", 'Harvard'],
    ["5801 S Ellis Ave, Chicago, IL 60637", 'University of Chicago'],
    ["Yale, New Haven, CT 06520", 'Yale'],
    ["116th St & Broadway, New York, NY 10027", 'Columbia University'],
    ["450 Serra Mall, Stanford, CA 94305", 'Stanford'],
    ["77 Massachusetts Ave, Cambridge, MA 02139", 'MIT'],
    ["Duke University, Durham, NC 27708", 'Duke University'],
    ["University of Pennsylvania, Philadelphia, PA 19104",
        'University of Pennsylvania'],
    ["Johns Hopkins University, Baltimore, MD 21218", 'Johns Hopkins']
]

lat1, lng1 = lookup_lat_lng("1 Brookings Dr, St. Louis, MO 63130")

for address, name in schools:
    lat2,lng2 = lookup_lat_lng(address)
    dist = distance_lat_lng(lat1,lng1,lat2,lng2)
    print("School '{}', distance to wustl is: {}".format(name,dist))
```

Output

```
School 'Princeton', distance to wustl is: 1354.4748428037537
School 'Harvard', distance to wustl is: 1670.6358699966058
School 'University of Chicago', distance to wustl is:
418.07123096093096
School 'Yale', distance to wustl is: 1508.209168740192
School 'Columbia University', distance to wustl is: 1418.2512902029155
School 'Stanford', distance to wustl is: 2780.7830466634337
School 'MIT', distance to wustl is: 1672.4354422735219
School 'Duke University', distance to wustl is: 1046.7924543575177
School 'University of Pennsylvania', distance to wustl is:
1307.1873732319766
School 'Johns Hopkins', distance to wustl is: 1184.3754484499111
```

Chapter 3

Introduction to TensorFlow

3.1 Part 3.1: Deep Learning and Neural Network Introduction

Neural networks were one of the first machine learning models. Their popularity has fallen twice and is now on its third rise. Deep learning implies the use of neural networks. The "deep" in deep learning refers to a neural network with many hidden layers. Because neural networks have been around for so long, they have quite a bit of baggage. Researchers have created many different training algorithms, activation/transfer functions, and structures over the years. This course is only concerned with the latest, most current state of the art techniques for deep neural networks. I am not going to spend any time discussing the history of neural networks. If you would like to learn about some of the more classic structures of neural networks, there are several modules dedicated to this in your coursebook. For the latest technology, I wrote an article for the Society of Actuaries on deep learning as the third generation of neural networks.

Neural networks accept input and produce output. The input to a neural network is called the feature vector. The size of this vector is always a fixed length. Changing the size of the feature vector means recreating the entire neural network. Though the feature vector is called a "vector," this is not always the case. A vector implies a 1D array. Historically the input to a neural network was always 1D. However, with modern neural networks, you might see input data, such as:

- **1D vector** - Classic input to a neural network, similar to rows in a spreadsheet. Common in predictive modeling.
- **2D Matrix** - Grayscale image input to a convolutional neural network (CNN).
- **3D Matrix** - Color image input to a convolutional neural network (CNN).
- **nD Matrix** - Higher-order input to a CNN.

Before CNN's, programs sent the image input to a neural network by merely squashing the image matrix into a long array by placing the image's rows side-by-side. CNNs are different, as the nD matrix passes through the neural network layers.

Initially, this course will focus upon 1D input to neural networks. However, later sessions will focus more heavily upon higher dimension input.

Dimensions The term dimension can be confusing in neural networks. In the sense of a 1D input vector, dimension refers to how many elements are in that 1D array. For example, a neural network with ten input neurons has ten dimensions. However, now that we have CNN's, the input has dimensions too. The input to the neural network will *usually* have 1, 2, or 3 dimensions. Four or more dimensions are unusual. You might have a 2D input to a neural network that has 64x64 pixels. This configuration would result in 4,096 input neurons. This network is either 2D or 4,096D, depending on which set of dimensions you are referencing.

3.1.1 Classification or Regression

Like many models, neural networks can function in classification or regression:

- **Regression** - You expect a number as your neural network's prediction.
- **Classification** - You expect a class/category as your neural network's prediction.

A classification and regression neural network is shown by Figure 3.1.

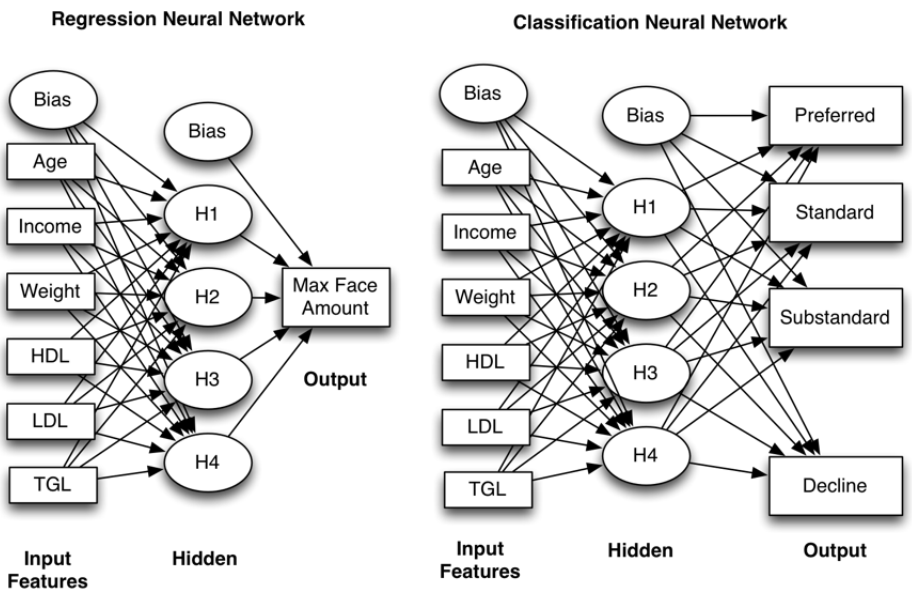


Figure 3.1: Neural Network Classification and Regression

Notice that the output of the regression neural network is numeric, and the output of the classification is a class. Regression, or two-class classification, networks always have a single output. Classification neural networks have an output neuron for each category.

3.1.2 Neurons and Layers

Most neural network structures use some type of neuron. Many different kinds of neural networks exist, and programmers introduce experimental neural network structures all the time. Consequently, it is not possible to cover every neural network architecture. However, there are some commonalities among neural network implementations. An algorithm that is called a neural network will typically be composed of individual, interconnected units even though these units may or may not be called neurons. The name for a neural network processing unit varies among the literature sources. It could be called a node, neuron, or unit.

A diagram shows the abstract structure of a single artificial neuron in Figure 3.2.

The artificial neuron receives input from one or more sources that may be other neurons or data fed into the network from a computer program. This input is usually floating-point or binary. Often binary input is encoded to floating-point by representing true or false as 1 or 0. Sometimes the program also depicts the binary input as using a bipolar system with true as one and false as -1.

An artificial neuron multiplies each of these inputs by a weight. Then it adds these multiplications and passes this sum to an activation function. Some neural networks do not use an activation function. The following equation summarizes the calculated output of a neuron:

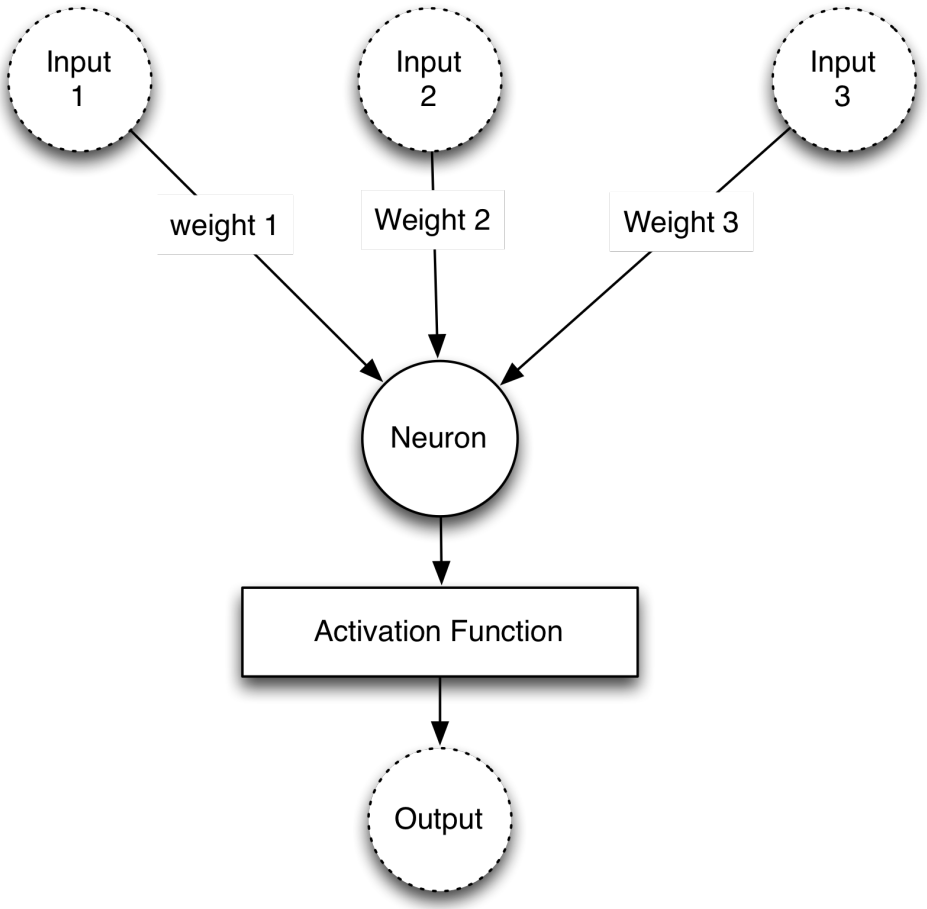


Figure 3.2: An Artificial Neuron

$f(x, w) = \phi(\sum_i (\theta_i \cdot x_i))$

In the above equation, the variables x and θ represent the input and weights of the neuron. The variable i corresponds to the number of weights and inputs. You must always have the same number of weights as inputs. The neural network multiplies each weight by its respective input and feeds the products of these multiplications into an activation function that is denoted by the Greek letter ϕ (phi). This process results in a single output from the neuron.

The above neuron has two inputs plus the bias as a third. This neuron might accept the following input feature vector:

$[1, 2]$

Because a bias neuron is present, the program should append the value of one, as follows:

$[1, 2, 1]$

The weights for a 3-input layer (2 real inputs + bias) will always have additional weight, for the bias. A weight vector might be:

$[0.1, 0.2, 0.3]$

To calculate the summation, perform the following:

$0.1\mathit{1} + \mathit{0.2}\mathit{2} + 0.3 * 1 = 0.8$

The program passes a value of 0.8 to the ϕ (phi) function, which represents the activation function.

The above figure shows the structure with just one building block. You can chain together many artificial neurons to build an artificial neural network (ANN). Think of the artificial neurons as building blocks for which the input and output circles are the connectors. Figure 3.3 shows an artificial neural network composed of three neurons:

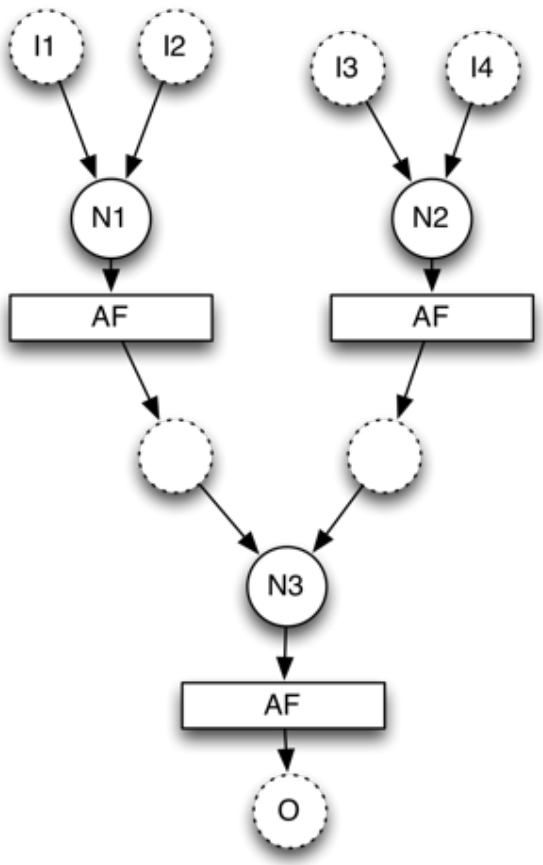


Figure 3.3: Three Neuron Neural Network

The above diagram shows three interconnected neurons. This representation is essentially this figure, minus a few inputs, repeated three times, and then connected. It also has a total of four inputs and a single output. The output of neurons **N1** and **N2** feed **N3** to produce the output O. To calculate the output for this network; we perform the previous equation three times. The first two times calculate **N1** and **N2**, and the third calculation uses the output of **N1** and **N2** to calculate **N3**.

Neural network diagrams do not typically show the level of detail seen in the previous figure. To simplify the chart, we can omit the activation functions and intermediate outputs, and this process results in Figure 3.4.

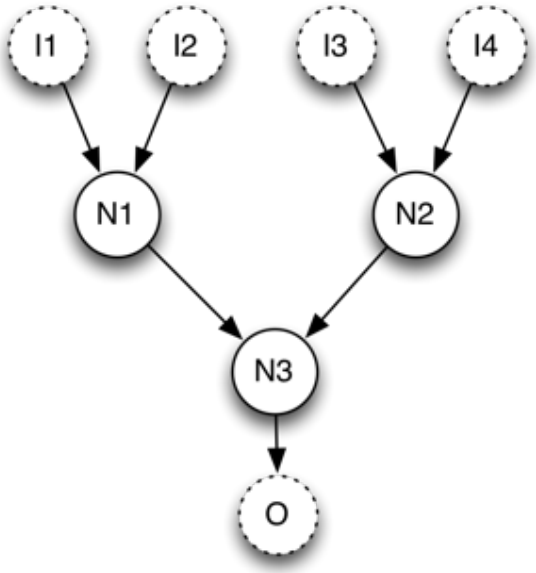


Figure 3.4: Three Neuron Neural Network

Looking at the previous figure, you can see two additional components of neural networks. First, consider the graph represents the inputs and outputs as abstract dotted line circles. The input and output could be parts of a more extensive neural network. However, the input and output are often a particular type of neuron that accepts data from the computer program using the neural network, and the output neurons return a result to the program. This type of neuron is called an input neuron. We will discuss these neurons in the next section. This figure shows the neurons arranged in layers. The input neurons are the first layer, the **N1** and **N2** neurons create the second layer, the third layer contains **N3**, and the fourth layer has O. Most neural networks arrange neurons into layers.

The neurons that form a layer share several characteristics. First, every neuron in a layer has the same activation function. However, the activation functions employed be each layer may be different. Each of the layers fully connects to the next layer. In other words, every neuron in one layer has a connection to neurons in the previous layer. The former figure is not fully connected. Several layers are missing connections. For example, **I1** and **N2** do not connect. The next neural network in Figure 3.5 is fully connected and has an additional layer.

In this figure, you see a fully connected, multilayered neural network. Networks, such as this one, will always have an input and output layer. The hidden layer structure determines the name of the network architecture. The network in this figure is a two-hidden-layer network. Most networks will have between zero and two hidden layers. Unless you have implemented deep learning strategies, networks with more than two hidden layers are rare.

You might also notice that the arrows always point downward or forward from the input to the output. This type of neural network is called a feedforward neural network. Later in this course, we will see recurrent neural networks that form inverted loops among the neurons.

3.1.3 Types of Neurons

In the last section, we briefly introduced the idea that different types of neurons exist. Now we will explain all the neuron types described in the course. Not every neural network will use every kind of neuron. It is

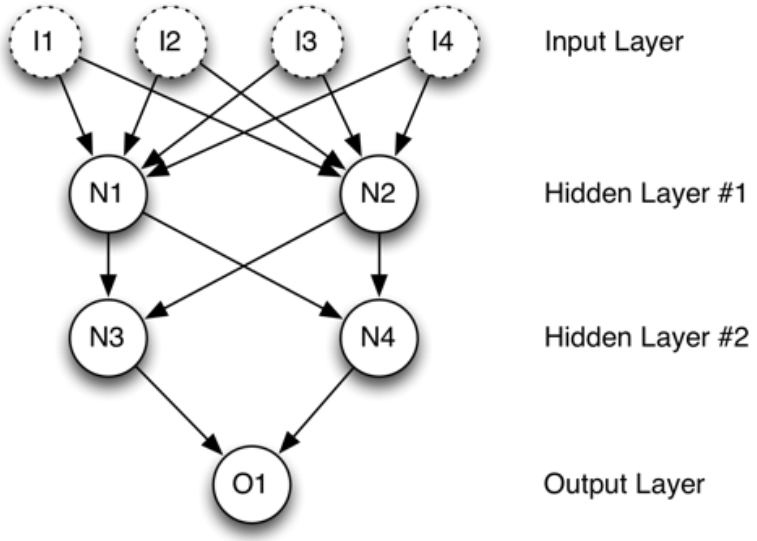


Figure 3.5: Fully Connected Neural Network Diagram

also possible for a single neuron to fill the role of several different neuron types.

There are usually four types of neurons in a neural network:

- **Input Neurons** - Each input neuron is mapped to one element in the feature vector.
- **Hidden Neurons** - Hidden neurons allow the neural network to be abstract and process the input into the output.
- **Output Neurons** - Each output neuron calculates one part of the output.
- **Context Neurons** - Holds state between calls to the neural network to predict.
- **Bias Neurons** - Work similar to the y-intercept of a linear equation.

These neurons are grouped into layers:

- **Input Layer** - The input layer accepts feature vectors from the dataset. Input layers usually have a bias neuron.
- **Output Layer** - The output from the neural network. The output layer does not have a bias neuron.
- **Hidden Layers** - Layers that occur between the input and output layers. Each hidden layer will usually have a bias neuron.

3.1.4 Input and Output Neurons

Nearly every neural network has input and output neurons. The input neurons accept data from the program for the network. The output neuron provides processed data from the network back to the program. The program will group these input and output neurons into separate layers called the input and output layer. The program normally represents the input to a neural network as an array or vector. The number of elements contained in the vector must be equal to the number of input neurons. For example, a neural network with three input neurons might accept the following input vector:

$[0.5, 0.75, 0.2]$

Neural networks typically accept floating-point vectors as their input. Likewise, neural networks will output a vector with length equal to the number of output neurons. The output will often be a single value from a single output neuron. To be consistent, we will represent the output of a single output neuron network as a single-element vector.

Notice that input neurons do not have activation functions. As demonstrated earlier, input neurons are little more than placeholders. The input is simply weighted and summed. Furthermore, the size of the input and output vectors for the neural network will be the same if the neural network has neurons that are both input and output.

3.1.5 Hidden Neurons

Hidden neurons have two essential characteristics. First, hidden neurons only receive input from other neurons, such as input or other hidden neurons. Second, hidden neurons only output to other neurons, such as output or other hidden neurons. Hidden neurons help the neural network understand the input, and they form the output. However, these hidden layers do not directly process to the incoming data or the eventual output. Programmers often group hidden neurons into fully connected hidden layers.

A common question for programmers concerns the number of hidden neurons in a network. Since the answer to this question is complex, more than one section of the course will include a relevant discussion of the number of hidden neurons. Before deep learning, researchers generally suggested that anything more than a single hidden layer is excessive (Hornik, 1991). Researchers have proven that a single-hidden-layer neural network can function as a universal approximator. In other words, this network should be able to learn to produce (or approximate) any output from any input as long as it has enough hidden neurons in a single layer.

Another reason why researchers used to scoff at the idea of additional hidden layers is that these layers would impede the training of the neural network. Training refers to the process that determines good weight values. Before researchers introduced deep learning techniques, we simply did not have an efficient way to train a deep network, which are neural networks with a large number of hidden layers. Although a single-hidden-layer neural network can theoretically learn anything, deep learning facilitates a more complex representation of patterns in the data.

3.1.6 Bias Neurons

Programmers add bias neurons to neural networks to help them learn patterns. Bias neurons function like an input neuron that always produces a value of 1. Because the bias neurons have a constant output of 1, they are not connected to the previous layer. The value of 1, which is called the bias activation, can be set to values other than 1. However, 1 is the most common bias activation. Not all neural networks have bias neurons. Figure 3.6 shows a single-hidden-layer neural network with bias neurons:

The above network contains three bias neurons. Every level, except for the output layer, includes a single bias neuron. Bias neurons allow the program to shift the output of an activation function. We will see precisely how this shifting occurs later in the module when we discuss activation functions.

3.1.7 Context Neurons

Recurrent neural networks make use of context neurons to hold state. This type of neuron allows the neural network to maintain state. As a result, a given input may not always produce the same output. This inconsistency is similar to the workings of biological brains. Consider how context factors in your response when you hear a loud horn. If you hear the noise while you are crossing the street, you might startle, stop walking, and look in the direction of the horn. If you hear the horn while you are watching an action-adventure film in a movie theatre, you don't respond in the same way. Therefore, prior inputs give you the context for processing the audio input of a horn.

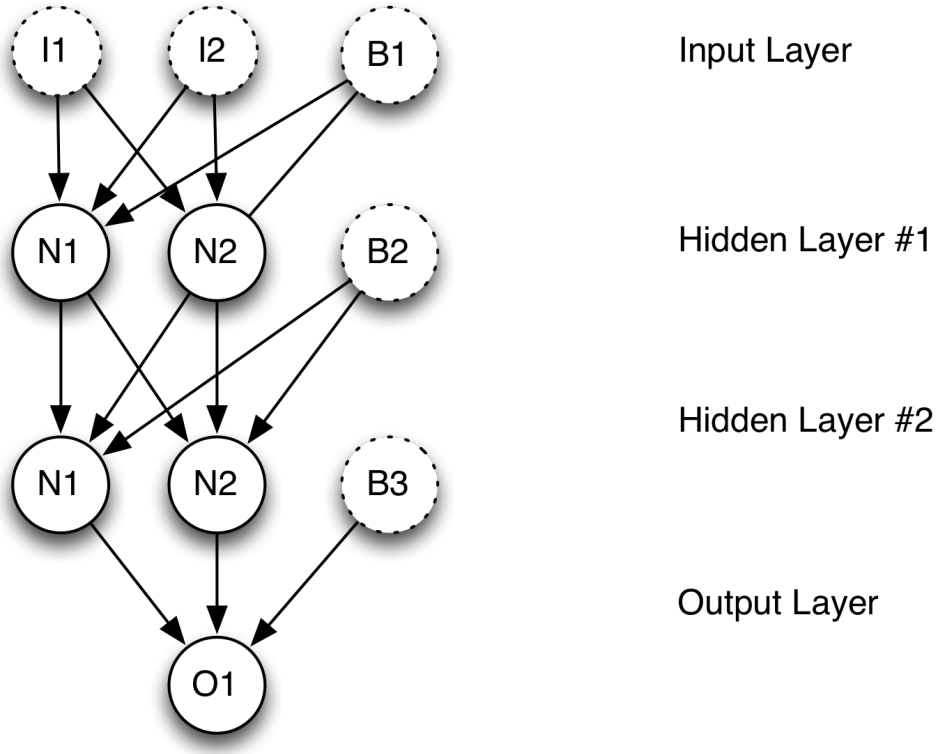


Figure 3.6: Neural Network with Bias Neurons

Time series is one application of context neurons. You might need to train a neural network to learn input signals to perform speech recognition or to predict trends in security prices. Context neurons are one way for neural networks to deal with time-series data. Figure 3.7 shows how a neural network might arrange context neurons:

This neural network has a single input and output neuron. Between the input and output layers are two hidden neurons and two context neurons. Other than the two context neurons, this network is the same as networks demonstrated previously.

Each context neuron holds a value that starts at 0 and always receives a copy of either hidden one or hidden two from the previous use of the network. The two dashed lines in this figure mean that the context neuron is a direct copy with no other weighting. The other lines indicate that the output is weighted by one of the six weight values listed above. The calculation of the output still occurs in the same way. The value of the output neuron would be the sum of all four inputs, multiplied by their weights, and applied to the activation function.

3.1.8 Other Neuron Types

The individual units that comprise a neural network are not always called neurons. Researchers will sometimes refer to these neurons as nodes, units, or summations. In later modules of this course, we will explore deep learning that utilizes Boltzmann machines to fill the role of neurons. You will almost always construct neural networks of weighted connections between these units.

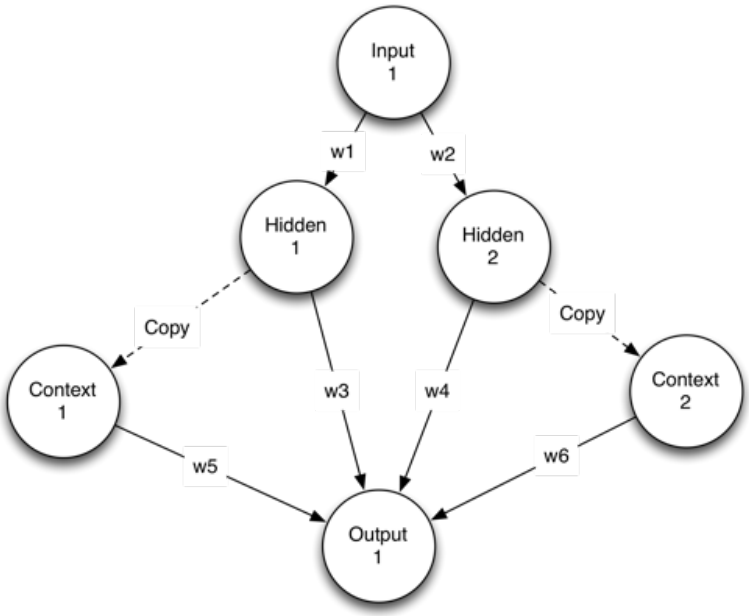


Figure 3.7: Neural Network with Context Neurons

3.1.9 Why are Bias Neurons Needed?

The activation functions seen in the previous section specifies the output of a single neuron. Together, the weight and bias of a neuron shape the output of the activation to produce the desired output. To see how this process occurs, consider the following equation. It represents a single-input sigmoid activation neural network.

$f(x, w, b) = \frac{1}{1+e^{-(wx+b)}}$

The x variable represents the single input to the neural network. The w and b variables specify the weight and bias of the neural network. The above equation is a combination of the weighted sum of the inputs and the sigmoid activation function. For this section, we will consider the sigmoid function because it demonstrates the effect that a bias neuron has.

The weights of the neuron allow you to adjust the slope or shape of the activation function. Figure 3.8 shows the effect on the output of the sigmoid activation function if the weight is varied:

The above diagram shows several sigmoid curves using the following parameters:

$f(x, 0.5, 0.0)$
$f(x, 1.0, 0.0)$
$f(x, 1.5, 0.0)$
$f(x, 2.0, 0.0)$

To produce the curves, we did not use bias, which is evident in the third parameter of 0 in each case. Using four weight values yields four different sigmoid curves in the above figure. No matter the weight, we always get the same value of 0.5 when x is 0 because all of the curves hit the same point when x is 0. We might need the neural network to produce other values when the input is near 0.5.

Bias does shift the sigmoid curve, which allows values other than 0.5 when x is near 0. Figure 3.9 shows

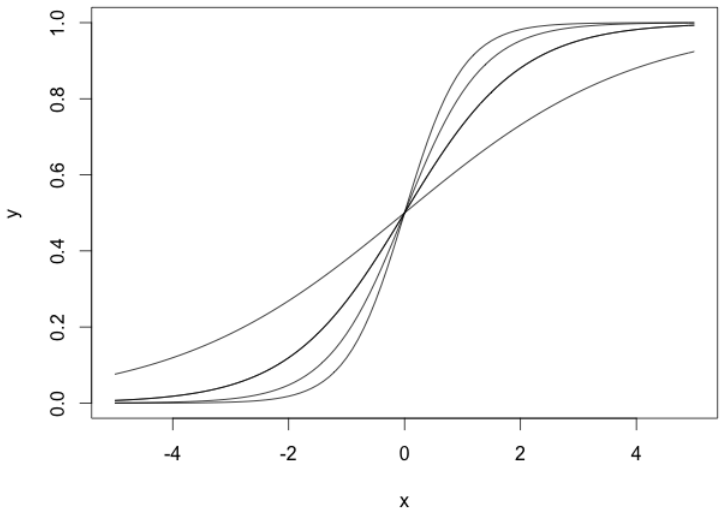


Figure 3.8: Neuron Weight Shifting

the effect of using a weight of 1.0 with several different biases:

The above diagram shows several sigmoid curves with the following parameters:

$f(x, 1.0, 1.0)$
$f(x, 1.0, 0.5)$
$f(x, 1.0, 1.5)$
$f(x, 1.0, 2.0)$

We used a weight of 1.0 for these curves in all cases. When we utilized several different biases, sigmoid curves shifted to the left or right. Because all the curves merge at the top right or bottom left, it is not a complete shift.

When we put bias and weights together, they produced a curve that created the necessary output from a neuron. The above curves are the output from only one neuron. In a complete network, the output from many different neurons will combine to produce intricate output patterns.

3.1.10 Modern Activation Functions

Activation functions, also known as transfer functions, are used to calculate the output of each layer of a neural network. Historically neural networks have used a hyperbolic tangent, sigmoid/logistic, or linear activation function. However, modern deep neural networks primarily make use of the following activation functions:

- **Rectified Linear Unit (ReLU)** - Used for the output of hidden layers.[10]
- **Softmax** - Used for the output of classification neural networks. Softmax Example
- **Linear** - Used for the output of regression neural networks (or 2-class classification).

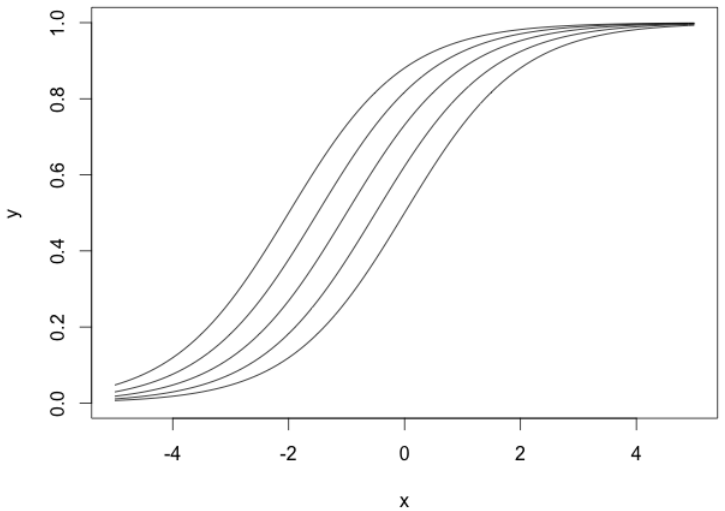


Figure 3.9: Neuron Bias Shifting

3.1.11 Linear Activation Function

The most basic activation function is the linear function because it does not change the neuron output at all. The following equation 1.2 shows how the program typically implements a linear activation function:

$\phi(x) = x$

As you can observe, this activation function simply returns the value that the neuron inputs passed to it. Figure 3.10 shows the graph for a linear activation function:

Regression neural networks, those that learn to provide numeric values, will usually use a linear activation function on their output layer. Classification neural networks, those that determine an appropriate class for their input, will often utilize a softmax activation function for their output layer.

3.1.12 Rectified Linear Units (ReLU)

Introduced in 2000 by Teh & Hinton, the rectified linear unit (ReLU) has seen very rapid adoption over the past few years. Before the ReLU activation function, the programmers generally regarded the hyperbolic tangent as the activation function of choice. Most current research now recommends the ReLU due to superior training results. As a result, most neural networks should utilize the ReLU on hidden layers and either softmax or linear on the output layer. The following equation shows the straightforward ReLU function:

$\phi(x) = \max(0, x)$

We will now examine why ReLU typically performs better than other activation functions for hidden layers. Part of the increased performance is because the ReLU activation function is a linear, non-saturating function. Unlike the sigmoid/logistic or the hyperbolic tangent activation functions, the ReLU does not saturate to -1, 0, or 1. A saturating activation function moves towards and eventually attains a value. The hyperbolic tangent function, for example, saturates to -1 as x decreases and one as x increases. Figure 3.11 shows the graph of the ReLU activation function:

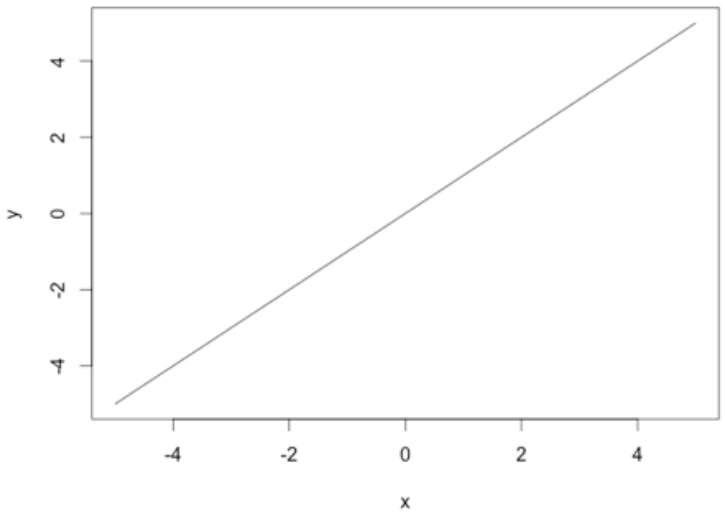


Figure 3.10: Linear Activation Function

Most current research states that the hidden layers of your neural network should use the ReLU activation.

3.1.13 Softmax Activation Function

The final activation function that we will examine is the softmax activation function. Along with the linear activation function, you can usually find the softmax function in the output layer of a neural network. Classification neural networks typically employ the softmax function. The neuron that has the highest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the output of the neural network to represent the probability that the input falls into each of the classes. Without the softmax, the neuron's outputs are simply numeric values, with the highest indicating the winning class.

To see how the program uses the softmax activation function, we will look at a typical neural network classification problem. The iris data set contains four measurements for 150 different iris flowers. Each of these flowers belongs to one of three species of iris. When you provide the measurements of a flower, the softmax function allows the neural network to give you the probability that these measurements belong to each of the three species. For example, the neural network might tell you that there is an 80% chance that the iris is setosa, a 15% probability that it is virginica, and only a 5% probability of versicolor. Because these are probabilities, they must add up to 100%. There could not be an 80% probability of setosa, a 75% probability of virginica, and a 20% probability of versicolor---this type of result would be nonsensical.

To classify input data into one of three iris species, you will need one output neuron for each of the three species. The output neurons do not inherently specify the probability of each of the three species. Therefore, it is desirable to provide probabilities that sum to 100%. The neural network will tell you the probability of a flower being each of the three species. To get the probability, use the softmax function in the following equation:

$$\phi_i(x) = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

In the above equation, i represents the index of the output neuron (o) that the program is calculating,

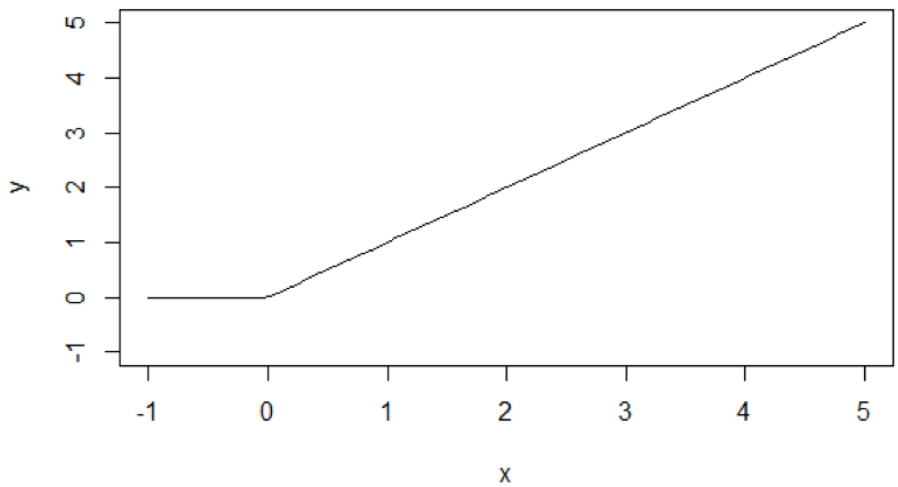


Figure 3.11: Rectified Linear Units (ReLU)

and j represents the indexes of all neurons in the group/level. The variable z designates the array of output neurons. It's important to note that the program calculates the softmax activation differently than the other activation functions in this module. When softmax is the activation function, the output of a single neuron is dependent on the other output neurons.

To see the softmax function in operation, refer to this Softmax example web site.

Consider a trained neural network that classifies data into three categories, such as the three iris species. In this case, you would use one output neuron for each of the target classes. Consider if the neural network were to output the following:

- **Neuron 1**: setosa: 0.9
- **Neuron 2**: versicolour: 0.2
- **Neuron 3**: virginica: 0.4

From the above output, we can see that the neural network considers the data to represent a setosa iris. However, these numbers are not probabilities. The 0.9 value does not represent a 90% likelihood of the data representing a setosa. These values sum to 1.5. For the program to treat them as probabilities, they must sum to 1.0. The output vector for this neural network is the following:

$[0.9, 0.2, 0.4]$

If you provide this vector to the softmax function it will return the following vector:

$[0.47548495534876745, 0.2361188410001125, 0.28839620365112]$

The above three values do sum to 1.0 and can be treated as probabilities. The likelihood of the data representing a setosa iris is 48% because the first value in the vector rounds to 0.48 (48%). You can calculate this value in the following manner:

$$sum = \exp(0.9) + \exp(0.2) + \exp(0.4) = 5.17283056695839$$
$$j_0 = \exp(0.9)/sum = 0.47548495534876745$$

$j_1 = \exp(0.2)/sum = 0.2361188410001125$
$j_2 = \exp(0.4)/sum = 0.28839620365112$

3.1.14 Classic Activation Functions

3.1.15 Step Activation Function

The step or threshold activation function is another simple activation function. Neural networks were originally called perceptrons. McCulloch & Pitts (1943) introduced the original perceptron and used a step activation function like the following equation:[28]

$$\phi(x) = \begin{cases} 1, \text{if x >= 0.5.} \\ 0, \text{otherwise.} \end{cases}$$

This equation outputs a value of 1.0 for incoming values of 0.5 or higher and 0 for all other values. Step functions, also known as threshold functions, only return 1 (true) for values that are above the specified threshold, as seen in Figure 3.12.

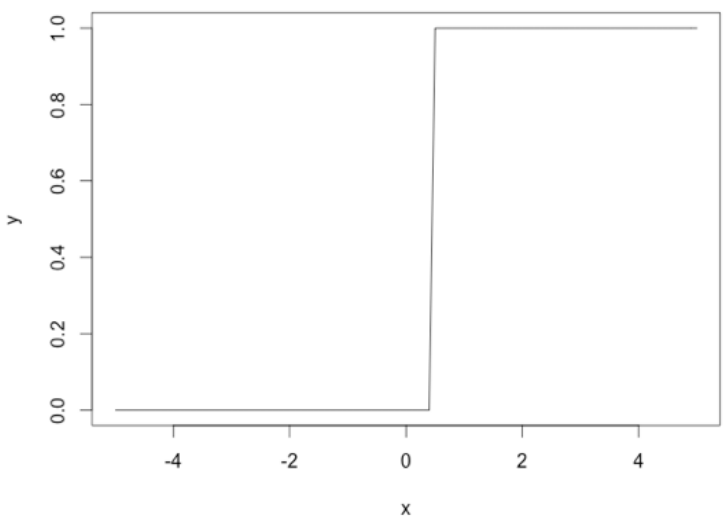


Figure 3.12: Step Activation Function

3.1.16 Sigmoid Activation Function

The sigmoid or logistic activation function is a very common choice for feedforward neural networks that need to output only positive numbers. Despite its widespread use, the hyperbolic tangent or the rectified linear unit (ReLU) activation function is usually a more suitable choice. We introduce the ReLU activation function later in this module. The following equation shows the sigmoid activation function:

$\phi(x) = \frac{1}{1+e^{-x}}$

Use the sigmoid function to ensure that values stay within a relatively small range, as seen in Figure 3.13:

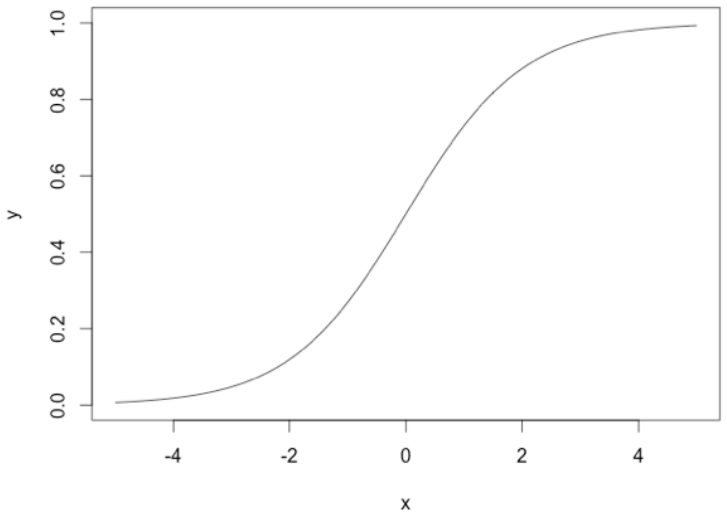


Figure 3.13: Sigmoid Activation Function

As you can see from the above graph, values can be forced to a range. Here, the function compressed values above or below 0 to the approximate range between 0 and 1.

3.1.17 Hyperbolic Tangent Activation Function

The hyperbolic tangent function is also a prevalent activation function for neural networks that must output values in the range between -1 and 1. This activation function is simply the hyperbolic tangent (tanh) function, as shown in the following equation:

$\phi(x) = \tanh(x)$

The graph of the hyperbolic tangent function has a similar shape to the sigmoid activation function, as seen in Figure 3.14.

The hyperbolic tangent function has several advantages over the sigmoid activation function.

3.1.18 Why ReLU?

Why is the ReLU activation function so popular? It was one of the critical improvements to neural networks that makes deep learning work.[30]Before deep learning, the sigmoid activation function was prevalent. We covered the sigmoid activation function earlier in this module. Frameworks, like Keras often train neural networks with gradient descent. For the neural network to make use of gradient descent, it is necessary to take the derivative of the activation function. The program must derive partial derivatives of each of the weights with respect to the error function. Figure 3.15 shows a derivative, the instantaneous rate of change.

The derivative of the sigmoid function is given here:

$\phi'(x) = \phi(x)(1 - \phi(x))$

Textbooks often give this derivative in other forms. We use the above form for computational efficiency. To see how we determined this derivative, refer to the following article.

We present the graph of the sigmoid derivative in Figure 3.16.

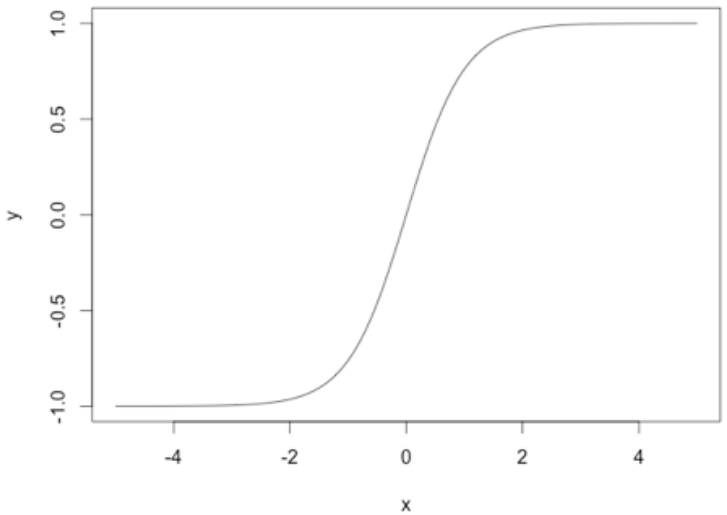


Figure 3.14: Hyperbolic Tangent Activation Function

The derivative quickly saturates to zero as x moves from zero. This is not a problem for the derivative of the ReLU, which is given here:

$$\phi'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

3.1.19 Module 3 Assignment

You can find the first assignment here: assignment 3

3.2 Part 3.2: Introduction to Tensorflow and Keras

TensorFlow is an open-source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for both research and production by different teams in many commercial Google products, such as speech recognition, Gmail, Google Photos, and search, many of which had previously used its predecessor DistBelief. TensorFlow was originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license on November 9, 2015.

- TensorFlow Homepage
- TensorFlow GitHib
- TensorFlow Google Groups Support
- TensorFlow Google Groups Developer Discussion
- TensorFlow FAQ

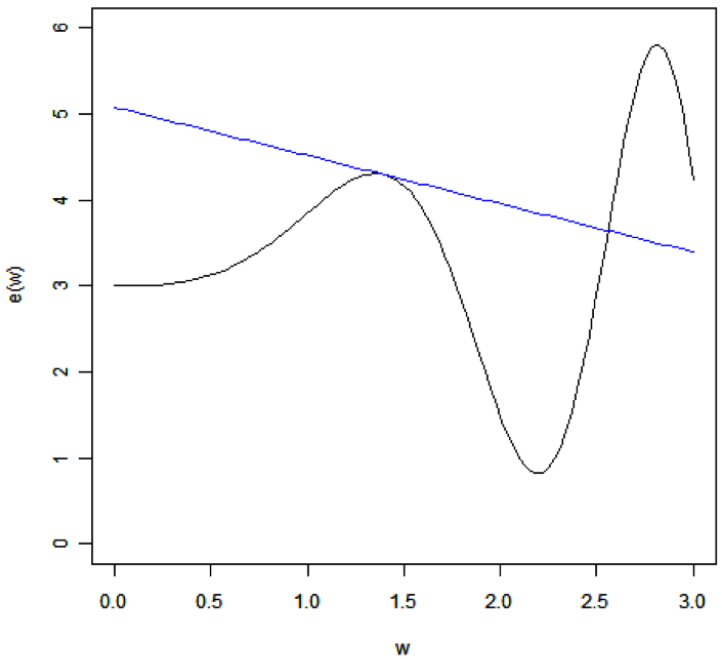


Figure 3.15: Derivative

3.2.1 Why TensorFlow

- Supported by Google
- Works well on Windows, Linux, and Mac
- Excellent GPU support
- Python is an easy to learn programming language
- Python is extremely popular in the data science community

3.2.2 Deep Learning Tools

TensorFlow is not the only game in town. The biggest competitor to TensorFlow/Keras is PyTorch. Listed below are some of the deep learning toolkits actively being supported:

- - Google's deep learning API. The focus of this class, along with Keras.
- - Also by Google, higher level framework that allows the use of TensorFlow, MXNet and Theano interchangeably.
- - PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing. It is primarily developed by Facebook's AI Research lab.

Other deep learning tools:

- Apache foundation's deep learning API. Can be used through Keras.

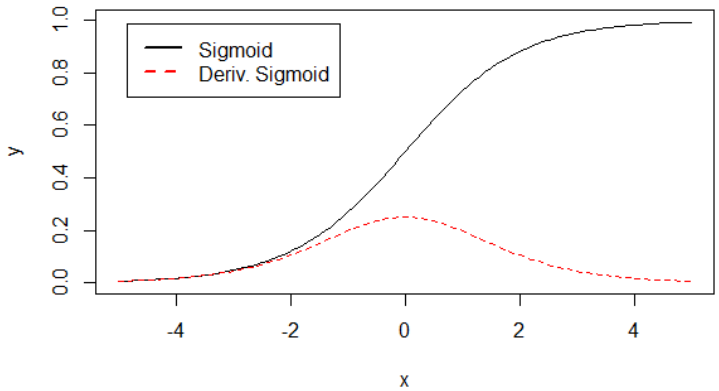


Figure 3.16: Sigmoid Derivative

- is used by Google DeepMind, the Facebook AI Research Group, IBM, Yandex and the Idiap Research Institute. It has been used for some of the most advanced deep learning projects in the world. However, it requires the LUAprogramming language. It is very advanced, but it is not mainstream. I have not worked with Torch (yet!).
- - **Baidu**'s deep learning API.
- - Java based. Supports all major platforms. GPU support in Java!
- - Microsoft. Support for Windows/Linux, command line only. Bindings for predictions for C#/Python. GPU support.
- - Java based. Supports all major platforms. Limited support for computer vision. No GPU support.

In my opinion, the two primary Python libraries for deep learning are PyTorch and Keras. Generally, PyTorch requires more lines of code to perform the deep learning applications presented in this course. This trait of PyTorch gives Keras an easier learning curve than PyTorch. However, if you are creating entirely new neural network structures, in a research setting, PyTorch can make for easier access to some of the low-level internals of deep learning.

3.2.3 Using TensorFlow Directly

Most of the time in the course, we will communicate with TensorFlow using Keras[7], which allows you to specify the number of hidden layers and create the neural network. TensorFlow is a low-level mathematics API, similar to Numpy. However, unlike Numpy, TensorFlow is built for deep learning. TensorFlow compiles these compute graphs into highly efficient C++/CUDA code.

3.2.4 TensorFlow Linear Algebra Examples

TensorFlow is a library for linear algebra. Keras is a higher-level abstraction for neural networks that you build upon TensorFlow. In this section, I will demonstrate some basic linear algebra that employs TensorFlow directly and does not make use of Keras. First, we will see how to multiply a row and column matrix.

Code

```
import tensorflow as tf
```

```
# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)

print(product)
print(float(product))
```

Output

```
tf.Tensor([[12.]], shape=(1, 1), dtype=float32)
12.0
```

This example multiplied two TensorFlow constant tensors. Next, we will see how to subtract a constant from a variable.

Code

```
import tensorflow as tf

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# Add an op to subtract 'a' from 'x'.  Run it and print the result
sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
# ==> [-2. -1.]
```

Output

```
tf.Tensor([-2. -1.], shape=(2,), dtype=float32)
[-2. -1.]
```

Of course, variables are only useful if their values can be changed. The program can accomplish this change in value by calling the assign function.

Code

```
x.assign([4.0, 6.0])
```

Output

```
<tf.Variable 'UnreadVariable' shape=(2,) dtype=float32,
numpy=array([4., 6.], dtype=float32)>
```

The program can now perform the subtraction with this new value.

Code

```
sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
```

Output

```
tf.Tensor([1. 3.], shape=(2,), dtype=float32)
[1. 3.]
```

In the next section, we will see a TensorFlow example that has nothing to do with neural networks.

3.2.5 TensorFlow Mandelbrot Set Example

Next, we examine another example where we use TensorFlow directly. To demonstrate that TensorFlow is mathematical and does not only provide neural networks, we will also first use it for a non-machine learning rendering task. The code presented here is capable of rendering a Mandelbrot set. Note, I based this example on a Mandelbrot example that I found here. I've updated the code slightly to comply with current versions of TensorFlow.

Code

```
# Import libraries for simulation
import tensorflow as tf
import numpy as np

# Imports for visualization
import PIL.Image
from io import BytesIO
from IPython.display import Image, display

def DisplayFractal(a, fmt='jpeg'):
  """Display an array of iteration counts as a
     colorful picture of a fractal."""
  a_cyclic = (6.28*a/20.0).reshape(list(a.shape)+[1])
  img = np.concatenate([10+20*np.cos(a_cyclic),
```

```
                                30+50*np.sin(a_cyclic),
                                155-80*np.cos(a_cyclic)], 2)
    img[a==a.max()] = 0
    a = img
    a = np.uint8(np.clip(a, 0, 255))
    f = BytesIO()
    PIL.Image.fromarray(a).save(f, fmt)
    display(Image(data=f.getvalue()))

# Use NumPy to create a 2D array of complex numbers

Y, X = np.mgrid[-1.3:1.3:0.005, -2:1:0.005]
Z = X+1j*Y

xs = tf.constant(Z.astype(np.complex64))
zs = tf.Variable(xs)
ns = tf.Variable(tf.zeros_like(xs, tf.float32))



# Operation to update the zs and the iteration count.
#
# Note: We keep computing zs after they diverge! This
#       is very wasteful! There are better, if a little
#       less simple, ways to do this.
#
for i in range(200):
    # Compute the new values of z: z^2 + x
    zs_ = zs*zs + xs

    # Have we diverged with this new value?
    not_diverged = tf.abs(zs_) < 4

    zs.assign(zs_),
    ns.assign_add(tf.cast(not_diverged, tf.float32))

DisplayFractal(ns.numpy())
```

Output

```
<IPython.core.display.Image object>
```

Mandlebrot render is both simple and infinitely complex at the same time. This view shows the entire Mandlebrot universe at the same time, as it is completely zoomed out. However, if you zoom in on any non-black portion of the plot, you will find infinite hidden complexity.

3.2.6 Introduction to Keras

Keras is a layer on top of Tensorflow that makes it much easier to create neural networks. Rather than define the graphs, as you see above, you set the individual layers of the network with a much more high-level API. Unless you are performing research into entirely new structures of deep neural networks, it is unlikely that you need to program TensorFlow directly.

For this class, we will use usually use TensorFlow through Keras, rather than direct TensorFlow

3.2.7 Simple TensorFlow Regression: MPG

This example shows how to encode the MPG dataset for regression. This dataset is slightly more complicated than Iris, because:

- Input has both numeric and categorical
- Input has missing values

This example uses functions defined above in this notepad, the "helpful functions". These functions allow you to build the feature vector for a neural network. Consider the following:

- Predictors/Inputs
  - Fill any missing inputs with the median for that column. Use **missing_median**.
  - Encode textual/categorical values with **encode_text_dummy**.
  - Encode numeric values with **encode_numeric_zscore**.
- Output
  - Discard rows with missing outputs.
  - Encode textual/categorical values with **encode_text_index**.
  - Do not encode output numeric values.
- Produce final feature vectors (x) and expected output (y) with **to_xy**.

To encode categorical values that are part of the feature vector, use the functions from above if the categorical value is the target (as was the case with Iris, use the same technique as Iris). The iris technique allows you to decode back to Iris text strings from the predictions.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']
```

```
# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

Output

```
Train on 398 samples
Epoch 1/100
398/398 - 0s - loss: 532928.3769
Epoch 2/100
398/398 - 0s - loss: 253903.1740
Epoch 3/100
398/398 - 0s - loss: 100463.9936
Epoch 4/100
398/398 - 0s - loss: 39376.4139
Epoch 5/100
398/398 - 0s - loss: 14699.5505
Epoch 6/100
398/398 - 0s - loss: 10070.2764
Epoch 7/100
398/398 - 0s - loss: 8767.4534

...

Epoch 99/100
398/398 - 0s - loss: 173.2709
Epoch 100/100
398/398 - 0s - loss: 173.1735
<tensorflow.python.keras.callbacks.History at 0x275f09bc408>
```

3.2.8 Introduction to Neural Network Hyperparameters

If you look at the above code, you will see that the neural network contains four layers. The first layer is the input layer because it contains the **input_dim** parameter that the programmer sets to be the number of

inputs that the dataset has. The network needs one input neuron for every column in the data set (including dummy variables).

There are also several hidden layers, with 25 and 10 neurons each. You might be wondering how the programmer chose these numbers. Selecting a hidden neuron structure is one of the most common questions about neural networks. Unfortunately, there is not a right answer. These are hyperparameters. They are settings that can affect neural network performance, yet there are not a clearly defined means of setting them.

In general, more hidden neurons mean more capability to fit complex problems. However, too many neurons can lead to overfitting and lengthy training times. Too few can lead to underfitting the problem and will sacrifice accuracy. Also, how many layers you have is another hyperparameter. In general, more layers allow the neural network to be able to perform more of its feature engineering and data preprocessing. But this also comes at the expense of training times and the risk of overfitting. In general, you will see that neuron counts start larger near the input layer and tend to shrink towards the output layer in a sort of triangular fashion.

Some techniques use machine learning to optimize these values. These will be discussed in Module 8.3.

3.2.9 Controlling the Amount of Output

The program produces one line of output for each training epoch. You can eliminate this output by setting the verbose setting of the fit command:

- **verbose=0** - No progress output (use with Jupyter if you do not want output)
- **verbose=1** - Display progress bar, does not work well with Jupyter
- **verbose=2** - Summary progress output (use with Jupyter if you want to know the loss at each epoch)

3.2.10 Regression Prediction

Next, we will perform actual predictions. The program assigns these predictions to the **pred** variable. These are all MPG predictions from the neural network. Notice that this is a 2D array? You can always see the dimensions of what Keras returns by printing out **pred.shape**. Neural networks can return multiple values, so the result is always an array. Here the neural network only returns one value per prediction (there are 398 cars, so 398 predictions). However, a 2D range is needed because the neural network has the potential of returning more than one value.

Code

```
pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

Output

```
Shape: (398, 1)
[[23.08987 ]
 [23.776747]
 [22.190557]
 [22.24273 ]
 [22.536469]
 [27.865118]
 [27.486967]
```

```
 [27.316473]
 [27.953611]
 [24.391483]]
```

We would like to see how good these predictions are. We know what the correct MPG is for each car, so we can measure how close the neural network was.

Code

```
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Final score (RMSE): {score}")
```

Output

```
Final score (RMSE): 13.157274080377979
```

The number printed above is the average amount that the predictions were above or below the expected output. We can also print out the first ten cars, with predictions and actual MPG.

Code

```
# Sample predictions
for i in range(10):
    print(f"{i+1}. Car name: {cars[i]}, MPG: {y[i]}, "
          + "predicted MPG: {pred[i]}")
```

Output

```
1. Car name: chevrolet chevelle malibu, MPG: 18.0, predicted MPG:
{pred[i]}
2. Car name: buick skylark 320, MPG: 15.0, predicted MPG: {pred[i]}
3. Car name: plymouth satellite, MPG: 18.0, predicted MPG: {pred[i]}
4. Car name: amc rebel sst, MPG: 16.0, predicted MPG: {pred[i]}
5. Car name: ford torino, MPG: 17.0, predicted MPG: {pred[i]}
6. Car name: ford galaxie 500, MPG: 15.0, predicted MPG: {pred[i]}
7. Car name: chevrolet impala, MPG: 14.0, predicted MPG: {pred[i]}
8. Car name: plymouth fury iii, MPG: 14.0, predicted MPG: {pred[i]}
9. Car name: pontiac catalina, MPG: 14.0, predicted MPG: {pred[i]}
10. Car name: amc ambassador dpl, MPG: 15.0, predicted MPG: {pred[i]}
```

3.2.11 Simple TensorFlow Classification: Iris

Classification is the process by which a neural network attempts to classify the input into one or more classes. The simplest way of evaluating a classification network is to track the percentage of training set items that were classified incorrectly. We typically score human results in this manner. For example, you might have taken multiple-choice exams in school in which you had to shade in a bubble for choices A, B, C, or D. If

you chose the wrong letter on a 10-question exam, you would earn a 90%. In the same way, we can grade computers; however, most classification algorithms do not merely choose A, B, C, or D. Computers typically report a classification as their percent confidence in each class. Figure 3.17 shows how a computer and a human might both respond to question number 1 on an exam.

Figure 3.17: Classification Neural Network Output

As you can see, the human test taker marked the first question as "B." However, the computer test taker had an 80% (0.8) confidence in "B" and was also somewhat sure with 10% (0.1) on "A." The computer then distributed the remaining points on the other two. In the simplest sense, the machine would get 80% of the score for this question if the correct answer were "B." The computer would get only 5% (0.05) of the points if the correct answer were "D."

What we just saw is a straightforward example of how to perform the Iris classification using TensorFlow. The iris.csv file is used, rather than using the built-in data that many of the Google examples require.

Make sure that you always run previous code blocks. If you run the code block below, without the code block above, you will get errors

Code

```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values


# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
```

```
model.add(Dense(y.shape[1],activation='softmax')) # Output

model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

Output

```
Train on 150 samples
Epoch 1/100
150/150 - 0s - loss: 1.1343
Epoch 2/100
150/150 - 0s - loss: 1.0067
Epoch 3/100
150/150 - 0s - loss: 0.9150
Epoch 4/100
150/150 - 0s - loss: 0.8430
Epoch 5/100
150/150 - 0s - loss: 0.7892
Epoch 6/100
150/150 - 0s - loss: 0.7348
Epoch 7/100
150/150 - 0s - loss: 0.6847

...

Epoch 99/100
150/150 - 0s - loss: 0.0852
Epoch 100/100
150/150 - 0s - loss: 0.0871
<tensorflow.python.keras.callbacks.History at 0x275f1d9bdc8>
```

Code

```
# Print out number of species found:

print(species)
```

Output

```
Index(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'],
dtype='object')
```

Now that you have a neural network training, we would like to be able to use it. The following code makes use of our neural network. Exactly like before, we will generate predictions. Notice that three values come back for each of the 150 iris flowers. There were three types of iris (Iris-setosa, Iris-versicolor, and Iris-virginica).

Code

```
pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

Output

```
Shape: (150, 3)
[[9.98257935e-01 1.74211117e-03 1.47250105e-08]
 [9.94762123e-01 5.23775769e-03 9.35751956e-08]
 [9.97034669e-01 2.96533993e-03 5.68744980e-08]
 [9.94459629e-01 5.54029271e-03 1.55113412e-07]
 [9.98531222e-01 1.46873493e-03 1.33291875e-08]
 [9.98098075e-01 1.90198515e-03 1.37531018e-08]
 [9.96991158e-01 3.00874189e-03 7.64488419e-08]
 [9.97346044e-01 2.65395525e-03 3.05425694e-08]
 [9.92665589e-01 7.33401440e-03 3.12981911e-07]
 [9.95905280e-01 4.09475202e-03 5.66798697e-08]]
```

If you would like to turn of scientific notation, the following line can be used:

Code

```
np.set_printoptions(suppress=True)
```

Now we see these values rounded up.

Code

```
print(y[0:10])
```

Output

```
[[1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]
 [1 0 0]]
```

Usually, the program considers the column with the highest prediction to be the prediction of the neural network. It is easy to convert the predictions to the expected iris species. The argmax function finds the index of the maximum prediction for each row.

Code

```
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
print(f"Predictions: {predict_classes}")
print(f"Expected: {expected_classes}")
```

Output

```
Predictions: [0 0 0 
```

Code

```
from sklearn.metrics import accuracy_score

correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

Output

```
Accuracy: 0.9733333333333334
```

The code below performs two ad hoc predictions. The first prediction is simply a single iris flower, and the second predicts two iris flowers. Notice that the argmax in the second prediction requires **axis=1**? Since we have a 2D array now, we must specify which axis to take the argmax over. The value **axis=1** specifies we want the max column index for each row.

Code

```
sample_flower = np.array( [[5.0,3.0,4.0,2.0]], dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred)
print(f"Predict that {sample_flower} is: {species[pred]}")
```

Output

```
[[0.00208851 0.19842853 0.799483  ]]
Predict that [[5. 3. 4. 2.]] is: Iris-virginica
```

You can also predict two sample flowers.

Code

```
sample_flower = np.array( [[5.0,3.0,4.0,2.0],[5.2,3.5,1.5,0.8]],\
        dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred,axis=1)
print(f"Predict that these two flowers {sample_flower} ")
print(f"are: {species[pred]}")
```

Output

```
[[0.00208851 0.19842838 0.79948306]
 [0.9900221  0.00997756 0.00000035]]
Predict that these two flowers [[5.  3.  4.  2. ]
 [5.2 3.5 1.5 0.8]]
```

```
are: Index(['Iris-virginica', 'Iris-setosa'], dtype='object')
```

3.3 Part 3.3: Saving and Loading a Keras Neural Network

Complex neural networks will take a long time to fit/train. It is helpful to be able to save these neural networks so that they can be reloaded later. A reloaded neural network will not require retraining. Keras provides three formats for neural network saving.

- **YAML** - Stores the neural network structure (no weights) in the YAML file format.
- **JSON** - Stores the neural network structure (no weights) in the JSON file format.
- **HDF5** - Stores the complete neural network (with weights) in the HDF5 file format. Do not confuse HDF5 with HDFS. They are different. We do not use HDFS in this class.

Usually you will want to save in HDF5.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
        'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
```

```
model.fit(x,y,verbose=2,epochs=100)

# Predict
pred = model.predict(x)

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Before save score (RMSE): {score}")

# save neural network structure to JSON (no weights)
model_json = model.to_json()
with open(os.path.join(save_path,"network.json"), "w") as json_file:
    json_file.write(model_json)

# save neural network structure to YAML (no weights)
model_yaml = model.to_yaml()
with open(os.path.join(save_path,"network.yaml"), "w") as yaml_file:
    yaml_file.write(model_yaml)

# save entire network to HDF5 (save everything, suggested)
model.save(os.path.join(save_path,"network.h5"))
```

Output

```
Train on 398 samples
Epoch 1/100
398/398 - 0s - loss: 21424.3914
Epoch 2/100
398/398 - 0s - loss: 5458.3984
Epoch 3/100
398/398 - 0s - loss: 1736.3177
Epoch 4/100
398/398 - 0s - loss: 1013.3963
Epoch 5/100
398/398 - 0s - loss: 827.7916
Epoch 6/100
398/398 - 0s - loss: 744.9987
Epoch 7/100
398/398 - 0s - loss: 724.7204

...

Epoch 99/100
398/398 - 0s - loss: 12.6923
Epoch 100/100
398/398 - 0s - loss: 12.7204
Before save score (RMSE): 3.6842460468319183
```

The code below sets up a neural network and reads the data (for predictions), but it does not clear the model directory or fit the neural network. The weights from the previous fit are used.

Now we reload the network and perform another prediction. The RMSE should match the previous one exactly if the neural network was really saved and reloaded.

Code

```
from tensorflow.keras.models import load_model
model2 = load_model(os.path.join(save_path,"network.h5"))
pred = model2.predict(x)
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"After load score (RMSE): {score}")
```

Output

```
After load score (RMSE): 3.6842460468319183
```

3.4 Part 3.4: Early Stopping in Keras to Prevent Overfitting

Overfitting occurs when a neural network is trained to the point that it begins to memorize rather than generalize, as demonstrated in Figure 3.18.

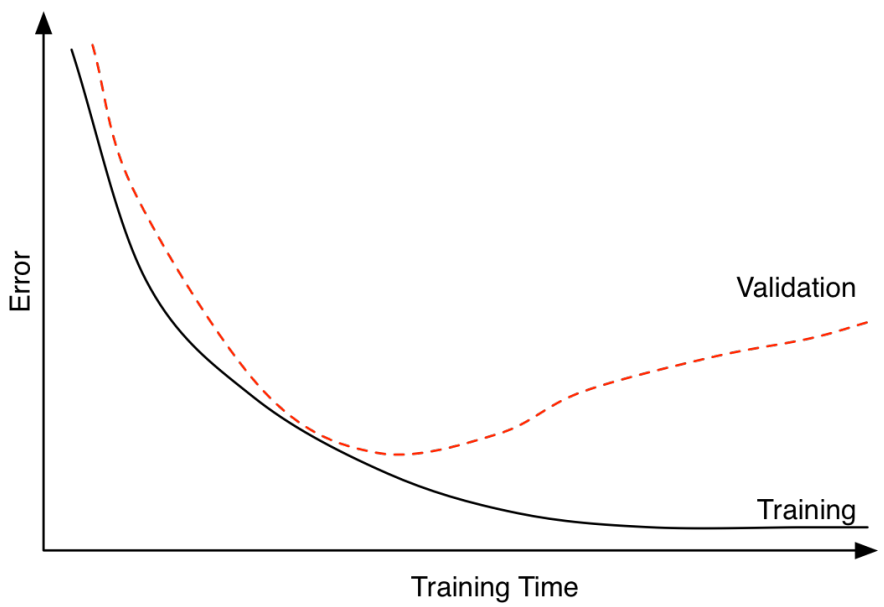


Figure 3.18: Training vs Validation Error for Overfitting

It is important to segment the original dataset into several datasets:

- **Training Set**
- **Validation Set**
- **Holdout Set**

There are several different ways that these sets can be constructed. The following programs demonstrate some of these.

The first method is a training and validation set. The training data are used to train the neural network until the validation set no longer improves. This attempts to stop at a near optimal training point. This method will only give accurate "out of sample" predictions for the validation set, this is usually 20% or so of the data. The predictions for the training data will be overly optimistic, as these were the data that the neural network was trained on. Figure 3.19 demonstrates how a dataset is divided.

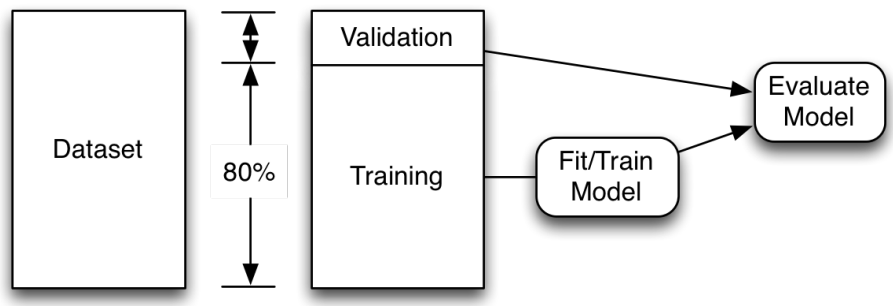


Figure 3.19: Training with a Validation Set

3.4.1 Early Stopping with Classification

Code

```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
```

```
y = dummies.values

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
        callbacks=[monitor],verbose=2,epochs=1000)
```

Output

```
Train on 112 samples, validate on 38 samples
Epoch 1/1000
112/112 - 0s - loss: 1.1940 - val_loss: 1.1126
Epoch 2/1000
112/112 - 0s - loss: 1.0545 - val_loss: 0.9984
Epoch 3/1000
112/112 - 0s - loss: 0.9533 - val_loss: 0.9130
Epoch 4/1000
112/112 - 0s - loss: 0.8823 - val_loss: 0.8365
Epoch 5/1000
112/112 - 0s - loss: 0.8243 - val_loss: 0.7619
Epoch 6/1000
112/112 - 0s - loss: 0.7592 - val_loss: 0.7059
Epoch 7/1000
112/112 - 0s - loss: 0.7142 - val_loss: 0.6644

...

Epoch 107/1000
Restoring model weights from the end of the best epoch.
112/112 - 0s - loss: 0.1001 - val_loss: 0.0869
Epoch 00107: early stopping
<tensorflow.python.keras.callbacks.History at 0x22a9ad34708>
```

There are a number of parameters that are specified to the **EarlyStopping** object.

- **min_delta** This value should be kept small. It simply means the minimum change in error to be registered as an improvement. Setting it even smaller will not likely have a great deal of impact.

- **patience** How long should the training wait for the validation error to improve?
- **verbose** How much progress information do you want?
- **mode** In general, always set this to "auto". This allows you to specify if the error should be minimized or maximized. Consider accuracy, where higher numbers are desired vs log-loss/RMSE where lower numbers are desired.
- **restore_best_weights** This should always be set to true. This restores the weights to the values they were at when the validation set is the highest. Unless you are manually tracking the weights yourself (we do not use this technique in this course), you should have Keras perform this step for you.

As you can see from above, the entire number of requested epochs were not used. The neural network training stopped once the validation set no longer improved.

Code

```
from sklearn.metrics import accuracy_score

pred = model.predict(x_test)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y_test,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

Output

```
Accuracy: 1.0
```

3.4.2 Early Stopping with Regression

The following code demonstrates how we can apply early stopping to a regression problem. The technique is similar to the early stopping for classification code that we just saw.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
```

```
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=5, verbose=1, mode='auto',
        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
        callbacks=[monitor], verbose=2,epochs=1000)
```

Output

```
Train on 298 samples, validate on 100 samples
Epoch 1/1000
298/298 - 0s - loss: 254618.1117 - val_loss: 104859.9187
Epoch 2/1000
298/298 - 0s - loss: 53735.2417 - val_loss: 10033.3467
Epoch 3/1000
298/298 - 0s - loss: 3456.0443 - val_loss: 2832.0205
Epoch 4/1000
298/298 - 0s - loss: 4912.1159 - val_loss: 5504.1926
Epoch 5/1000
298/298 - 0s - loss: 4154.7669 - val_loss: 2042.1780
Epoch 6/1000
298/298 - 0s - loss: 1411.5907 - val_loss: 1259.3724
Epoch 7/1000
298/298 - 0s - loss: 1189.8836 - val_loss: 1435.5145

...

Epoch 317/1000
Restoring model weights from the end of the best epoch.
298/298 - 0s - loss: 32.9764 - val_loss: 29.1071
Epoch 00317: early stopping
```

```
<tensorflow.python.keras.callbacks.History at 0x22a9acc8608>
```

Finally, we evaluate the error.

Code

```
# Measure RMSE error.  RMSE is common for regression.
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")
```

Output

```
Final score (RMSE): 5.291219300799398
```

3.5 Part 3.5: Extracting Weights and Manual Network Calculation

3.5.1 Weight Initialization

The weights of a neural network determine the output for the neural network. The process of training can adjust these weights so the neural network produces useful output. Most neural network training algorithms begin by initializing the weights to a random state. Training then progresses through a series of iterations that continuously improve the weights to produce better output.

The random weights of a neural network impact how well that neural network can be trained. If a neural network fails to train, you can remedy the problem by simply restarting with a new set of random weights. However, this solution can be frustrating when you are experimenting with the architecture of a neural network and trying different combinations of hidden layers and neurons. If you add a new layer, and the network's performance improves, you must ask yourself if this improvement resulted from the new layer or from a new set of weights. Because of this uncertainty, we look for two key attributes in a weight initialization algorithm:

- How consistently does this algorithm provide good weights?
- How much of an advantage do the weights of the algorithm provide?

One of the most common, yet least effective, approaches to weight initialization is to set the weights to random values within a specific range. Numbers between -1 and +1 or -5 and +5 are often the choice. If you want to ensure that you get the same set of random weights each time, you should use a seed. The seed specifies a set of predefined random weights to use. For example, a seed of 1000 might produce random weights of 0.5, 0.75, and 0.2. These values are still random; you cannot predict them, yet you will always get these values when you choose a seed of 1000.
Not all seeds are created equal. One problem with random weight initialization is that the random weights created by some seeds are much more difficult to train than others. In fact, the weights can be so bad that training is impossible. If you find that you cannot train a neural network with a particular weight set, you should generate a new set of weights using a different seed.

Because weight initialization is a problem, there has been considerable research around it. In this course we use the Xavier weight initialization algorithm, introduced in 2006 by Glorot & Bengio[9], produces good

weights with reasonable consistency. This relatively simple algorithm uses normally distributed random numbers.

To use the Xavier weight initialization, it is necessary to understand that normally distributed random numbers are not the typical random numbers between 0 and 1 that most programming languages generate. In fact, normally distributed random numbers are centered on a mean (μ, mu) that is typically 0. If 0 is the center (mean), then you will get an equal number of random numbers above and below 0. The next question is how far these random numbers will venture from 0. In theory, you could end up with both positive and negative numbers close to the maximum positive and negative ranges supported by your computer. However, the reality is that you will more likely see random numbers that are between 0 and three standard deviations from the center.

The standard deviation (σ, sigma) parameter specifies the size of this standard deviation. For example, if you specified a standard deviation of 10, then you would mainly see random numbers between -30 and +30, and the numbers nearer to 0 have a much higher probability of being selected.

The above figure illustrates that the center, which in this case is 0, will be generated with a 0.4 (40%) probability. Additionally, the probability decreases very quickly beyond -2 or +2 standard deviations. By defining the center and how large the standard deviations are, you are able to control the range of random numbers that you will receive.

The Xavier weight initialization sets all of the weights to normally distributed random numbers. These weights are always centered at 0; however, their standard deviation varies depending on how many connections are present for the current layer of weights. Specifically, Equation 4.2 can determine the standard deviation:

$$Var(W) = \frac{2}{n_{in}+n_{out}}$$

The above equation shows how to obtain the variance for all of the weights. The square root of the variance is the standard deviation. Most random number generators accept a standard deviation rather than a variance. As a result, you usually need to take the square root of the above equation. Figure 3.20 shows how one layer might be initialized.

This process is completed for each layer in the neural network.

3.5.2 Manual Neural Network Calculation

In this section we will build a neural network and analyze it down the individual weights. We will train a simple neural network that learns the XOR function. It is not hard to simply hand-code the neurons to provide an XOR function; however, for simplicity, we will allow Keras to train this network for us. We will just use 100K epochs on the ADAM optimizer. This is massive overkill, but it gets the result, and our focus here is not on tuning. The neural network is small. Two inputs, two hidden neurons, and a single output.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import numpy as np

# Create a dataset for the XOR function
x = np.array([
    [0,0],
    [1,0],
    [0,1],
    [1,1]
])
```

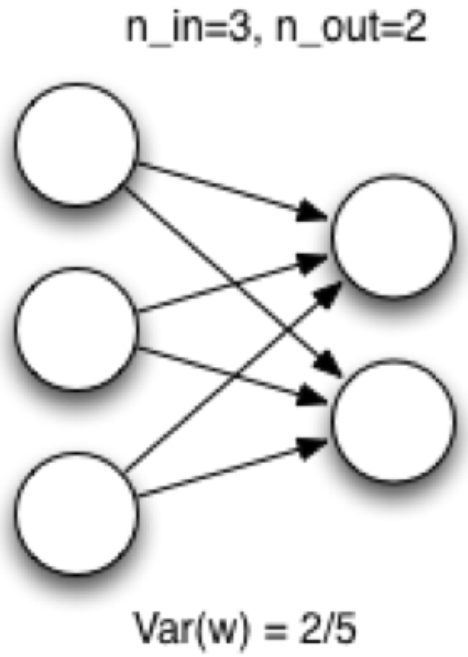


Figure 3.20: Xavier Weight Initialization

```
y = np.array([
    0,
    1,
    1,
    0
])

# Build the network
# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

done = False
cycle = 1

while not done:
    print("Cycle #{}".format(cycle))
    cycle+=1
    model = Sequential()
    model.add(Dense(2, input_dim=2, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(x,y,verbose=0,epochs=10000)
```

```
    # Predict
    pred = model.predict(x)

    # Check if successful.  It takes several runs with this
    # small of a network
    done = pred[0]<0.01 and pred[3]<0.01 and pred[1] > 0.9 \
        and pred[2] > 0.9
    print(pred)
```

Output

```
Cycle #1
[[0.49999997]
 [0.49999997]
 [0.49999997]
 [0.49999997]]
Cycle #2
[[0.33333334]
 [1.        ]
 [0.33333334]
 [0.33333334]]
Cycle #3
[[0.33333334]
 [1.        ]
 [0.33333334]
 [0.33333334]]
Cycle #4
[[0.]
 [1.]
 [1.]
 [0.]]
```

Code

```
pred[3]
```

Output

```
array([0.], dtype=float32)
```

The output above should have two numbers near 0.0 for the first and forth spots (input [[0,0]] and [[1,1]]). The middle two numbers should be near 1.0 (input [[1,0]] and [[0,1]]). These numbers are in scientific notation. Due to random starting weights, it is sometimes necessary to run the above through several cycles to get a good result.

Now that the neural network is trained, lets dump the weights.

Code

```
# Dump weights
for layerNum, layer in enumerate(model.layers):
    weights = layer.get_weights()[0]
    biases = layer.get_weights()[1]

    for toNeuronNum, bias in enumerate(biases):
        print(f'{layerNum}B -> L{layerNum+1}N{toNeuronNum}: {bias}')

    for fromNeuronNum, wgt in enumerate(weights):
        for toNeuronNum, wgt2 in enumerate(wgt):
            print(f'L{layerNum}N{fromNeuronNum} \
                -> L{layerNum+1}N{toNeuronNum} = {wgt2}')
```

Output

```
0B -> L1N0: 1.3025760914331386e-08
0B -> L1N1: -1.4192625741316078e-08
L0N0                    -> L1N0 = 0.659289538860321
L0N0                    -> L1N1 = -0.9533336758613586
L0N1                    -> L1N0 = -0.659289538860321
L0N1                    -> L1N1 = 0.9533336758613586
1B -> L2N0: -1.9757269598130733e-08
L1N0                    -> L2N0 = 1.5167843103408813
L1N1                    -> L2N0 = 1.0489506721496582
```

If you rerun this, you probably get different weights. There are many ways to solve the XOR function.

In the next section, we copy/paste the weights from above and recreate the calculations done by the neural network. Because weights can change with each training, the weights used for the below code came from this:

```
0B -> L1N0: -1.2913415431976318
0B -> L1N1: -3.021530048386012e-08
L0N0 -> L1N0 = 1.2913416624069214
L0N0 -> L1N1 = 1.1912699937820435
L0N1 -> L1N0 = 1.2913411855697632
L0N1 -> L1N1 = 1.1912697553634644
1B -> L2N0: 7.626241297587034e-36
L1N0 -> L2N0 = -1.548777461051941
L1N1 -> L2N0 = 0.8394404649734497
```

Code

```
input0 = 0
input1 = 1
```

```
hidden0Sum = (input0*1.3)+(input1*1.3)+(-1.3)
hidden1Sum = (input0*1.2)+(input1*1.2)+(0)

print(hidden0Sum) # 0
print(hidden1Sum) # 1.2

hidden0 = max(0,hidden0Sum)
hidden1 = max(0,hidden1Sum)

print(hidden0) # 0
print(hidden1) # 1.2

outputSum = (hidden0*-1.6)+(hidden1*0.8)+(0)
print(outputSum) # 0.96

output = max(0,outputSum)

print(output) # 0.96
```

Output

```
0.0
1.2
0
1.2
0.96
0.96
```

Chapter 4

Training for Tabular Data

4.1 Part 4.1: Encoding a Feature Vector for Keras Deep Learning

Neural networks can accept many types of data. We will begin with tabular data, where there are well defined rows and columns. This is the sort of data you would typically see in Microsoft Excel. An example of tabular data is shown below.

Neural networks require numeric input. This numeric form is called a feature vector. Each row of training data typically becomes one vector. The individual input neurons each receive one feature (or column) from this vector. In this section, we will see how to encode the following tabular data into a feature vector.

Code

```
import pandas as pd

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 5)

display(df)
```

Output

	id	job	area	income	...	pop_dense	retail_dense	crime	product
0	1	vv	c	50876.0	...	0.885827	0.492126	0.071100	b
1	2	kd	c	60369.0	...	0.874016	0.342520	0.400809	c
...	...	...	...	...	...	...	...	...	...
1998	1999	qp	c	67949.0	...	0.909449	0.598425	0.117803	c
1999	2000	pe	c	61467.0	...	0.925197	0.539370	0.451973	c

The following observations can be made from the above data:

- The target column is the column that you seek to predict. There are several candidates here. However, we will initially use product. This field specifies what product someone bought.
- There is an ID column. This column should not be fed into the neural network as it contains no information useful for prediction.
- Many of these fields are numeric and might not require any further processing.
- The income column does have some missing values.
- There are categorical values: job, area, and product.

To begin with, we will convert the job code into dummy variables.

Code

```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

dummies = pd.get_dummies(df['job'],prefix="job")
print(dummies.shape)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)

display(dummies)
```

Output

	job_11	job_al	job_am	job_ax	...	job_rn	job_sa	job_vv	job_zz
0	0	0	0	0	...	0	0	1	0
1	0	0	0	0	...	0	0	0	0
2	0	0	0	0	...	0	0	0	0
3	1	0	0	0	...	0	0	0	0
4	0	0	0	0	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
1995	0	0	0	0	...	0	0	1	0
1996	0	0	0	0	...	0	0	0	0
1997	0	0	0	0	...	0	0	0	0
1998	0	0	0	0	...	0	0	0	0
1999	0	0	0	0	...	0	0	0	0

```
(2000, 33)
```

Because there are 33 different job codes, there are 33 dummy variables. We also specified a prefix, because the job codes (such as "ax") are not that meaningful by themselves. Something such as "job_ax" also tells us the origin of this field.

Next, we must merge these dummies back into the main data frame. We also drop the original "job" field, as it is now represented by the dummies.

Code

```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df = pd.concat([df,dummies],axis=1)
df.drop('job', axis=1, inplace=True)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)

display(df)
```

Output

	id	area	income	aspect	...	job_rn	job_sa	job_vv	job_zz
0	1	c	50876.0	13.100000	...	0	0	1	0
1	2	c	60369.0	18.625000	...	0	0	0	0
2	3	c	55126.0	34.766667	...	0	0	0	0
3	4	c	51690.0	15.808333	...	0	0	0	0
4	5	d	28347.0	40.941667	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
1995	1996	c	51017.0	38.233333	...	0	0	1	0
1996	1997	d	26576.0	33.358333	...	0	0	0	0
1997	1998	d	28595.0	39.425000	...	0	0	0	0
1998	1999	c	67949.0	5.733333	...	0	0	0	0
1999	2000	c	61467.0	16.891667	...	0	0	0	0

We also introduce dummy variables for the area column.

Code

```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)
display(df)
```

Output

	id	income	aspect	subscriptions	...	area_a	area_b	area_c	area_d
0	1	50876.0	13.100000	1	...	0	0	1	0
1	2	60369.0	18.625000	2	...	0	0	1	0
2	3	55126.0	34.766667	1	...	0	0	1	0
3	4	51690.0	15.808333	1	...	0	0	1	0
4	5	28347.0	40.941667	3	...	0	0	0	1
...	...	...	...	...	...	...	...	...	...
1995	1996	51017.0	38.233333	1	...	0	0	1	0
1996	1997	26576.0	33.358333	2	...	0	0	0	1
1997	1998	28595.0	39.425000	3	...	0	0	0	1
1998	1999	67949.0	5.733333	0	...	0	0	1	0
1999	2000	61467.0	16.891667	0	...	0	0	1	0

The last remaining transformation is to fill in missing income values.

Code

```
med = df['income'].median()
df['income'] = df['income'].fillna(med)
```

There are more advanced ways of filling in missing values, but they require more analysis. The idea would be to see if another field might give a hint as to what the income were. For example, it might be beneficial to calculate a median income for each of the areas or job categories. This is something to keep in mind for the class Kaggle competition.

At this point, the Pandas dataframe is ready to be converted to Numpy for neural network training. We need to know a list of the columns that will make up x (the predictors or inputs) and y (the target).

The complete list of columns is:

Code

```
print(list(df.columns))
```

Output

```
['id', 'income', 'aspect', 'subscriptions', 'dist_healthy',
'save_rate', 'dist_unhealthy', 'age', 'pop_dense', 'retail_dense',
'crime', 'product', 'job_11', 'job_al', 'job_am', 'job_ax', 'job_bf',
'job_by', 'job_cv', 'job_de', 'job_dz', 'job_e2', 'job_f8', 'job_gj',
'job_gv', 'job_kd', 'job_ke', 'job_kl', 'job_kp', 'job_ks', 'job_kw',
'job_mm', 'job_nb', 'job_nn', 'job_ob', 'job_pe', 'job_po', 'job_pq',
'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_sa', 'job_vv', 'job_zz',
'area_a', 'area_b', 'area_c', 'area_d']
```

This includes both the target and predictors. We need a list with the target removed. We also remove **id** because it is not useful for prediction.

Code

```
x_columns = df.columns.drop('product').drop('id')
print(list(x_columns))
```

Output

```
['income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate',
'dist_unhealthy', 'age', 'pop_dense', 'retail_dense', 'crime',
'job_11', 'job_al', 'job_am', 'job_ax', 'job_bf', 'job_by', 'job_cv',
'job_de', 'job_dz', 'job_e2', 'job_f8', 'job_gj', 'job_gv', 'job_kd',
'job_ke', 'job_kl', 'job_kp', 'job_ks', 'job_kw', 'job_mm', 'job_nb',
'job_nn', 'job_ob', 'job_pe', 'job_po', 'job_pq', 'job_pz', 'job_qp',
'job_qw', 'job_rn', 'job_sa', 'job_vv', 'job_zz', 'area_a', 'area_b',
'area_c', 'area_d']
```

4.1.1 Generate X and Y for a Classification Neural Network

We can now generate x and y. Note, this is how we generate y for a classification problem. Regression would not use dummies and would simply encode the numeric value of the target.

Code

```
# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

We can display the x and y matrices.

Code

```
print(x)
print(y)
```

Output

```
[[5.08760000e+04 1.31000000e+01 1.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [6.03690000e+04 1.86250000e+01 2.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [5.51260000e+04 3.47666667e+01 1.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 ...
 [2.85950000e+04 3.94250000e+01 3.00000000e+00 ... 0.00000000e+00
```

```
  0.00000000e+00  1.00000000e+00]
 [6.79490000e+04  5.73333333e+00  0.00000000e+00  ...  0.00000000e+00
  1.00000000e+00  0.00000000e+00]
 [6.14670000e+04  1.68916667e+01  0.00000000e+00  ...  0.00000000e+00
  1.00000000e+00  0.00000000e+00]]
[[0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 1 0]
 [0 0 1 ... 0 0 0]
 [0 0 1 ... 0 0 0]]
```

The x and y values are now ready for a neural network. Make sure that you construct the neural network for a classification problem. Specifically,

- Classification neural networks have an output neuron count equal to the number of classes.
- Classification neural networks should use **categorical_crossentropy** and a **softmax** activation function on the output layer.

4.1.2 Generate X and Y for a Regression Neural Network

For a regression neural network, the x values are generated the same. However, y does not use dummies. Make sure to replace **income** with your actual target.

Code

```
y = df['income'].values
```

4.1.3 Module 4 Assignment

You can find the first assignment here: assignment 4

4.2 Part 4.2: Multiclass Classification with ROC and AUC

- **Binary Classification** - Classification between two possibilities (positive and negative). Common in medical testing, does the person have the disease (positive) or not (negative).
- **Classification** - Classification between more than 2. The iris dataset (3-way classification).
- **Regression** - Numeric prediction. How many MPG does a car get? (covered in next video)

In this class session we will look at some visualizations for all three.

It is important to evaluate the level of error in the results produced by a neural network. In this part we will look at how to evaluate error for both classification and regression neural networks.

4.2.1 Binary Classification and ROC Charts

Binary classification occurs when a neural network must choose between two options, which might be true/-false, yes/no, correct/incorrect, or buy/sell. To see how to use binary classification, we will consider a

classification system for a credit card company. This classification system must decide how to respond to a new potential customer. This system will either "issue a credit card" or "decline a credit card."

When you have only two classes that you can consider, the objective function's score is the number of false positive predictions versus the number of false negatives. False negatives and false positives are both types of errors, and it is important to understand the difference. For the previous example, issuing a credit card would be the positive. A false positive occurs when a credit card is issued to someone who will become a bad credit risk. A false negative happens when a credit card is declined to someone who would have been a good risk.

Because only two options exist, we can choose the mistake that is the more serious type of error, a false positive or a false negative. For most banks issuing credit cards, a false positive is worse than a false negative. Declining a potentially good credit card holder is better than accepting a credit card holder who would cause the bank to undertake expensive collection activities.

Consider the following program that uses the wcbreast_wdbc dataset to classify if a breast tumor is cancerous (malignant) or not (benign).

Code

```
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/wcbreast_wdbc.csv",
    na_values=['NA','?'])

pd.set_option('display.max_columns', 5)
pd.set_option('display.max_rows', 5)

display(df)
```

Output

	id	diagnosis	...	worst_symmetry	worst_fractal_dimension
0	842302	M	...	0.4601	0.11890
1	842517	M	...	0.2750	0.08902
...	...	...	...	...	...
567	927241	M	...	0.4087	0.12400
568	92751	B	...	0.2871	0.07039

ROC curves can be a bit confusing. However, they are very common. It is important to know how to read them. Even their name is confusing. Do not worry about their name, it comes from electrical engineering (EE).

Binary classification is common in medical testing. Often you want to diagnose if someone has a disease. This can lead to two types of errors, know as false positives and false negatives:

- **False Positive** - Your test (neural network) indicated that the patient had the disease; however, the patient did not have the disease.
- **False Negative** - Your test (neural network) indicated that the patient did not have the disease; however, the patient did have the disease.
- **True Positive** - Your test (neural network) correctly identified that the patient had the disease.
- **True Negative** - Your test (neural network) correctly identified that the patient did not have the disease.

Types of errors can be seen in Figure 4.1.

True vs False Positives	Type-1 Error	Sensitivity of Test
True vs False Negatives	Type-2 Error	Specificity of Test

Figure 4.1: Type of Error

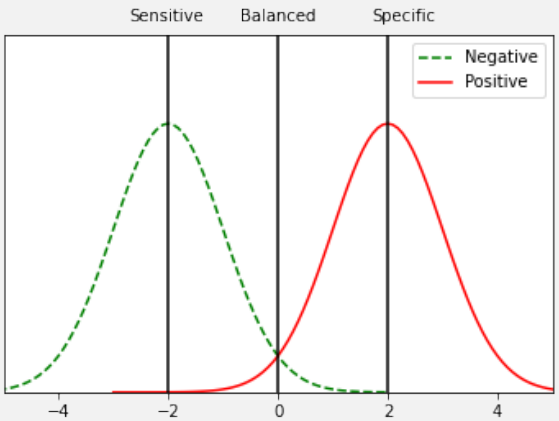
Neural networks classify in terms of probability of it being positive. However, at what probability do you give a positive result? Is the cutoff 50%? 90%? Where you set this cutoff is called the threshold. Anything above the cutoff is positive, anything below is negative. Setting this cutoff allows the model to be more sensitive or specific:

More info on Sensitivity vs Specificity: Khan Academy

Code

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
import math

mu1 = -2
mu2 = 2
variance = 1
sigma = math.sqrt(variance)
x1 = np.linspace(mu1 - 5*sigma, mu1 + 4*sigma, 100)
x2 = np.linspace(mu2 - 5*sigma, mu2 + 4*sigma, 100)
plt.plot(x1, stats.norm.pdf(x1, mu1, sigma)/1,color="green",
         linestyle='dashed')
plt.plot(x2, stats.norm.pdf(x2, mu2, sigma)/1,color="red")
plt.axvline(x=-2,color="black")
plt.axvline(x=0,color="black")
plt.axvline(x=+2,color="black")
plt.text(-2.7,0.55,"Sensitive")
plt.text(-0.7,0.55,"Balanced")
plt.text(1.7,0.55,"Specific")
plt.ylim([0,0.53])
plt.xlim([-5,5])
plt.legend(['Negative','Positive'])
plt.yticks([])
plt.show()
```

Output



Code

```
from scipy.stats import zscore

# Prepare data - apply z-score to ALL x columns
# Only do this if you have no categoricals (and are sure you
# want to use z-score across the board)
x_columns = df.columns.drop('diagnosis').drop('id')
for col in x_columns:
    df[col] = zscore(df[col])

# Convert to numpy - Regression
x = df[x_columns].values
y = df['diagnosis'].map({'M':1,"B":0}).values # Binary classification,
                                              # M is 1 and B is 0
```

Code

```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

# Plot a confusion matrix.
# cm is the confusion matrix, names are the names of the classes.
def plot_confusion_matrix(cm, names, title='Confusion␣matrix',
                          cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(names))
    plt.xticks(tick_marks, names, rotation=45)
    plt.yticks(tick_marks, names)
    plt.tight_layout()
```

```
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


# Plot an ROC. pred - the predictions, y - the expected output.
def plot_roc(pred,y):
    fpr, tpr, _ = roc_curve(y, pred)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC)')
    plt.legend(loc="lower right")
    plt.show()
```

4.2.2 ROC Chart Example

The following code demonstrates how to implement a ROC chart in Python.

Code

```
# Classification neural network
import numpy as np
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu',
                kernel_initializer='random_normal'))
model.add(Dense(50,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(25,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(1,activation='sigmoid',kernel_initializer='random_normal'))
model.compile(loss='binary_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics =['accuracy'])
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
```

```
    patience=5, verbose=1, mode='auto', restore_best_weights=True)

model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

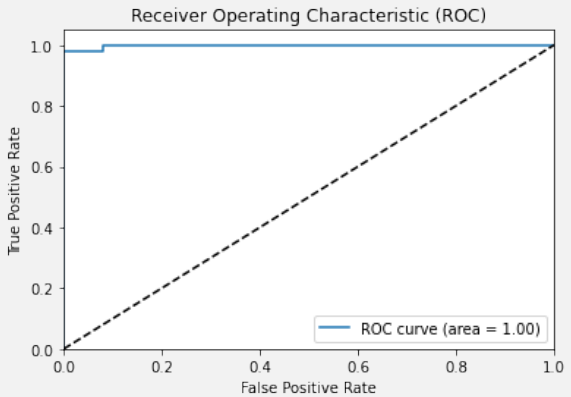
Output

```
Train on 426 samples, validate on 143 samples
Epoch 1/1000
426/426 - 1s - loss: 0.6823 - accuracy: 0.8873 - val_loss: 0.6580 -
val_accuracy: 0.9650
Epoch 2/1000
426/426 - 0s - loss: 0.6171 - accuracy: 0.9343 - val_loss: 0.5181 -
val_accuracy: 0.9650
Epoch 3/1000
426/426 - 0s - loss: 0.4115 - accuracy: 0.9413 - val_loss: 0.2385 -
val_accuracy: 0.9580
Epoch 4/1000
426/426 - 0s - loss: 0.1872 - accuracy: 0.9507 - val_loss: 0.0922 -
val_accuracy: 0.9720
Epoch 5/1000
426/426 - 0s - loss: 0.1031 - accuracy: 0.9718 - val_loss: 0.0613 -

...

Restoring model weights from the end of the best epoch.
426/426 - 0s - loss: 0.0449 - accuracy: 0.9859 - val_loss: 0.0476 -
val_accuracy: 0.9860
Epoch 00012: early stopping
<tensorflow.python.keras.callbacks.History at 0x1f618f96448>
```

Code

```
pred = model.predict(x_test)
plot_roc(pred,y_test)
```

Output



4.2.3 Multiclass Classification Error Metrics

If you want to predict more than one outcome, you will need more than one output neuron. Because a single neuron can predict two outcomes, a neural network with two output neurons is somewhat rare. If there are three or more outcomes, there will be three or more output neurons. The following sections will examine several metrics for evaluating classification error. The following classification neural network will be used to evaluate.

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
```

```
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Code

```
# Classification neural network
import numpy as np
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu',
                kernel_initializer='random_normal'))
model.add(Dense(50,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(25,activation='relu',kernel_initializer='random_normal'))
model.add(Dense(y.shape[1],activation='softmax',
                kernel_initializer='random_normal'))
model.compile(loss='categorical_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics =['accuracy'])
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

Output

```
Train on 1500 samples, validate on 500 samples
Epoch 1/1000
1500/1500 - 1s - loss: 1.5542 - accuracy: 0.4193 - val_loss: 1.1433 -
val_accuracy: 0.4980
Epoch 2/1000
1500/1500 - 0s - loss: 1.1488 - accuracy: 0.4760 - val_loss: 1.0920 -
val_accuracy: 0.4980
```

```
Epoch 3/1000
1500/1500 - 0s - loss: 1.0847 - accuracy: 0.5027 - val_loss: 1.0355 -
val_accuracy: 0.5260
Epoch 4/1000
1500/1500 - 0s - loss: 0.9356 - accuracy: 0.6240 - val_loss: 0.9037 -
val_accuracy: 0.6320
Epoch 5/1000
1500/1500 - 0s - loss: 0.8340 - accuracy: 0.6693 - val_loss: 0.8218 -

...

Restoring model weights from the end of the best epoch.
1500/1500 - 0s - loss: 0.6407 - accuracy: 0.7260 - val_loss: 0.7542 -
val_accuracy: 0.6880
Epoch 00023: early stopping
<tensorflow.python.keras.callbacks.History at 0x1fbd556fc88>
```

4.2.4 Calculate Classification Accuracy

Accuracy is the number of rows where the neural network correctly predicted the target class. Accuracy is only used for classification, not regression.

$$accuracy = \frac{c}{N}$$

Where c is the number correct and N is the size of the evaluated set (training or validation). Higher accuracy numbers are desired.

As we just saw, by default, Keras will return the percent probability for each class. We can change these prediction probabilities into the actual iris predicted with **argmax**.

Code

```
pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)
# raw probabilities to chosen class (highest probability)
```

Now that we have the actual iris flower predicted, we can calculate the percent accuracy (how many were correctly classified).

Code

```
from sklearn import metrics

y_compare = np.argmax(y_test,axis=1)
score = metrics.accuracy_score(y_compare, pred)
print("Accuracy score: {}".format(score))
```

Output

```
Accuracy score: 0.7
```

4.2.5 Calculate Classification Log Loss

Accuracy is like a final exam with no partial credit. However, neural networks can predict a probability of each of the target classes. Neural networks will give high probabilities to predictions that are more likely. Log loss is an error metric that penalizes confidence in wrong answers. Lower log loss values are desired.

The following code shows the output of predict_proba:

Code

```
from IPython.display import display

# Don't display numpy in scientific notation
np.set_printoptions(precision=4)
np.set_printoptions(suppress=True)

# Generate predictions
pred = model.predict(x_test)

print("Numpy array of predictions")
display(pred[0:5])

print("As percent probability")
print(pred[0]*100)

score = metrics.log_loss(y_test, pred)
print("Log loss score: {}".format(score))

# raw probabilities to chosen class (highest probability)
pred = np.argmax(pred,axis=1)
```

Output

```
Numpy array of predictions
array([[0.    , 0.1082, 0.7827, 0.1084, 0.0008, 0.    , 0.    ],
       [0.    , 0.7501, 0.2489, 0.    , 0.0009, 0.    , 0.    ],
       [0.    , 0.7138, 0.284 , 0.0001, 0.0021, 0.0001, 0.    ],
       [0.    , 0.3254, 0.6667, 0.0063, 0.0015, 0.    , 0.    ],
       [0.    , 0.0529, 0.6657, 0.2809, 0.0004, 0.    , 0.    ]],
      dtype=float32)As percent probability
[ 0.     10.819  78.2673 10.8365  0.0765  0.0006  0.    ]
Log loss score: 0.737357779353857
```

Log loss is calculated as follows:

$$\text{log loss} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right)$$

You should use this equation only as an objective function for classifications that have two outcomes. The variable y-hat is the neural network's prediction, and the variable y is the known correct answer. In this case, y will always be 0 or 1. The training data have no probabilities. The neural network classifies it either into one class (1) or the other (0).

The variable N represents the number of elements in the training set the number of questions in the test. We divide by N because this process is customary for an average. We also begin the equation with a negative because the log function is always negative over the domain 0 to 1. This negation allows a positive score for the training to minimize.

You will notice two terms are separated by the addition (+). Each contains a log function. Because y will be either 0 or 1, then one of these two terms will cancel out to 0. If y is 0, then the first term will reduce to 0. If y is 1, then the second term will be 0.

If your prediction for the first class of a two-class prediction is y-hat, then your prediction for the second class is 1 minus y-hat. Essentially, if your prediction for class A is 70% (0.7), then your prediction for class B is 30% (0.3). Your score will increase by the log of your prediction for the correct class. If the neural network had predicted 1.0 for class A, and the correct answer was A, your score would increase by log (1), which is 0. For log loss, we seek a low score, so a correct answer results in 0. Some of these log values for a neural network's probability estimate for the correct class:

- -log(1.0) = 0
- -log(0.95) = 0.02
- -log(0.9) = 0.05
- -log(0.8) = 0.1
- -log(0.5) = 0.3
- -log(0.1) = 1
- -log(0.01) = 2
- -log(1.0e-12) = 12
- -log(0.0) = negative infinity

As you can see, giving a low confidence to the correct answer affects the score the most. Because log (0) is negative infinity, we typically impose a minimum value. Of course, the above log values are for a single training set element. We will average the log values for the entire training set.

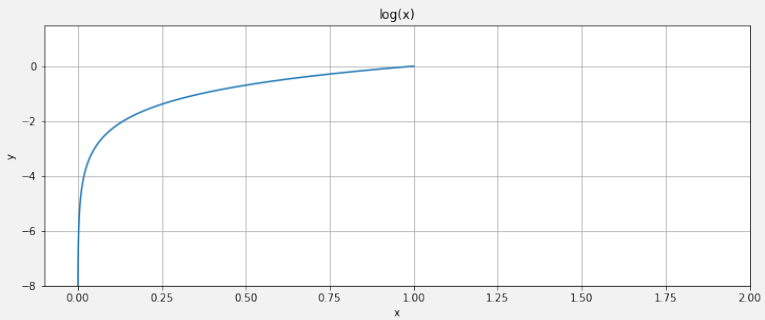
The log function is useful to penalizing wrong answers. The following code demonstrates the utility of the log function:

Code

```
%matplotlib inline
from matplotlib.pyplot import figure, show
from numpy import arange, sin, pi

#t = arange(1e-5, 5.0, 0.00001)
#t = arange(1.0, 5.0, 0.00001) # computer scientists
t = arange(0.0, 1.0, 0.00001)  # data      scientists

fig = figure(1,figsize=(12, 10))

ax1 = fig.add_subplot(211)
ax1.plot(t, np.log(t))
ax1.grid(True)
ax1.set_ylim((-8, 1.5))
ax1.set_xlim((-0.1, 2))
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('log(x)')
```

```
show()
```

Output



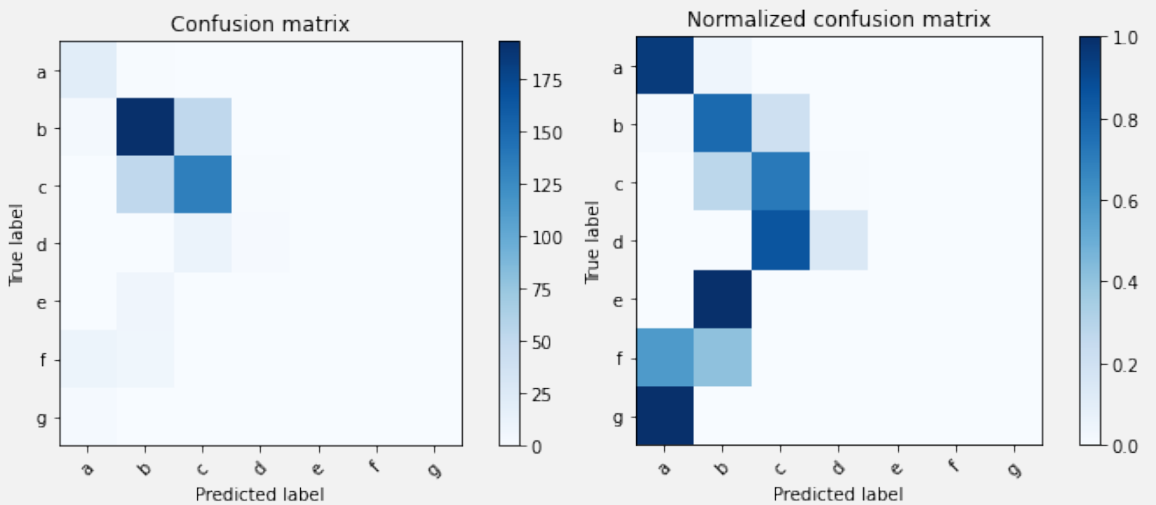
Code

```
import numpy as np
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

# Compute confusion matrix
cm = confusion_matrix(y_compare, pred)
np.set_printoptions(precision=2)
print('Confusion matrix, without normalization')
print(cm)
plt.figure()
plot_confusion_matrix(cm, products)

# Normalize the confusion matrix by row (i.e by the number of samples
# in each class)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
print('Normalized confusion matrix')
print(cm_normalized)
plt.figure()
plot_confusion_matrix(cm_normalized, products,
        title='Normalized confusion matrix')

plt.show()
```

Output

```
Confusion matrix, without normalization
[[ 21   1   0   0   0   0   0]
 [  4 193  52   0   0   0   0]
 [  0  52 134   1   0   0   0]
```

```
 [  0   0  12   2   0   0   0]
 [  0   8   0   0   0   0   0]
 [ 10   7   0   0   0   0   0]
 [  3   0   0   0   0   0   0]]
Normalized confusion matrix
[[0.95 0.05 0.   0.   0.   0.   0.  ]
 [0.02 0.78 0.21 0.   0.   0.   0.  ]
 [0.   0.28 0.72 0.01 0.   0.   0.  ]
 [0.   0.   0.86 0.14 0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.  ]
 [0.59 0.41 0.   0.   0.   0.   0.  ]
 [1.   0.   0.   0.   0.   0.   0.  ]]
```



4.3 Part 4.3: Keras Regression for Deep Neural Networks with RMSE

Regression results are evaluated differently than classification. Consider the following code that trains a neural network for regression on the data set **jh-simple-dataset.csv**.

Code

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])
```

```
# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)
```

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto',
                        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
```

```
            callbacks=[monitor],verbose=2,epochs=1000)
```

Output

```
Train on 1500 samples, validate on 500 samples
Epoch 1/1000
1500/1500 - 1s - loss: 1905.4454 - val_loss: 1628.1341
Epoch 2/1000
1500/1500 - 0s - loss: 1331.4213 - val_loss: 889.0575
Epoch 3/1000
1500/1500 - 0s - loss: 554.8426 - val_loss: 303.7261
Epoch 4/1000
1500/1500 - 0s - loss: 276.2087 - val_loss: 241.2495
Epoch 5/1000
1500/1500 - 0s - loss: 232.2832 - val_loss: 208.2143
Epoch 6/1000
1500/1500 - 0s - loss: 198.5331 - val_loss: 179.5262
Epoch 7/1000
1500/1500 - 0s - loss: 169.0791 - val_loss: 154.5270

...

Epoch 124/1000
Restoring model weights from the end of the best epoch.
1500/1500 - 0s - loss: 0.4353 - val_loss: 0.5538
Epoch 00124: early stopping
<tensorflow.python.keras.callbacks.History at 0x1a40e6b0d0>
```

4.3.1 Mean Square Error

The mean square error is the sum of the squared differences between the prediction ($\hat{y}$) and the expected (y). MSE values are not of a particular unit. If an MSE value has decreased for a model, that is good. However, beyond this, there is not much more you can determine. Low MSE values are desired.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Code

```
from sklearn import metrics

# Predict
pred = model.predict(x_test)

# Measure MSE error.
score = metrics.mean_squared_error(pred,y_test)
print("Final score (MSE): {}".format(score))
```

Output

```
Final score (MSE): 0.5463447829677607
```

4.3.2 Root Mean Square Error

The root mean square (RMSE) is essentially the square root of the MSE. Because of this, the RMSE error is in the same units as the training data outcome. Low RMSE values are desired.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

Code

```
import numpy as np

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {}".format(score))
```

Output

```
Final score (RMSE): 0.7391513938076291
```

4.3.3 Lift Chart

To generate a lift chart, perform the following activities:

- Sort the data by expected output. Plot the blue line above.
- For every point on the x-axis plot the predicted value for that same data point. This is the green line above.
- The x-axis is just 0 to 100% of the dataset. The expected always starts low and ends high.
- The y-axis is ranged according to the values predicted.

Reading a lift chart:

- The expected and predict lines should be close. Notice where one is above the ot other.
- The below chart is the most accurate on lower age.

Code

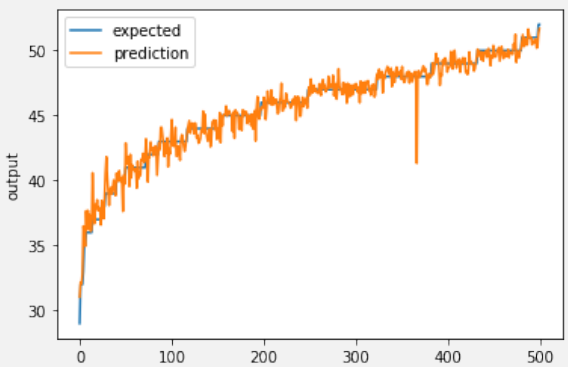
```
# Regression chart.
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
```

```
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()
```

Code

```
# Plot the chart
chart_regression(pred.flatten(),y_test)
```

Output



4.4 Part 4.4: Training Neural Networks

4.4.1 Classic Backpropagation

Backpropagation is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams (1986) introduced backpropagation, and it remains popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Classic backpropagation has been extended and modified to give rise to many different training algorithms. In this chapter, we will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and then end the chapter with stochastic gradient descent (SGD).

Backpropagation is the primary means by which a neural network's weights are determined during training. Backpropagation works by calculating a weight change amount (v_t) for every weight(θ, theata) in the neural network. This value is subtracted from every weight by the following equation:

$$\theta_t = \theta_{t-1} - v_t$$

This process is repeated for every iteration(t). How the weight change is calculated depends on the training algorithm. Classic backpropagation simply calculates a gradient (∇, nabla) for every weight in the neural network with respect to the error function (J) of the neural network. The gradient is scaled by a learning rate (η, eta).

$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1})$

The learning rate is an important concept for backpropagation training. Setting the learning rate can be complex:

- Too low of a learning rate will usually converge to a good solution; however, the process will be very slow.
- Too high of a learning rate will either fail outright, or converge to a higher error than a better learning rate.

Common values for learning rate are: 0.1, 0.01, 0.001, etc.

Backpropagation is a type of gradient descent, and many texts will use these two terms interchangeably. Gradient descent refers to the calculation of a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will give you an indication about how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the expected output. In fact, we can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

With respect to the error function, the gradient is essentially the partial derivative of each weight in the neural network. Each weight has a gradient that is the slope of the error function. A weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

- Zero gradient - The weight is not contributing to the error of the neural network.
- Negative gradient - The weight should be increased to achieve a lower error.
- Positive gradient - The weight should be decreased to achieve a lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process. First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had an infinite amount of computation resources, we would simply try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some sort of shortcut to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 4.2 is a graph that demonstrates the error for a single weight:

Looking at this chart, you can easily see that the optimal weight is the location where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. However, we can determine the slope of the error curve at a particular weight. In the above chart, we see the slope of the error curve at 1.5. The straight line that barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error.

The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given

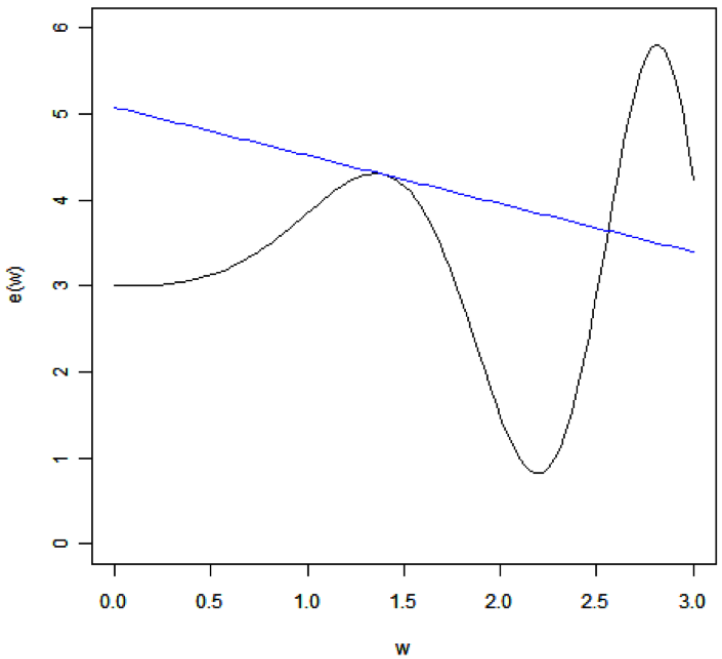


Figure 4.2: Derivative

weight.

Derivatives are one of the most fundamental concepts in calculus. For the purposes of this book, you just need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to adjust the weight for a lower error. Using our working definition of the gradient, we will now show how to calculate it.

The following link, from the book, shows how a simple neural network is trained with backpropagation.

4.4.2 Momentum Backpropagation

Momentum adds another term to the calculation of v_t:

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1}) + \lambda v_{t-1}$$

Like the learning rate, momentum adds another training parameter that scales the effect of momentum. Momentum backpropagation has two training parameters: learning rate (η, eta) and momentum (λ, lambda). Momentum simply adds the scaled value of the previous weight change amount (v_{t-1}) to the current weight change amount(v_t).

This has the effect of adding additional force behind a direction a weight was moving. Figure 4.3 shows how this might allow the weight to escape a local minima.

A very common value for momentum is 0.9.

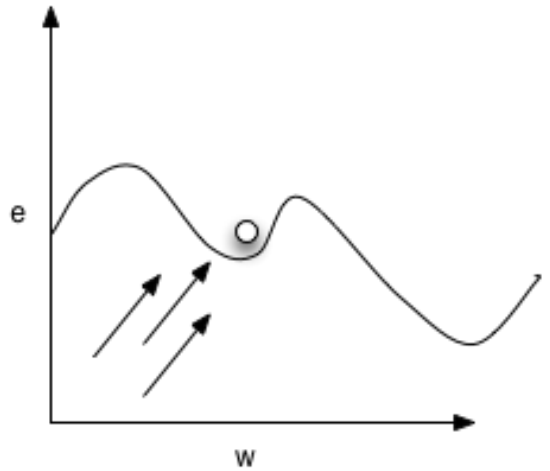


Figure 4.3: Momentum

4.4.3 Batch and Online Backpropagation

How often should the weights of a neural network be updated? Gradients can be calculated for a training set element. These gradients can also be summed together into batches and the weights updated once per batch.

- **Online Training** - Update the weights based on gradients calculated from a single training set element.
- **Batch Training** - Update the weights based on the sum of the gradients over all training set elements.
- **Batch Size** - Update the weights based on the sum of some batch size of training set elements.
- **Mini-Batch Training** - The same as batch size, but with a very small batch size. Mini-batches are very popular and they are often in the 32-64 element range.

Because the batch size is smaller than the complete training set size, it may take several batches to make it completely through the training set.

- **Step/Iteration** - The number of batches that were processed.
- **Epoch** - The number of times the complete training set was processed.

4.4.4 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is currently one of the most popular neural network training algorithms. It works very similarly to Batch/Mini-Batch training, except that the batches are made up of a random set of training elements.

This leads to a very irregular convergence in error during training as shown in Figure 4.4.

Image from Wikipedia

Because the neural network is trained on a random sample of the complete training set each time, the error does not make a smooth transition downward. However, the error usually does go down.

Advantages to SGD include:

- Computationally efficient. Even with a very large training set, each training step can be relatively fast.
- Decreases overfitting by focusing on only a portion of the training set each step.

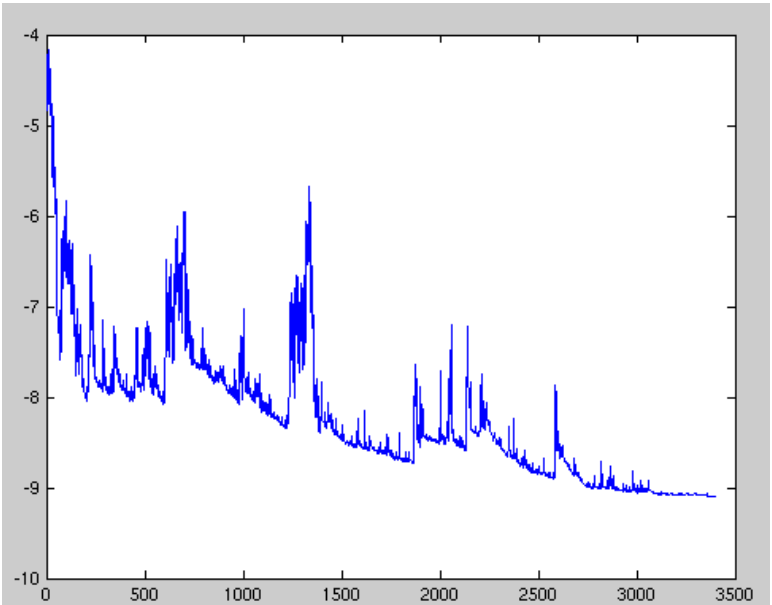


Figure 4.4: SGD Error

4.4.5 Other Techniques

One problem with simple backpropagation training algorithms is that they are highly sensative to learning rate and momentum. This is difficult because:

- Learning rate must be adjusted to a small enough level to train an accurate neural network.
- Momentum must be large enough to overcome local minima, yet small enough to not destabilize the training.
- A single learning rate/momentum is often not good enough for the entire training process. It is often useful to automatically decrease learning rate as the training progresses.
- All weights share a single learning rate/momentum.

Other training techniques:

- **Resilient Propagation** - Use only the magnitude of the gradient and allow each neuron to learn at its own rate. No need for learning rate/momentum; however, only works in full batch mode.
- **Nesterov accelerated gradient** - Helps mitigate the risk of choosing a bad mini-batch.
- **Adagrad** - Allows an automatically decaying per-weight learning rate and momentum concept.
- **Adadelta** - Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- **Non-Gradient Methods** - Non-gradient methods can *sometimes* be useful, though rarely outperform gradient-based backpropagation methods. These include: simulated annealing, genetic algorithms, particle swarm optimization, Nelder Mead, and many more.

4.4.6 ADAM Update

ADAM is the first training algorithm you should try. It is very effective. Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates that it uses.[23]Adam

estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with an exponentially decaying average of past gradients (m):

$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

This average accomplishes a similar goal as classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

The values m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. However, they will have a strong bias towards zero in the initial training cycles. The first moment's bias is corrected as follows.

$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$

Similarly, the second moment is also corrected:

$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$

These bias-corrected first and second moment estimates are applied to the ultimate Adam update rule, as follows:

$\theta_t = \theta_{t-1} - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \eta} \hat{m}_t$

Adam is very tolerant to initial learning rate (\alpha) and other training parameters. Kingma and Ba (2014) propose default values of 0.9 for β_1, 0.999 for β_2, and 10-8 for η.

4.4.7 Methods Compared

The following image shows how each of these algorithms train (image credits: [author](Alec Radford), where I found it):

4.4.8 Specifying the Update Rule in Tensorflow

TensorFlow allows the update rule to be set to one of:

- Adagrad
- **Adam**
- Ftrl
- Momentum
- RMSProp
- **SGD**

Code

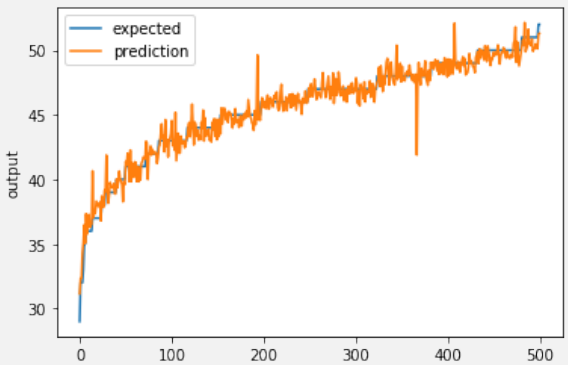
```
%matplotlib inline

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Regression chart.
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)
```

```
# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam') # Modify here
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)

# Plot the chart
pred = model.predict(x_test)
chart_regression(pred.flatten(),y_test)
```

Output

```
Restoring model weights from the end of the best epoch.
Epoch 00105: early stopping
```



4.5 Part 4.5: Error Calculation from Scratch

We will now look at how to calculate RMSE and logloss by hand.

4.5.1 Regression

Code

```
from sklearn import metrics
import numpy as np

predicted = [1.1,1.9,3.4,4.2,4.3]
expected = [1,2,3,4,5]
```

```
score_mse = metrics.mean_squared_error(predicted,expected)
score_rmse = np.sqrt(score_mse)
print("Score (MSE): {}".format(score_mse))
print("Score (RMSE): {}".format(score_rmse))
```

Output

```
Score (MSE): 0.14200000000000007
Score (RMSE): 0.37682887362833556
```

Code

```
score_mse = ((predicted[0]-expected[0])**2 + (predicted[1]-expected[1])**2
+ (predicted[2]-expected[2])**2 + (predicted[3]-expected[3])**2
+ (predicted[4]-expected[4])**2)/len(predicted)
score_rmse = np.sqrt(score_mse)

print("Score (MSE): {}".format(score_mse))
print("Score (RMSE): {}".format(score_rmse))
```

Output

```
Score (MSE): 0.14200000000000007
Score (RMSE): 0.37682887362833556
```

4.5.2 Classification

We will now look at how to calculate a logloss by hand. For this we look at a binary prediction. The expected is always 0 or 1. The predicted is some number between 0-1 that indicates the probability true (1). Therefore, a prediction of 1.0 is completely correct if the expected is 1 and completely wrong if the expected is 0.

Code

```
from sklearn import metrics

expected = [1,1,0,0,0]
predicted = [0.9,0.99,0.1,0.05,0.06]

print(metrics.log_loss(expected,predicted))
```

Output

```
0.06678801305495843
```

Now we attempt to calculate the same logloss manually.

Code

```
import numpy as np

score_logloss = (np.log(1.0-np.abs(expected[0]-predicted[0]))+\
np.log(1.0-np.abs(expected[1]-predicted[1]))+\
np.log(1.0-np.abs(expected[2]-predicted[2]))+\
np.log(1.0-np.abs(expected[3]-predicted[3]))+\
np.log(1.0-np.abs(expected[4]-predicted[4])))\
*(-1/len(predicted))

print(f'Score Logloss {score_logloss}')
```

Output

```
Score Logloss 0.06678801305495843
```

Chapter 5

Regularization and Dropout

5.1 Part 5.1: Introduction to Regularization: Ridge and Lasso

Regularization is a technique that reduces overfitting, which occurs when neural networks attempt to memorize training data, rather than learn from it. Humans are capable of overfitting as well. Before we examine the ways that a machine accidentally overfits, we will first explore how humans can suffer from it.

Human programmers often take certification exams to show their competence in a given programming language. To help prepare for these exams, the test makers often make practice exams available. Consider a programmer who enters a loop of taking the practice exam, studying more, and then taking the practice exam again. At some point, the programmer has memorized much of the practice exam, rather than learning the techniques necessary to figure out the individual questions. The programmer has now overfit to the practice exam. When this programmer takes the real exam, his actual score will likely be lower than what he earned on the practice exam.

A computer can overfit as well. Although a neural network received a high score on its training data, this result does not mean that the same neural network will score high on data that was not inside the training set. Regularization is one of the techniques that can prevent overfitting. A number of different regularization techniques exist. Most work by analyzing and potentially modifying the weights of a neural network as it trains.

5.1.1 L1 and L2 Regularization

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting (Ng, 2004). Both of these algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases the regularization algorithm is attached to the training algorithm by adding an additional objective.

Both of these algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. For gradient-descent-based algorithms, such as backpropagation, you can add this penalty calculation to the calculated gradients. For objective-function-based training, such as simulated annealing, the penalty is negatively combined with the objective score.

We are going to look at linear regression to see how L1 and L2 regularization work. The following code sets up the auto-mpg data for this purpose.

Code

```
from sklearn.linear_model import LassoCV
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
names = ['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']
x = df[names].values
y = df['mpg'].values # regression

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=45)
```

Code

```
# Simple function to evaluate the coefficients of a regression
%matplotlib inline
from IPython.display import display, HTML

def report_coef(names,coef,intercept):
    r = pd.DataFrame( { 'coef': coef, 'positive': coef>=0
}, index = names )
    r = r.sort_values(by=['coef'])
    display(r)
    print(f"Intercept: {intercept}")
    r['coef'].plot(kind='barh', color=r['positive'].map(
        {True: 'b', False: 'r'}))
```

5.1.2 Linear Regression

To understand L1/L2 regularization, it is good to start with linear regression. L1/L2 were first introduced for linear regression. They can also be used for neural networks. To fully understand L1/L2 we will begin with how they are used with linear regression.

The following code uses linear regression to fit the auto-mpg data set. The RMSE reported will not be as good as a neural network.

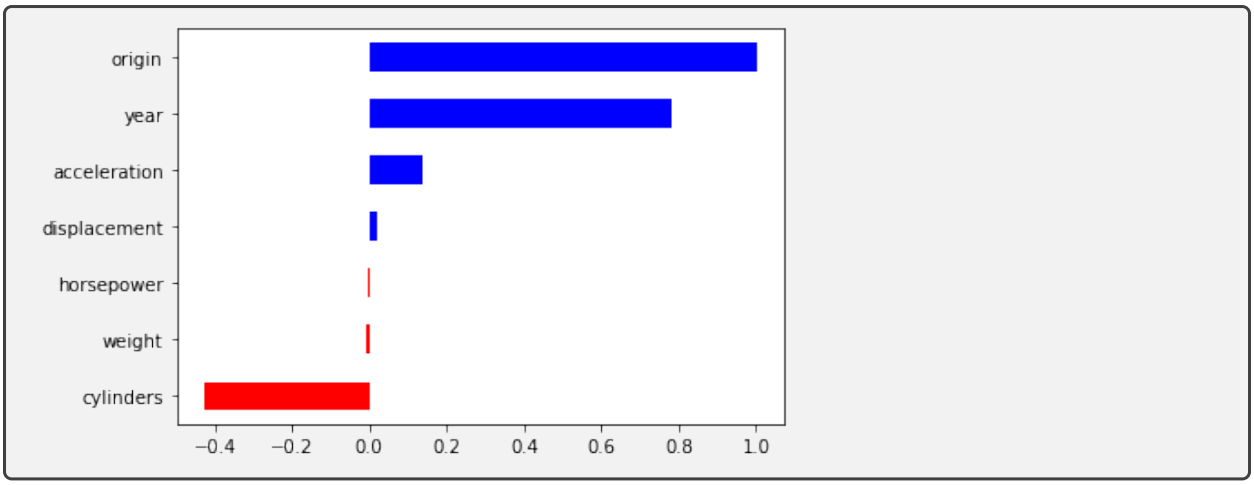
Code

```
import sklearn

# Create linear regression
regressor = sklearn.linear_model.LinearRegression()

# Fit/train linear regression
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")

report_coef(
  names,
  regressor.coef_,
  regressor.intercept_)
```

Output

	coef	positive
cylinders	-0.427721	False
weight	-0.007255	False
horsepower	-0.005491	False
displacement	0.020166	True
acceleration	0.138575	True
year	0.783047	True
origin	1.003762	True

```
Final score (RMSE): 3.0019345985860784
Intercept: -19.101231042200112
```



5.1.3 L1 (Lasso) Regularization

L1 Regularization, also called LASSO (Least Absolute Shrinkage and Selection Operator) is should be used to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When a weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach 0, the neural network training determines that the feature is unnecessary. If your data set has a large number of input features that may not be needed, L1 regularization can help the neural network detect and ignore unnecessary features.

L1 is implemented by adding the following error to the objective to minimize:

$$E_1 = \alpha \sum_w |w|$$

You should use L1 regularization to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When a weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.
Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach 0, the neural network training determines that the feature is unnecessary. If your data set has a large number of input features that may not be needed, L1 regularization can help the neural network detect and ignore unnecessary features.

The following code demonstrates lasso regression. Notice the effect of the coefficients compared to the previous section that used linear regression.

Code

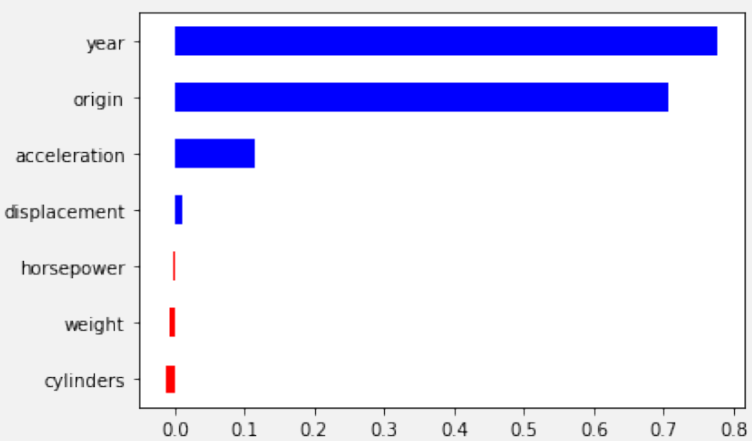
```
import sklearn
from sklearn.linear_model import Lasso

# Create linear regression
regressor = Lasso(random_state=0,alpha=0.1)

# Fit/train LASSO
regressor.fit(x_train,y_train)
```

```
# Predict
pred = regressor.predict(x_test)

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")

report_coef(
  names,
  regressor.coef_,
  regressor.intercept_)
```

Output

	coef	positive
cylinders	-0.012995	False
weight	-0.007328	False
horsepower	-0.002715	False
displacement	0.011601	True
acceleration	0.114391	True
origin	0.708222	True
year	0.777480	True

```
Final score (RMSE): 3.0604021904033303
Intercept: -18.506677982383252
```



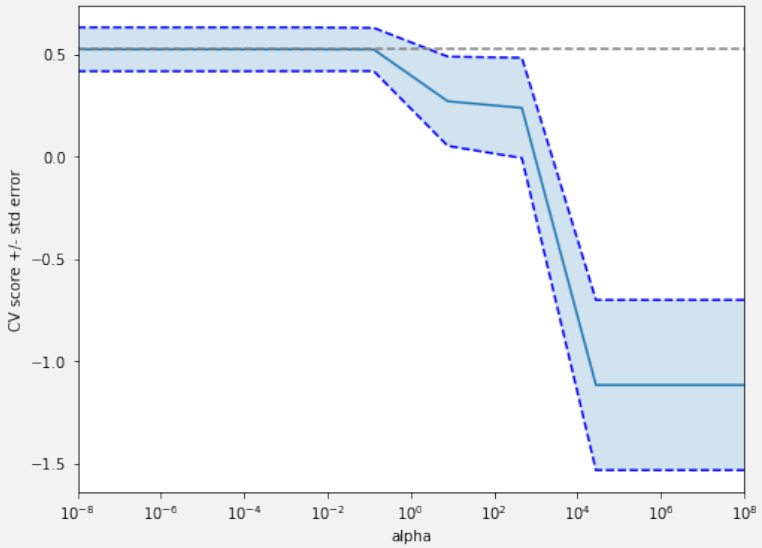
Code

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LassoCV
from sklearn.linear_model import Lasso
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

lasso = Lasso(random_state=42)
alphas = np.logspace(-8, 8, 10)

scores = list()
scores_std = list()

n_folds = 3

for alpha in alphas:
    lasso.alpha = alpha
    this_scores = cross_val_score(lasso, x, y, cv=n_folds, n_jobs=1)
    scores.append(np.mean(this_scores))
    scores_std.append(np.std(this_scores))

scores, scores_std = np.array(scores), np.array(scores_std)

plt.figure().set_size_inches(8, 6)
plt.semilogx(alphas, scores)

# plot error lines showing +/- std. errors of the scores
std_error = scores_std / np.sqrt(n_folds)

plt.semilogx(alphas, scores + std_error, 'b--')
plt.semilogx(alphas, scores - std_error, 'b--')

# alpha=0.2 controls the translucency of the fill color
plt.fill_between(alphas, scores + std_error, scores - std_error, alpha=0.2)

plt.ylabel('CV score +/- std error')
plt.xlabel('alpha')
plt.axhline(np.max(scores), linestyle='--', color='.5')
plt.xlim([alphas[0], alphas[-1]])
```

Output

```
(1e-08, 100000000.0)
```



5.1.4 L2 (Ridge) Regularization

You should use Tikhonov/Ridge/L2 regularization when you are less concerned about creating a space network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting.

$E_2 = \alpha \sum_w w^2$

Like the L1 algorithm, the α value determines how important the L2 objective is compared to the neural network's error. Typical L2 values are below 0.1 (10%). The main calculation performed by L2 is the summing of the squares of all of the weights. The bias values are not summed.

You should use L2 regularization when you are less concerned about creating a space network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting. Generally L2 regularization will produce better overall performance than L1. However, L1 might be useful in situations where there are a large number of inputs and some of the weaker inputs should be pruned.

The following code uses L2 with linear regression (Ridge regression):

Code

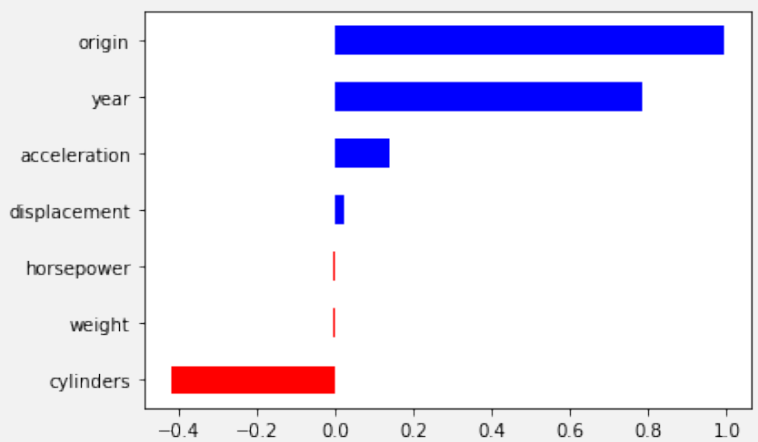
```
import sklearn
from sklearn.linear_model import Ridge

# Create linear regression
regressor = Ridge(alpha=1)

# Fit/train Ridge
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

# Measure RMSE error.  RMSE is common for regression.
```

```
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {score}")

report_coef(
  names,
  regressor.coef_,
  regressor.intercept_)
```

Output

	coef	positive
cylinders	-0.421393	False
weight	-0.007257	False
horsepower	-0.005385	False
displacement	0.020006	True
acceleration	0.138470	True
year	0.782889	True
origin	0.994621	True

```
Final score (RMSE): {score}
Intercept: -19.07980074425469
```



5.1.5 ElasticNet Regularization

The ElasticNet regression combines both L1 and L2. Both penalties are applied. The amount of L1 and L2 are governed by the parameters alpha and beta.

$a * L1 + b * L2$

Code

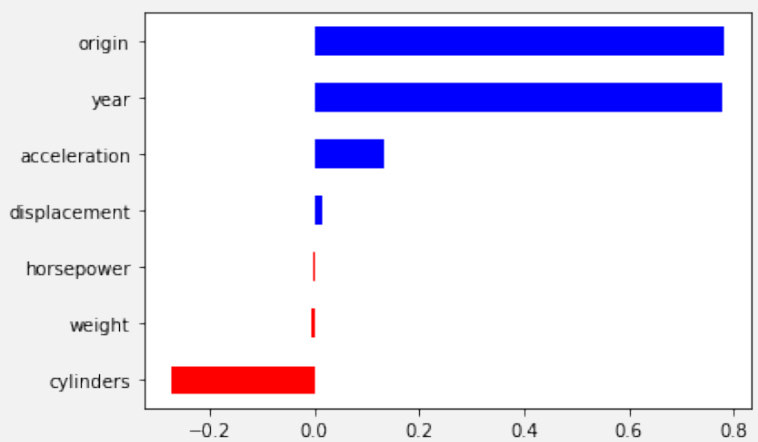
```
import sklearn
from sklearn.linear_model import ElasticNet
```

```
# Create linear regression
regressor = ElasticNet(alpha=0.1, l1_ratio=0.1)

# Fit/train LASSO
regressor.fit(x_train,y_train)
# Predict
pred = regressor.predict(x_test)

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"Final score (RMSE): {score}")

report_coef(
  names,
  regressor.coef_,
  regressor.intercept_)
```

Output

	coef	positive
cylinders	-0.274010	False
weight	-0.007303	False
horsepower	-0.003231	False
displacement	0.016194	True
acceleration	0.132348	True
year	0.777482	True
origin	0.782781	True

```
Final score (RMSE): 3.0450899960775013
Intercept: -18.389355690429767
```



5.2 Part 5.2: Using K-Fold Cross-validation with Keras

Cross-validation can be used for a variety of purposes in predictive modeling. These include:

- Generating out-of-sample predictions from a neural network
- Estimate a good number of epochs to train a neural network for (early stopping)
- Evaluate the effectiveness of certain hyperparameters, such as activation functions, neuron counts, and layer counts

Cross-validation uses a number of folds, and multiple models, to provide each segment of data a chance to serve as both the validation and training set. Cross validation is shown in Figure 5.1.

It is important to note that there will be one model (neural network) for each fold. To generate predictions for new data, which is data not present in the training set, predictions from the fold models can be handled in several ways:

- Choose the model that had the highest validation score as the final model.
- Preset new data to the 5 models (one for each fold) and average the result (this is an ensemble).
- Retrain a new model (using the same settings as the cross-validation) on the entire dataset. Train for as many epochs, and with the same hidden layer structure.

Generally, I prefer the last approach and will retrain a model on the entire data set once I have selected hyper-parameters. Of course, I will always set aside a final holdout set for model validation that I do not use in any aspect of the training process.

5.2.1 Regression vs Classification K-Fold Cross-Validation

Regression and classification are handled somewhat differently with regards to cross-validation. Regression is the simpler case where you can simply break up the data set into K folds with little regard for where each item lands. For regression it is best that the data items fall into the folds as randomly as possible. It is also important to remember that not every fold will necessarily have exactly the same number of data items. It is not always possible for the data set to be evenly divided into K folds. For regression cross-validation we will use the Scikit-Learn class **KFold**.

Cross validation for classification could also use the **KFold** object; however, this technique would not ensure that the class balance remains the same in each fold as it was in the original. It is very important that the balance of classes that a model was trained on remains the same (or similar) to the training set. A drift in this distribution is one of the most important things to monitor after a trained model has been placed into actual use. Because of this, we want to make sure that the cross-validation itself does not introduce an unintended shift. This is referred to as stratified sampling and is accomplished by using the Scikit-Learn object **StratifiedKFold** in place of **KFold** whenever you are using classification. In summary, the following two objects in Scikit-Learn should be used:

- **KFold** When dealing with a regression problem.
- **StratifiedKFold** When dealing with a classification problem.

The following two sections demonstrate cross-validation with classification and regression.

5.2.2 Out-of-Sample Regression Predictions with K-Fold Cross-Validation

The following code trains the simple dataset using a 5-fold cross-validation. The expected performance of a neural network, of the type trained here, would be the score for the generated out-of-sample predictions. We begin by preparing a feature vector using the jh-simple-dataset to predict age. This is a regression problem.

Code

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values
```

Now that the feature vector is created a 5-fold cross-validation can be performed to generate out of sample predictions. We will assume 500 epochs, and not use early stopping. Later we will see how we can estimate a more optimal epoch count.

Code

```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
```

```
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
# Cross-Validate
kf = KFold(5, shuffle=True, random_state=42) # Use for KFold classification

oos_y = []
oos_pred = []

fold = 0
for train, test in kf.split(x):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    model = Sequential()
    model.add(Dense(20, input_dim=x.shape[1], activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),verbose=0,
              epochs=500)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    oos_pred.append(pred)

    # Measure this fold's RMSE
    score = np.sqrt(metrics.mean_squared_error(pred,y_test))
    print(f"Fold score (RMSE): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
score = np.sqrt(metrics.mean_squared_error(oos_pred,oos_y))
print(f"Final, out of sample score (RMSE): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
#oosDF.to_csv(filename_write,index=False)
```

Output

```
Fold #1
Fold score (RMSE): 0.6245484893737087
Fold #2
Fold score (RMSE): 0.5802295511082306
Fold #3
Fold score (RMSE): 0.6300965769274195
Fold #4
Fold score (RMSE): 0.4550931884841248
Fold #5
Fold score (RMSE): 1.0517027192572377
Final, out of sample score (RMSE): 0.6981314007708873
```

As you can see, the above code also reports the average number of epochs needed. A common technique is to then train on the entire dataset for the average number of epochs needed.

5.2.3 Classification with Stratified K-Fold Cross-Validation

The following code trains and fits the jh-simple-dataset dataset with cross-validation to generate out-of-sample . It also writes out the out of sample (predictions on the test set) results.

It is good to perform a stratified k-fold cross validation with classification data. This ensures that the percentages of each class remains the same across all folds. To do this, make use of the **StratifiedKFold** object, instead of the **KFold** object used in regression.

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
```

```
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

We will assume 500 epochs, and not use early stopping. Later we will see how we can estimate a more optimal epoch count.

Code

```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation

# np.argmax(pred,axis=1)
# Cross-validate
# Use for StratifiedKFold classification
kf = StratifiedKFold(5, shuffle=True, random_state=42)

oos_y = []
oos_pred = []
fold = 0

# Must specify y StratifiedKFold for
for train, test in kf.split(x,df['product']):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    model = Sequential()
    model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
    model.add(Dense(25, activation='relu')) # Hidden 2
    model.add(Dense(y.shape[1],activation='softmax')) # Output
    model.compile(loss='categorical_crossentropy', optimizer='adam')
```

```
    model.fit(x_train,y_train,validation_data=(x_test,y_test),verbose=0,\
              epochs=500)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    # raw probabilities to chosen class (highest probability)
    pred = np.argmax(pred,axis=1)
    oos_pred.append(pred)

    # Measure this fold's accuracy
    y_compare = np.argmax(y_test,axis=1) # For accuracy calculation
    score = metrics.accuracy_score(y_compare, pred)
    print(f"Fold score (accuracy): {score}")

# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y,axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
#oosDF.to_csv(filename_write,index=False)
```

Output

```
Fold #1
Fold score (accuracy): 0.6766169154228856
Fold #2
Fold score (accuracy): 0.6691542288557214
Fold #3
Fold score (accuracy): 0.6907730673316709
Fold #4
Fold score (accuracy): 0.6733668341708543
Fold #5
Fold score (accuracy): 0.654911838790932
Final score (accuracy): 0.673
```

5.2.4 Training with both a Cross-Validation and a Holdout Set

If you have a considerable amount of data, it is always valuable to set aside a holdout set before you cross-validate. This hold out set will be the final evaluation before you make use of your model for its real-world use. Figure 5.2 shows this division.

The following program makes use of a holdout set, and then still cross-validates.

Code

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values
```

Now that the data has been preprocessed, we are ready to build the neural network.

Code

```
from sklearn.model_selection import train_test_split
```

```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import KFold

# Keep a 10% holdout
x_main, x_holdout, y_main, y_holdout = train_test_split(
    x, y, test_size=0.10)


# Cross-validate
kf = KFold(5)

oos_y = []
oos_pred = []
fold = 0
for train, test in kf.split(x_main):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x_main[train]
    y_train = y_main[train]
    x_test = x_main[test]
    y_test = y_main[test]

    model = Sequential()
    model.add(Dense(20, input_dim=x.shape[1], activation='relu'))
    model.add(Dense(5, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),
              verbose=0,epochs=500)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    oos_pred.append(pred)

    # Measure accuracy
    score = np.sqrt(metrics.mean_squared_error(pred,y_test))
    print(f"Fold score (RMSE): {score}")


# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
```

```
oos_pred = np.concatenate(oos_pred)
score = np.sqrt(metrics.mean_squared_error(oos_pred,oos_y))
print()
print(f"Cross-validated score (RMSE): {score}")

# Write the cross-validated prediction (from the last neural network)
holdout_pred = model.predict(x_holdout)

score = np.sqrt(metrics.mean_squared_error(holdout_pred,y_holdout))
print(f"Holdout score (RMSE): {score}")
```

Output

```
Fold #1
Fold score (RMSE): 24.299626704604506
Fold #2
Fold score (RMSE): 0.6609159891625663
Fold #3
Fold score (RMSE): 0.4997884237817687
Fold #4
Fold score (RMSE): 1.1084218284103058
Fold #5
Fold score (RMSE): 0.614899992174395
Cross-validated score (RMSE): 10.888206072135832
Holdout score (RMSE): 0.6283593821273058
```

5.3 Part 5.3: L1 and L2 Regularization to Decrease Overfitting

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting[31]. Both of these algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases the regularization algorithm is attached to the training algorithm by adding an additional objective.

Both of these algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. For gradient-descent-based algorithms, such as backpropagation, you can add this penalty calculation to the calculated gradients. For objective-function-based training, such as simulated annealing, the penalty is negatively combined with the objective score.

Both L1 and L2 work differently in the way that they penalize the size of a weight. L2 will force the weights into a pattern similar to a Gaussian distribution; the L1 will force the weights into a pattern similar to a Laplace distribution, as demonstrated in Figure 5.3.

As you can see, L1 algorithm is more tolerant of weights further from 0, whereas the L2 algorithm is less tolerant. We will highlight other important differences between L1 and L2 in the following sections. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values. Keras allows l1/l2 to be directly added to your network.

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Code

```
########################################
# Keras with L1/L2 for Regression
########################################

import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers

# Cross-validate
kf = KFold(5, shuffle=True, random_state=42)

oos_y = []
oos_pred = []
fold = 0

for train, test in kf.split(x):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    #kernel_regularizer=regularizers.l2(0.01),

    model = Sequential()
    model.add(Dense(50, input_dim=x.shape[1],
            activation='relu',
             activity_regularizer=regularizers.l1(1e-4))) # Hidden 1
    model.add(Dense(25, activation='relu',
                    activity_regularizer=regularizers.l1(1e-4))) # Hidden 2
    model.add(Dense(y.shape[1],activation='softmax')) # Output
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),
              verbose=0,epochs=500)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    # raw probabilities to chosen class (highest probability)
    pred = np.argmax(pred,axis=1)
    oos_pred.append(pred)

    # Measure this fold's accuracy
    y_compare = np.argmax(y_test,axis=1) # For accuracy calculation
    score = metrics.accuracy_score(y_compare, pred)
    print(f"Fold score (accuracy): {score}")


# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
```

```
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y,axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
#oosDF.to_csv(filename_write,index=False)
```

Output

```
Fold #1
Fold score (accuracy): 0.64
Fold #2
Fold score (accuracy): 0.6775
Fold #3
Fold score (accuracy): 0.6825
Fold #4
Fold score (accuracy): 0.6675
Fold #5
Fold score (accuracy): 0.645
Final score (accuracy): 0.6625
```

5.4 Part 5.4: Drop Out for Keras to Decrease Overfitting

Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov (2012) introduced the dropout regularization algorithm.[37]Although dropout works in a different way than L1 and L2, it accomplishes the same goal---the prevention of overfitting. However, the algorithm goes about the task by actually removing neurons and connections---at least temporarily. Unlike L1 and L2, no weight penalty is added. Dropout does not directly seek to train small weights.

Dropout works by causing hidden neurons of the neural network to be unavailable during part of the training. Dropping part of the neural network causes the remaining portion to be trained to still achieve a good score even without the dropped neurons. This decreases coadaption between neurons, which results in less overfitting.

Most neural network frameworks implement dropout as a separate layer. Dropout layers function as a regular, densely connected neural network layer. The only difference is that the dropout layers will periodically drop some of their neurons during training. You can use dropout layers on regular feedforward neural networks.

The program implements a dropout layer as a dense layer that can eliminate some of its neurons. Contrary to popular belief about the dropout layer, the program does not permanently remove these discarded neurons. A dropout layer does not lose any of its neurons during the training process, and it will still have exactly the same number of neurons after training. In this way, the program only temporarily masks the neurons rather than dropping them.

Figure 5.4 shows how a dropout layer might be situated with other layers.

The discarded neurons and their connections are shown as dashed lines. The input layer has two input neurons as well as a bias neuron. The second layer is a dense layer with three neurons as well as a bias neuron. The third layer is a dropout layer with six regular neurons even though the program has dropped 50% of them. While the program drops these neurons, it neither calculates nor trains them. However, the final neural network will use all of these neurons for the output. As previously mentioned, the program only temporarily discards the neurons.

During subsequent training iterations, the program chooses different sets of neurons from the dropout layer. Although we chose a probability of 50% for dropout, the computer will not necessarily drop three neurons. It is as if we flipped a coin for each of the dropout candidate neurons to choose if that neuron was dropped out. You must know that the program should never drop the bias neuron. Only the regular neurons on a dropout layer are candidates.

The implementation of the training algorithm influences the process of discarding neurons. The dropout set frequently changes once per training iteration or batch. The program can also provide intervals where all neurons are present. Some neural network frameworks give additional hyper-parameters to allow you to specify exactly the rate of this interval.

Why dropout is capable of decreasing overfitting is a common question. The answer is that dropout can reduce the chance of a codependency developing between two neurons. Two neurons that develop a codependency will not be able to operate effectively when one is dropped out. As a result, the neural network can no longer rely on the presence of every neuron, and it trains accordingly. This characteristic decreases its ability to memorize the information presented to it, thereby forcing generalization.

Dropout also decreases overfitting by forcing a bootstrapping process upon the neural network. Bootstrapping is a very common ensemble technique. We will discuss ensembling in greater detail in Chapter 16, "Modeling with Neural Networks." Basically, ensembling is a technique of machine learning that combines multiple models to produce a better result than those achieved by individual models. Ensemble is a term that originates from the musical ensembles in which the final music product that the audience hears is the combination of many instruments.

Bootstrapping is one of the most simple ensemble techniques. The programmer using bootstrapping simply trains a number of neural networks to perform exactly the same task. However, each of these neural networks will perform differently because of some training techniques and the random numbers used in the neural network weight initialization. The difference in weights causes the performance variance. The output from this ensemble of neural networks becomes the average output of the members taken together. This process decreases overfitting through the consensus of differently trained neural networks.

Dropout works somewhat like bootstrapping. You might think of each neural network that results from a different set of neurons being dropped out as an individual member in an ensemble. As training progresses, the program creates more neural networks in this way. However, dropout does not require the same amount of processing as does bootstrapping. The new neural networks created are temporary; they exist only for a training iteration. The final result is also a single neural network, rather than an ensemble of neural networks to be averaged together.

The following animation that shows how dropout works: animation link

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
```

```
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Now we will see how to apply dropout to classification.

Code

```
########################################
# Keras with dropout for Classification
########################################

import pandas as pd
import os
import numpy as np
from sklearn import metrics
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras import regularizers

# Cross-validate
kf = KFold(5, shuffle=True, random_state=42)

oos_y = []
```

```
oos_pred = []
fold = 0

for train, test in kf.split(x):
    fold+=1
    print(f"Fold #{fold}")

    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    #kernel_regularizer=regularizers.l2(0.01),

    model = Sequential()
    model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
    model.add(Dropout(0.5))
    model.add(Dense(25, activation='relu', \
                activity_regularizer=regularizers.l1(1e-4))) # Hidden 2
    # Usually do not add dropout after final hidden layer
    #model.add(Dropout(0.5))
    model.add(Dense(y.shape[1],activation='softmax')) # Output
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    model.fit(x_train,y_train,validation_data=(x_test,y_test),\
              verbose=0,epochs=500)

    pred = model.predict(x_test)

    oos_y.append(y_test)
    # raw probabilities to chosen class (highest probability)
    pred = np.argmax(pred,axis=1)
    oos_pred.append(pred)

    # Measure this fold's accuracy
    y_compare = np.argmax(y_test,axis=1) # For accuracy calculation
    score = metrics.accuracy_score(y_compare, pred)
    print(f"Fold score (accuracy): {score}")


# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
oos_y_compare = np.argmax(oos_y,axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y_compare, oos_pred)
print(f"Final score (accuracy): {score}")
```

```
# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
#oosDF.to_csv(filename_write,index=False)
```

Output

```
Fold #1
Fold score (accuracy): 0.68
Fold #2
Fold score (accuracy): 0.695
Fold #3
Fold score (accuracy): 0.7425
Fold #4
Fold score (accuracy): 0.71
Fold #5
Fold score (accuracy): 0.6625
Final score (accuracy): 0.698
```

5.5 Part 5.5: Benchmarking Regularization Techniques

Quite a few hyperparameters have been introduced so far. Tweaking each of these values can have an effect on the score obtained by your neural networks. Some of the hyperparameters seen so far include:

- Number of layers in the neural network
- How many neurons in each layer
- What activation functions to use on each layer
- Dropout percent on each layer
- L1 and L2 values on each layer

To try out each of these hyperparameters you will need to run train neural networks with multiple settings for each hyperparameter. However, you may have noticed that neural networks often produce somewhat different results when trained multiple times. This is because the neural networks start with random weights. Because of this it is necessary to fit and evaluate a neural network times to ensure that one set of hyperparameters are actually better than another. Bootstrapping can be an effective means of benchmarking (comparing) two sets of hyperparameters.

Bootstrapping is similar to cross-validation. Both go through a number of cycles/folds providing validation and training sets. However, bootstrapping can have an unlimited number of cycles. Bootstrapping chooses a new train and validation split each cycle, with replacement. The fact that each cycle is chosen with replacement means that, unlike cross validation, there will often be repeated rows selected between cycles. If you run the bootstrap for enough cycles, there will be duplicate cycles.

In this part we will use bootstrapping for hyperparameter benchmarking. We will train a neural network for a specified number of splits (denoted by the SPLITS constant). For these examples we use 100. We will compare the average score at the end of the 100. By the end of the cycles the mean score will have converged somewhat. This ending score will be a much better basis of comparison than a single cross-validation. Additionally, the average number of epochs will be tracked to give an idea of a possible optimal value.

Because the early stopping validation set is also used to evaluate the the neural network as well, it might be slightly inflated. This is because we are both stopping and evaluating on the same sample. However, we are using the scores only as relative measures to determine the superiority of one set of hyperparameters to another, so this slight inflation should not present too much of a problem.

Because we are benchmarking, we will display the amount of time taken for each cycle. The following function can be used to nicely format a time span.

Code

```
# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

5.5.1 Additional Reading on Hyperparameter Tuning

I will add more here as I encounter additional good sources:

- A Recipe for Training Neural Networks

5.5.2 Bootstrapping for Regression

Regression bootstrapping uses the **ShuffleSplit** object to perform the splits. This is similar to **KFold** for cross validation, no balancing takes place. To demonstrate this technique we will attempt to predict the age column for the jh-simple-dataset this data is loaded by the following code.

Code

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)
```

```
# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values
```

The following code performs the bootstrap. The architecture of the neural network can be adjusted to compare many different configurations.

Code

```
import pandas as pd
import os
import numpy as np
import time
import statistics
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import ShuffleSplit

SPLITS = 50

# Bootstrap
boot = ShuffleSplit(n_splits=SPLITS, test_size=0.1, random_state=42)

# Track progress
mean_benchmark = []
epochs_needed = []
num = 0

# Loop through samples
for train, test in boot.split(x):
    start_time = time.time()
    num+=1
```

```
    # Split train and test
    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    # Construct neural network
    model = Sequential()
    model.add(Dense(20, input_dim=x_train.shape[1], activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')

    monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=5, verbose=0, mode='auto', restore_best_weights=True)

    # Train on the bootstrap sample
    model.fit(x_train,y_train,validation_data=(x_test,y_test),
              callbacks=[monitor],verbose=0,epochs=1000)
    epochs = monitor.stopped_epoch
    epochs_needed.append(epochs)

    # Predict on the out of boot (validation)
    pred = model.predict(x_test)

    # Measure this bootstrap's log loss
    score = np.sqrt(metrics.mean_squared_error(pred,y_test))
    mean_benchmark.append(score)
    m1 = statistics.mean(mean_benchmark)
    m2 = statistics.mean(epochs_needed)
    mdev = statistics.pstdev(mean_benchmark)

    # Record this iteration
    time_took = time.time() - start_time
    print(f"#{num}: score={score:.6f}, mean score={m1:.6f}, stdev={mdev:.6f},
          epochs={epochs}, mean epochs={int(m2)},
          time={hms_string(time_took)}")
```

Output

```
#1: score=0.713101, mean score=0.713101, stdev=0.000000, epochs=101,
mean epochs=101, time=0:00:14.91
#2: score=0.843020, mean score=0.778061, stdev=0.064960, epochs=123,
mean epochs=112, time=0:00:17.20
#3: score=0.793505, mean score=0.783209, stdev=0.053537, epochs=147,
mean epochs=123, time=0:00:22.03
#4: score=0.685434, mean score=0.758765, stdev=0.062786, epochs=114,
mean epochs=121, time=0:00:16.94
```

```
#5: score=0.514140, mean score=0.709840, stdev=0.112820, epochs=140,
mean epochs=125, time=0:00:20.09
#6: score=0.702493, mean score=0.708616, stdev=0.103026, epochs=116,
mean epochs=123, time=0:00:16.76
#7: score=0.602510, mean score=0.693458, stdev=0.102356, epochs=91,
mean epochs=118, time=0:00:13.23
#8: score=0.612711, mean score=0.683364, stdev=0.099399, epochs=114,

...

mean epochs=116, time=0:00:14.03
#49: score=1.007754, mean score=0.752828, stdev=0.197676, epochs=108,
mean epochs=116, time=0:00:19.17
#50: score=0.529380, mean score=0.748359, stdev=0.198174, epochs=172,
mean epochs=117, time=0:00:27.94
```

The bootstrapping process for classification is similar and is presented in the next section.

5.5.3 Bootstrapping for Classification

Regression bootstrapping uses the **StratifiedShuffleSplit** object to perform the splits. This is similar to **StratifiedKFold** for cross validation, as the classes are balanced so that the sampling has no effect on proportions. To demonstrate this technique we will attempt to predict the product column for the jh-simple-dataset this data is loaded by the following code.

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
```

```
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Code

```
import pandas as pd
import os
import numpy as np
import time
import statistics
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit

SPLITS = 50

# Bootstrap
boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1,
                              random_state=42)

# Track progress
mean_benchmark = []
epochs_needed = []
num = 0

# Loop through samples
for train, test in boot.split(x,df['product']):
    start_time = time.time()
    num+=1

    # Split train and test
    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]
```

```
    # Construct neural network
    model = Sequential()
    model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
    model.add(Dense(25, activation='relu')) # Hidden 2
    model.add(Dense(y.shape[1],activation='softmax')) # Output
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=25, verbose=0, mode='auto', restore_best_weights=True)

    # Train on the bootstrap sample
    model.fit(x_train,y_train,validation_data=(x_test,y_test),
              callbacks=[monitor],verbose=0,epochs=1000)
    epochs = monitor.stopped_epoch
    epochs_needed.append(epochs)

    # Predict on the out of boot (validation)
    pred = model.predict(x_test)

    # Measure this bootstrap's log loss
    y_compare = np.argmax(y_test,axis=1) # For log loss calculation
    score = metrics.log_loss(y_compare, pred)
    mean_benchmark.append(score)
    m1 = statistics.mean(mean_benchmark)
    m2 = statistics.mean(epochs_needed)
    mdev = statistics.pstdev(mean_benchmark)

    # Record this iteration
    time_took = time.time() - start_time
    print(f"#{num}: score={score:.6f}, mean score={m1:.6f}," +\
          f"stdev={mdev:.6f}, epochs={epochs}, mean epochs={int(m2)}," +\
          f" time={hms_string(time_took)}")
```

Output

```
#1: score=0.632018, mean score=0.632018, stdev=0.000000, epochs=100,
mean epochs=100, time=0:00:18.69
#2: score=0.655166, mean score=0.643592, stdev=0.011574, epochs=67,
mean epochs=83, time=0:00:13.48
#3: score=0.683039, mean score=0.656741, stdev=0.020859, epochs=45,
mean epochs=70, time=0:00:09.34
#4: score=0.655106, mean score=0.656332, stdev=0.018078, epochs=102,
mean epochs=78, time=0:00:19.19
#5: score=0.681267, mean score=0.661319, stdev=0.018998, epochs=57,
mean epochs=74, time=0:00:11.37
#6: score=0.687970, mean score=0.665761, stdev=0.019986, epochs=55,
mean epochs=71, time=0:00:10.77
#7: score=0.692665, mean score=0.669604, stdev=0.020761, epochs=62,
```

```
mean epochs=69, time=0:00:12.14
#8: score=0.744507, mean score=0.678967, stdev=0.031476, epochs=48,

...

mean epochs=65, time=0:00:13.92
#49: score=0.656282, mean score=0.676510, stdev=0.052215, epochs=60,
mean epochs=65, time=0:00:12.04
#50: score=0.696442, mean score=0.676908, stdev=0.051766, epochs=42,
mean epochs=64, time=0:00:08.78
```

5.5.4 Benchmarking

Now that we've seen how to bootstrap with both classification and regression we can start to try to optimize the hyperparameters for the jh-simple-dataset data. For this example we will encode for classification of the product column. Evaluation will be in log loss.

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],
               axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
```

```
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

I performed some optimization and the code is currently set to the best settings that I came up with. Later in this course we will see how we can use an automatic process to optimize the hyperparameters.

Code

```
import pandas as pd
import os
import numpy as np
import time
import tensorflow.keras.initializers
import statistics
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit
from tensorflow.keras.layers import LeakyReLU,PReLU

SPLITS = 100

# Bootstrap
boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1)

# Track progress
mean_benchmark = []
epochs_needed = []
num = 0

# Loop through samples
for train, test in boot.split(x,df['product']):
    start_time = time.time()
    num+=1

    # Split train and test
    x_train = x[train]
    y_train = y[train]
    x_test = x[test]
    y_test = y[test]

    # Construct neural network
    model = Sequential()
```

```
    model.add(Dense(100, input_dim=x.shape[1], activation=PReLU(), \
        kernel_regularizer=regularizers.l2(1e-4))) # Hidden 1
    model.add(Dropout(0.5))
    model.add(Dense(100, activation=PReLU(), \
        activity_regularizer=regularizers.l2(1e-4))) # Hidden 2
    model.add(Dropout(0.5))
    model.add(Dense(100, activation=PReLU(), \
        activity_regularizer=regularizers.l2(1e-4)
    )) # Hidden 3
#     model.add(Dropout(0.5)) - Usually better performance
# without dropout on final layer
    model.add(Dense(y.shape[1],activation='softmax')) # Output
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=100, verbose=0, mode='auto', restore_best_weights=True)

    # Train on the bootstrap sample
    model.fit(x_train,y_train,validation_data=(x_test,y_test), \
              callbacks=[monitor],verbose=0,epochs=1000)
    epochs = monitor.stopped_epoch
    epochs_needed.append(epochs)

    # Predict on the out of boot (validation)
    pred = model.predict(x_test)

    # Measure this bootstrap's log loss
    y_compare = np.argmax(y_test,axis=1) # For log loss calculation
    score = metrics.log_loss(y_compare, pred)
    mean_benchmark.append(score)
    m1 = statistics.mean(mean_benchmark)
    m2 = statistics.mean(epochs_needed)
    mdev = statistics.pstdev(mean_benchmark)

    # Record this iteration
    time_took = time.time() - start_time
    print(f"#{num}: score={score:.6f}, mean score={m1:.6f},"\
          stdev={mdev:.6f}, epochs={epochs},"\
          mean epochs={int(m2)}, time={hms_string(time_took)}")
```

Output

```
#1: score=0.667172, mean score=0.667172, stdev=0.000000, epochs=236,
mean epochs=236, time=0:01:41.02
#2: score=0.672654, mean score=0.669913, stdev=0.002741, epochs=166,
mean epochs=201, time=0:01:12.49
#3: score=0.638013, mean score=0.659280, stdev=0.015203, epochs=235,
mean epochs=212, time=0:01:37.79
#4: score=0.608298, mean score=0.646534, stdev=0.025704, epochs=236,
```

```
mean epochs=218, time=0:01:37.65
#5: score=0.713477, mean score=0.659923, stdev=0.035293, epochs=140,
mean epochs=202, time=0:00:59.98
#6: score=0.583086, mean score=0.647117, stdev=0.043104, epochs=224,
mean epochs=206, time=0:01:33.00
#7: score=0.662984, mean score=0.649383, stdev=0.040291, epochs=199,
mean epochs=205, time=0:01:22.04
#8: score=0.636290, mean score=0.647747, stdev=0.037937, epochs=264,

...

mean epochs=202, time=0:01:35.27
#99: score=0.587625, mean score=0.652611, stdev=0.046028, epochs=194,
mean epochs=202, time=0:01:32.13
#100: score=0.698883, mean score=0.653073, stdev=0.046028, epochs=148,
mean epochs=202, time=0:01:09.97
```

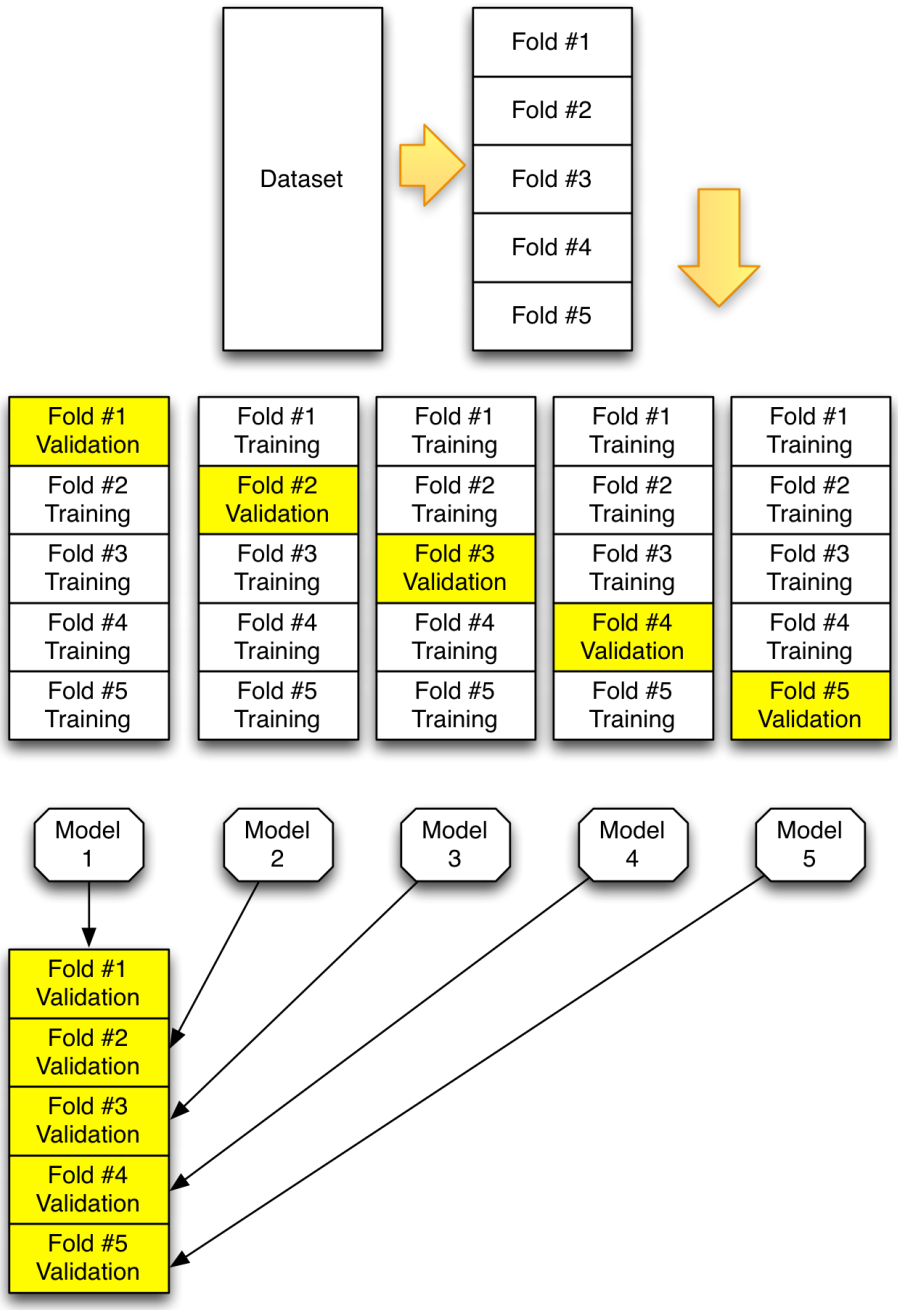


Figure 5.1: K-Fold Crossvalidation

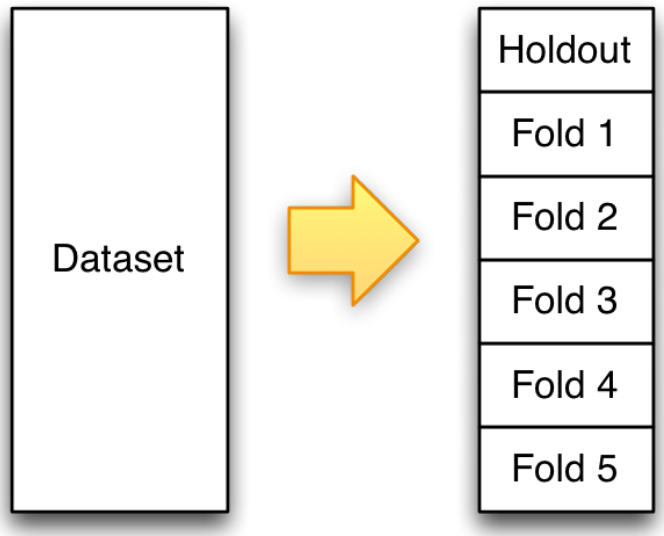


Figure 5.2: Cross Validation and a Holdout Set

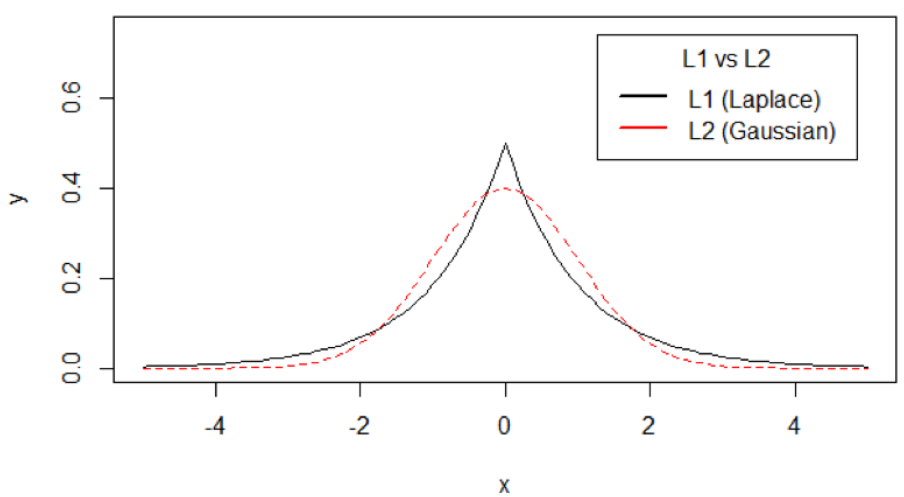


Figure 5.3: L1 vs L2

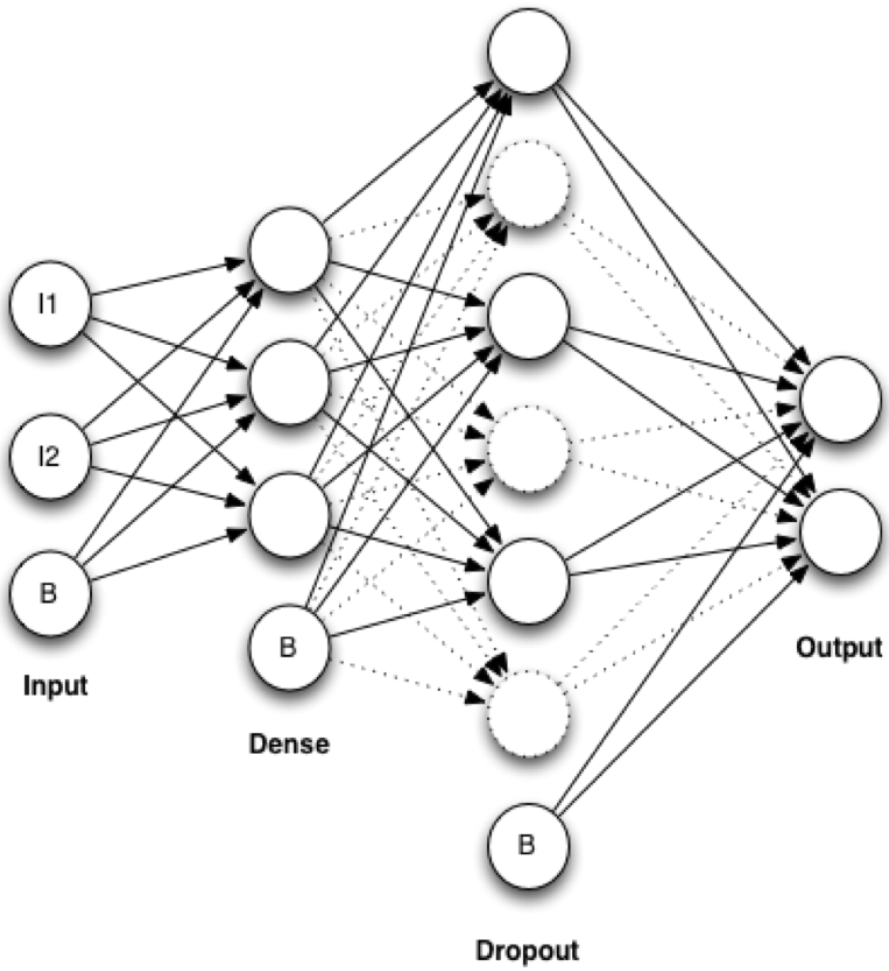


Figure 5.4: Dropout Regularization

Chapter 6

Convolutional Neural Networks (CNN) for Computer Vision

6.1 Part 6.1: Image Processing in Python

We will make use of images to demonstrate auto encoders. To use images in Python, we will make use of the Pillow package. This package can be installed with the following command.

```
pip install pillow
```

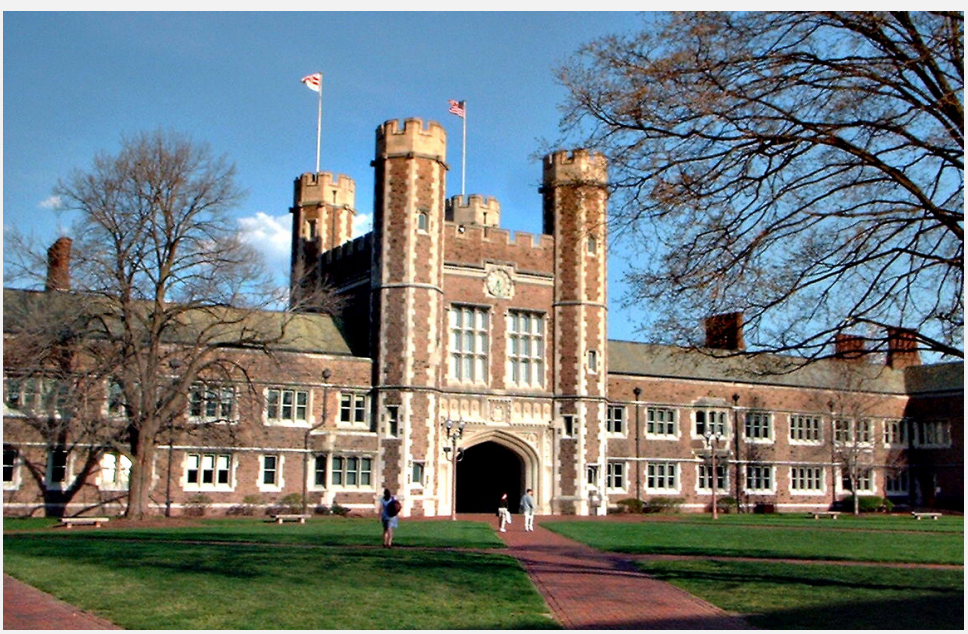
The following program uses Pillow to load and display an image.

Code

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO
import numpy as np

%matplotlib inline

url = "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))
img.load()

print(np.asarray(img))

img
```

Output

```
[[[ 86 133 177]
  [ 85 132 176]
  [ 84 133 176]
  ...
  [ 94 128 153]
  [ 91 128 155]
  [ 94 129 169]]
 [[ 86 133 177]
  [ 88 135 179]
  [ 88 137 180]
  ...
  [ 96 133 159]
  [ 92 136 165]
  [ 99 141 183]]
 [[ 83 130 174]

...

  [130  81  74]
  ...
  [ 11  17  17]
  [ 10  19  18]
  [ 10  19  18]]]
```



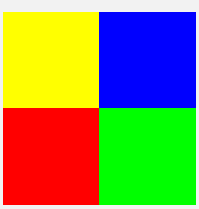
6.1.1 Creating Images (from pixels) in Python

Pillow can also be used to create an image from a 3D numpy cube. The rows and columns specify the pixels. The depth, of 3, specifies red, green and blue. Here a simple image is created.

Code

```
from PIL import Image
import numpy as np

w, h = 64, 64
data = np.zeros((h, w, 3), dtype=np.uint8)

# Yellow
for row in range(32):
    for col in range(32):
        data[row,col] = [255,255,0]

# Red
for row in range(32):
    for col in range(32):
        data[row+32,col] = [255,0,0]

# Green
for row in range(32):
    for col in range(32):
        data[row+32,col+32] = [0,255,0]

# Blue
for row in range(32):
    for col in range(32):
        data[row,col+32] = [0,0,255]


img = Image.fromarray(data, 'RGB')
img
```

Output



6.1.2 Transform Images in Python (at the pixel level)

We can combine the last two programs and modify images. Here we take the mean color of each pixel and form a grayscale image.

Code

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO

%matplotlib inline

url = "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg"
#url = "http://www.heatonresearch.com/images/about-jeff.jpg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))
img.load()

img_array = np.asarray(img)
rows = img_array.shape[0]
cols = img_array.shape[1]

print("Rows: {}, Cols: {}".format(rows,cols))

# Create new image
img2_array = np.zeros((rows, cols, 3), dtype=np.uint8)
for row in range(rows):
    for col in range(cols):
        t = np.mean(img_array[row,col])
        img2_array[row,col] = [t,t,t]

img2 = Image.fromarray(img2_array, 'RGB')
img2
```

Output

```
Rows: 744, Cols: 1157
```

6.1.3 Standardize Images

When processing several images together it is sometimes important to standardize them. The following code reads a sequence of images and causes them to all be of the same size and perfectly square. If the input images are not square, cropping will occur.

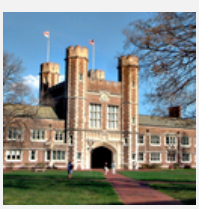
Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML

#url = "http://www.heatonresearch.com/images/about-jeff.jpg"

images = [
    "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/ff/"\
        "WashU_Graham_Chapel.JPG",
    "https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg"
]
```

```
def make_square(img):
    cols,rows = img.size
    extra = abs(rows-cols)/2

    if rows>cols:
        r = (0,extra,cols,cols+extra)
    else:
        r = (extra,0,rows+extra,rows)

    return img.crop(r)

x = []

for url in images:
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = make_square(img)
    img = img.resize((128,128), Image.ANTIALIAS)
    print(url)
    display(img)
    img_array = np.asarray(img)
    img_array = img_array.flatten()
    img_array = img_array.astype(np.float32)
    img_array = (img_array-128)/128
    x.append(img_array)


x = np.array(x)

print(x.shape)
```

Output

```
https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg
```



```
https://upload.wikimedia.org/wikipedia/commons/f/ff/WashU_Graham_Chapel.JPG
```

https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg

https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg

https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg

https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg

https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg

(7, 49152)

6.1.4 Adding Noise to an Image

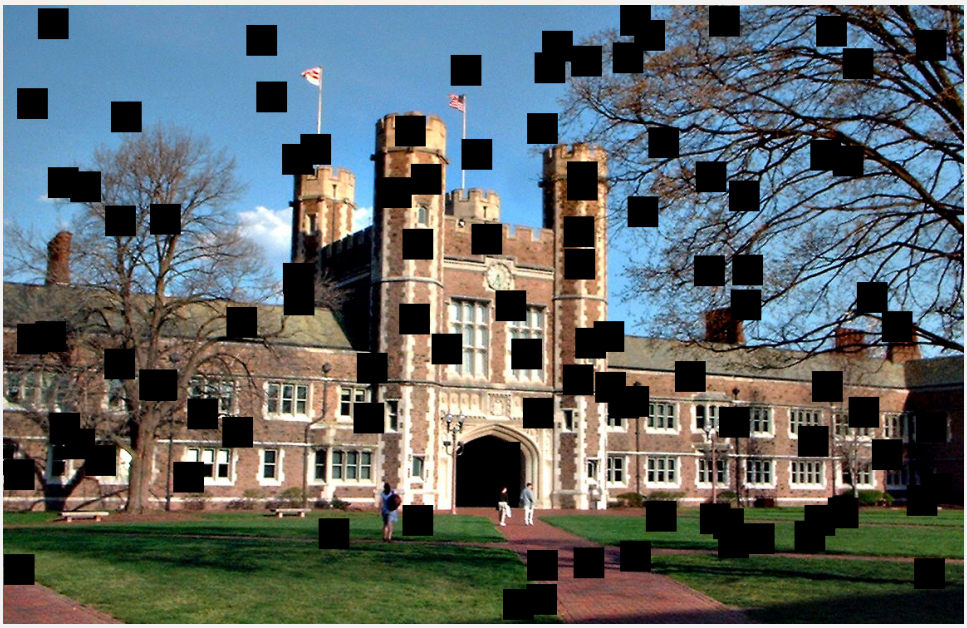
Auto encoders can handle noise. First it is important to see how to add noise to an image. There are many ways to add such noise. The following code adds random black squares to the image to produce noise.

Code

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO

%matplotlib inline


def add_noise(a):
    a2 = a.copy()
    rows = a2.shape[0]
    cols = a2.shape[1]
    s = int(min(rows,cols)/20) # size of spot is 1/20 of smallest dimension

    for i in range(100):
        x = np.random.randint(cols-s)
        y = np.random.randint(rows-s)
        a2[y:(y+s),x:(x+s)] = 0

    return a2

url = "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg"
#url = "http://www.heatonresearch.com/images/about-jeff.jpg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))
img.load()

img_array = np.asarray(img)
rows = img_array.shape[0]
cols = img_array.shape[1]

print("Rows: {}, Cols: {}".format(rows,cols))

# Create new image
img2_array = img_array.astype(np.uint8)
print(img2_array.shape)
img2_array = add_noise(img2_array)
img2 = Image.fromarray(img2_array, 'RGB')
img2
```

Output

```
Rows: 744, Cols: 1157
(744, 1157, 3)
```



6.1.5 Module 6 Assignment

You can find the first assignment here: assignment 6

6.2 Part 6.2: Keras Neural Networks for Digits and Fashion MNIST

6.2.1 Computer Vision

This class will focus on computer vision. There are some important differences and similarities with previous neural networks.

- We will usually use classification, though regression is still an option.
- The input to the neural network is now 3D (height, width, color)
- Data are not transformed, no z-scores or dummy variables.
- Processing time is much longer.
- We now have different layer times: dense layers (just like before), convolution layers and max pooling layers.
- Data will no longer arrive as CSV files. TensorFlow provides some utilities for going directly from image to the input for a neural network.

6.2.2 Computer Vision Data Sets

There are many data sets for computer vision. Two of the most popular are the MNIST digits data set and the CIFAR image data sets.

6.2.3 MNIST Digits Data Set

The MNIST Digits Data Set is very popular in the neural network research community. A sample of it can be seen in Figure 6.1.

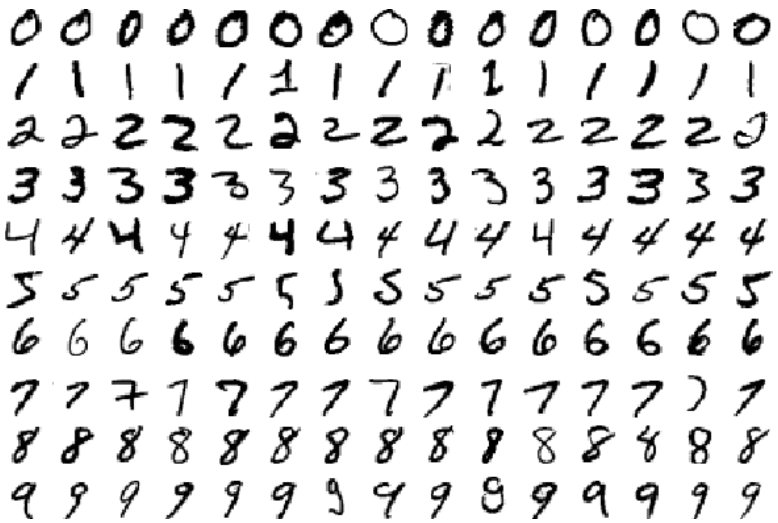


Figure 6.1: MNIST Data Set

This data set was generated from scanned forms, such as seen in Figure 6.2.

6.2.4 MNIST Fashion Data Set

Fashion-MNIST is a dataset of Zalando's article images---consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Fashion-MNIST is intended to serve as a direct **drop-in replacement** for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits. This data can be seen in Figure 6.3.

6.2.5 CIFAR Data Set

The CIFAR-10 and CIFAR-100 datasets are also frequently used by the neural network research community.

The CIFAR-10 data set contains low-rez images that are divided into 10 classes. The CIFAR-100 data set contains 100 classes in a hierarchy.

6.2.6 Other Resources

- Imagenet:Large Scale Visual Recognition Challenge 2014
- Andrej Karpathy - PhD student/instructor at Stanford.
  - CS231n Convolutional Neural Networks for Visual Recognition - Stanford course on computer vision/CNN's.
    - CS231n - GitHub
  - ConvNetJS - JavaScript library for deep learning.

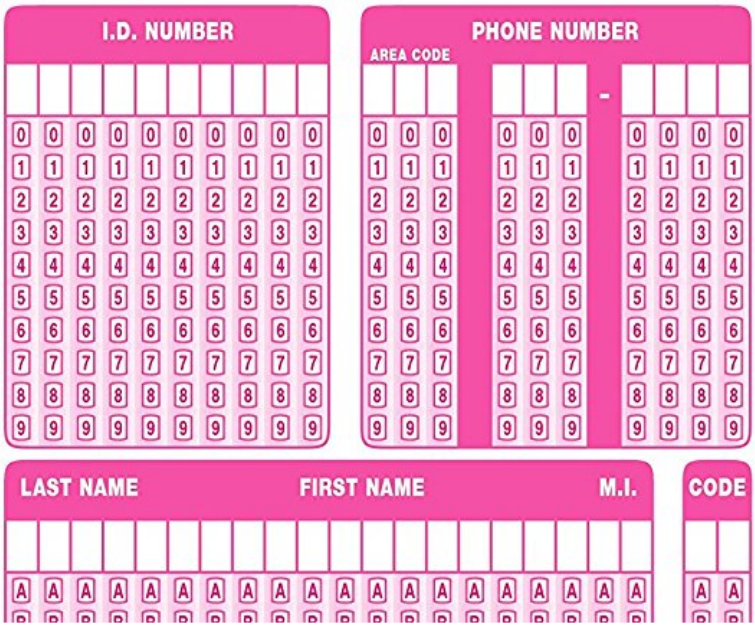


Figure 6.2: Exam Forms

6.2.7 Convolutional Neural Networks (CNNs)

The convolutional neural network (CNN) is a neural network technology that has profoundly impacted the area of computer vision (CV). Fukushima (1980)[8]introduced the original concept of a convolutional neural network, and LeCun, Bottou, Bengio & Haffner (1998)[24]greatly improved this work. From this research, Yan LeCun introduced the famous LeNet-5 neural network architecture. This chapter follows the LeNet-5 style of convolutional neural network.
Although computer vision primarily uses CNNs, this technology has some application outside of the field. You need to realize that if you want to utilize CNNs on non-visual data, you must find a way to encode your data so that it can mimic the properties of visual data.

CNNs are somewhat similar to the self-organizing map (SOM) architecture that we examined in Chapter 2, "Self-Organizing Maps." The order of the vector elements is crucial to the training. In contrast, most neural networks that are not CNNs or SOMs treat their input data as a long vector of values, and the order that you arrange the incoming features in this vector is irrelevant. For these types of neural networks, you cannot change the order after you have trained the network. In other words, CNNs and SOMs do not follow the standard treatment of input vectors.
The SOM network arranged the inputs into a grid. This arrangement worked well with images because the pixels in closer proximity to each other are important to each other. Obviously, the order of pixels in an image is significant. The human body is a relevant example of this type of order. For the design of the face, we are accustomed to eyes being near to each other. In the same way, neural network types like SOMs adhere to an order of pixels. Consequently, they have many applications to computer vision.

This advance in CNNs is due to years of research on biological eyes. In other words, CNNs utilize overlapping fields of input to simulate features of biological eyes. Until this breakthrough, AI had been unable to reproduce the capabilities of biological vision.
Scale, rotation, and noise have presented challenges in the past for AI computer vision research. You can observe the complexity of biological eyes in the example that follows. A friend raises a sheet of paper with

Figure 6.3: MNIST Fashon Data Set

a large number written on it. As your friend moves nearer to you, the number is still identifiable. In the same way, you can still identify the number when your friend rotates the paper. Lastly, your friend creates noise by drawing lines on top of the page, but you can still identify the number. As you can see, these examples demonstrate the high function of the biological eye and allow you to understand better the research breakthrough of CNNs. That is, this neural network has the ability to process scale, rotation, and noise in the field of computer vision. This network structure can be seen in Figure 6.5.

So far we have only seen one layer type (dense layers). By the end of this course we will have seen:

- **Dense Layers** - Fully connected layers. (introduced previously)
- **Convolution Layers** - Used to scan across images. (introduced this class)
- **Max Pooling Layers** - Used to downsample images. (introduced this class)
- **Dropout Layer** - Used to add regularization. (introduced next class)

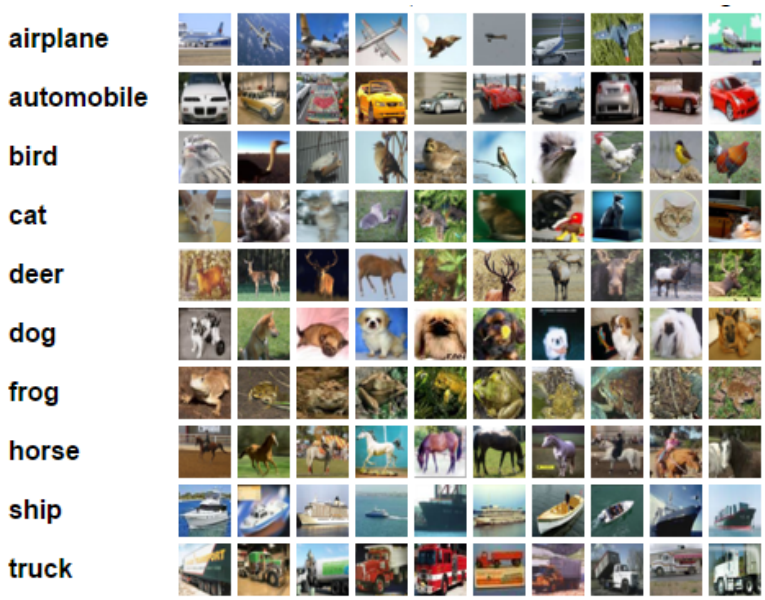


Figure 6.4: CIFAR Data Set

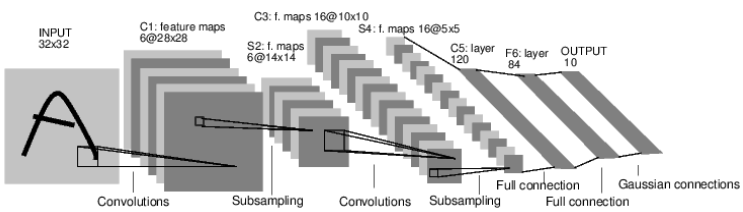


Figure 6.5: A LeNET-5 Network (LeCun, 1998)

6.2.8 Convolution Layers

The first layer that we will examine is the convolutional layer. We will begin by looking at the hyper-parameters that you must specify for a convolutional layer in most neural network frameworks that support the CNN:

- Number of filters
- Filter Size
- Stride
- Padding
- Activation Function/Non-Linearity

The primary purpose for a convolutional layer is to detect features such as edges, lines, blobs of color, and other visual elements. The filters can detect these features. The more filters that we give to a convolutional layer, the more features it can detect.

A filter is a square-shaped object that scans over the image. A grid can represent the individual pixels of a grid. You can think of the convolutional layer as a smaller grid that sweeps left to right over each row of the image. There is also a hyper parameter that specifies both the width and height of the square-shaped

filter. The following figure shows this configuration in which you see the six convolutional filters sweeping over the image grid:

A convolutional layer has weights between it and the previous layer or image grid. Each pixel on each convolutional layer is a weight. Therefore, the number of weights between a convolutional layer and its predecessor layer or image field is the following:

```
[FilterSize] * [FilterSize] * [# of Filters]
```

For example, if the filter size were 5 (5x5) for 10 filters, there would be 250 weights.

You need to understand how the convolutional filters sweep across the previous layer's output or image grid. Figure 6.6 illustrates the sweep:

0	0	0	0	0	0	0	0	0	0
0	1	3	2	8	4	2	1	3	0
0	0	5	4	8	7	3	2	1	0
0	8	1	8	4	1	3	6	2	0
0	18	4	8	1	23	2	4	17	0
0	19	8	24	14	22	10	11	12	0
0	20	62	23	9	21	6	7	4	0
0	3	13	17	5	13	16	2	8	0
0	0	0	0	0	0	0	0	0	0

Figure 6.6: Convolutional Neural Network

The above figure shows a convolutional filter with a size of 4 and a padding size of 1. The padding size is responsible for the boarder of zeros in the area that the filter sweeps. Even though the image is actually 8x7, the extra padding provides a virtual image size of 9x8 for the filter to sweep across. The stride specifies the number of positions at which the convolutional filters will stop. The convolutional filters move to the right, advancing by the number of cells specified in the stride. Once the far right is reached, the convolutional filter moves back to the far left, then it moves down by the stride amount and
continues to the right again.

Some constraints exist in relation to the size of the stride. Obviously, the stride cannot be 0. The convolutional filter would never move if the stride were set to 0. Furthermore, neither the stride, nor the convolutional filter size can be larger than the previous grid. There are additional constraints on the stride (s), padding (p) and the filter width (f) for an image of width (w). Specifically, the convolutional filter must be able to start at the far left or top boarder, move a certain number of strides, and land on the far right or bottom boarder. The following equation shows the number of steps a convolutional operator
must take to cross the image:

$$steps = \frac{w-f+2p}{s+1}$$

The number of steps must be an integer. In other words, it cannot have decimal places. The purpose of the padding (p) is to be adjusted to make this equation become an integer value.

6.2.9 Max Pooling Layers

Max-pool layers downsample a 3D box to a new one with smaller dimensions. Typically, you can always place a max-pool layer immediately following convolutional layer. The LENET shows the max-pool layer immediately after layers C1 and C3. These max-pool layers progressively decrease the size of the dimensions of the 3D boxes passing through them. This technique can avoid overfitting (Krizhevsky, Sutskever & Hinton, 2012).

A pooling layer has the following hyper-parameters:

- Spatial Extent (f)
- Stride (s)

Unlike convolutional layers, max-pool layers do not use padding. Additionally, max-pool layers have no weights, so training does not affect them. These layers simply downsample their 3D box input. The 3D box output by a max-pool layer will have a width equal to this equation:

$w_2 = \frac{w_1 - f}{s+1}$

The height of the 3D box produced by the max-pool layer is calculated similarly with this equation:

$h_2 = \frac{h_1 - f}{s+1}$

The depth of the 3D box produced by the max-pool layer is equal to the depth the 3D box received as input. The most common setting for the hyper-parameters of a max-pool layer are f =2 and s=2. The spatial extent (f) specifies that boxes of 2x2 will be scaled down to single pixels. Of these four pixels, the pixel with the maximum value will represent the 2x2 pixel in the new grid. Because squares of size 4 are replaced with size 1, 75% of the pixel information is lost. The following figure shows this transformation as a 6x6 grid becomes a 3x3:

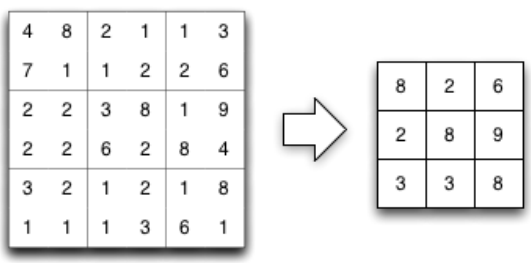


Figure 6.7: Max Pooling Layer

Of course, the above diagram shows each pixel as a single number. A grayscale image would have this characteristic. For an RGB image, we usually take the average of the three numbers to determine which pixel has the maximum value.

More information on CNN's

6.2.10 TensorFlow with CNNs

The following sections describe how to use TensorFlow/Keras with CNNs.

6.2.11 Access to Data Sets - DIGITS

Keras provides built in access classes for MNIST. It is important to note that MNIST data arrives already separated into two sets:

- **train** - Neural network will be trained with this.
- **test** - Used for validation.

Code

```
import tensorflow.keras
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
print("Shape of x_train: {}".format(x_train.shape))
print("Shape of y_train: {}".format(y_train.shape))
print()
print("Shape of x_test: {}".format(x_test.shape))
print("Shape of y_test: {}".format(y_test.shape))
```

Output

```
Shape of x_train: (60000, 28, 28)
Shape of y_train: (60000,)
Shape of x_test: (10000, 28, 28)
Shape of y_test: (10000,)
```

6.2.12 Display the Digits

The following code shows what the MNIST files contain.

Code

```
from IPython.display import display
import pandas as pd

# Display as text
pd.set_option('display.max_columns', 15)
pd.set_option('display.max_rows', 5)

print("Shape for dataset: {}".format(x_train.shape))
print("Labels: {}".format(y_train))

# Single MNIST digit
single = x_train[0]
print("Shape for single: {}".format(single.shape))
```

```
pd.DataFrame(single.reshape(28,28))
```

Output

	0	1	2	3	4	5	6	...	21	22	23	24	25	26	27
0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
26	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0

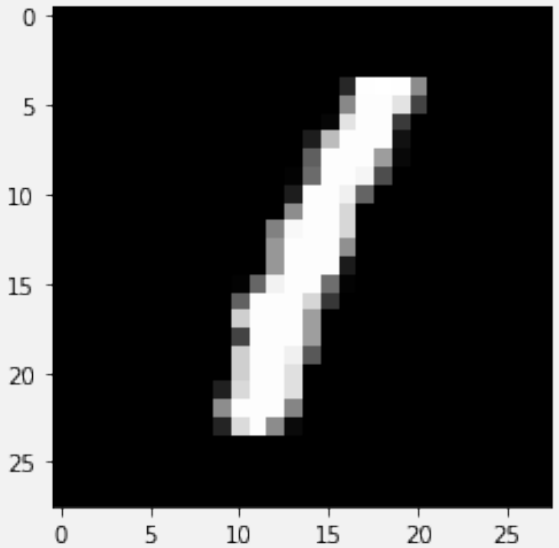
```
Shape for dataset: (60000, 28, 28)
Labels: [5 0 4 ... 5 6 8]
Shape for single: (28, 28)
```

Code

```
# Display as image
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
digit = 105 # Change to choose new digit
a = x_train[digit]
plt.imshow(a, cmap='gray', interpolation='nearest')
print("Image (#{}): Which is digit '{}'".format(digit,y_train[digit]))
```

Output

```
Image (#105): Which is digit '1'
```

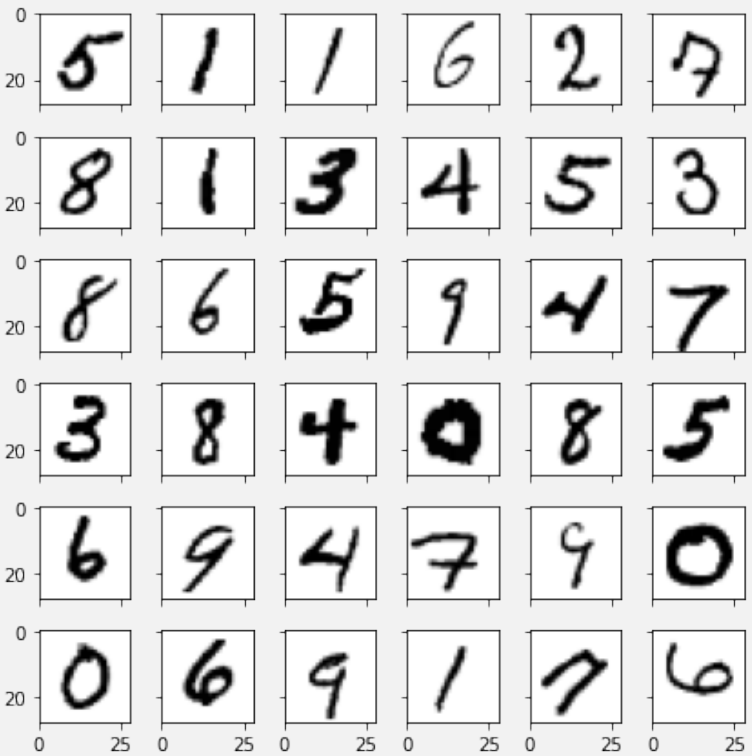


Code

```
import random

ROWS = 6
random_indices = random.sample(range(x_train.shape[0]), ROWS*ROWS)

sample_images = x_train[random_indices, :]

plt.clf()

fig, axes = plt.subplots(ROWS,ROWS,
                         figsize=(ROWS,ROWS),
                         sharex=True, sharey=True)

for i in range(ROWS*ROWS):
    subplot_row = i//ROWS
    subplot_col = i%ROWS
    ax = axes[subplot_row, subplot_col]

    plottable_image = np.reshape(sample_images[i,:], (28,28))
    ax.imshow(plottable_image, cmap='gray_r')

    ax.set_xbound([0,28])

plt.tight_layout()
plt.show()
```

Output

```
<Figure size 432x288 with 0 Axes>
```



Code

```
import tensorflow.keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import backend as K
batch_size = 128
num_classes = 10
epochs = 12
# input image dimensions
img_rows, img_cols = 28, 28
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
x_train = x_train.astype('float32')
```

```
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print("Training samples: {}".format(x_train.shape[0]))
print("Test samples: {}".format(x_test.shape[0]))
# convert class vectors to binary class matrices
y_train = tensorflow.keras.utils.to_categorical(y_train, num_classes)
y_test = tensorflow.keras.utils.to_categorical(y_test, num_classes)
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Output

```
x_train shape: (60000, 28, 28, 1)
Training samples: 60000
Test samples: 10000
```

6.2.13 Training/Fitting CNN - DIGITS

The following code will train the CNN for 20,000 steps. This can take awhile, you might want to scale the step count back. GPU training can help. My results:

- CPU Training Time: Elapsed time: 1:50:13.10
- GPU Training Time: Elapsed time: 0:13:43.06

Code

```
import tensorflow as tf
import time

start_time = time.time()

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
```

```
          verbose=2,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}'.format(score[0]))
print('Test accuracy: {}'.format(score[1]))

elapsed_time = time.time() - start_time
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

Output

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 - 4s - loss: 0.2450 - accuracy: 0.9253 - val_loss: 0.0574
- val_accuracy: 0.9818
Epoch 2/12
60000/60000 - 2s - loss: 0.0868 - accuracy: 0.9743 - val_loss: 0.0386
- val_accuracy: 0.9872
Epoch 3/12
60000/60000 - 2s - loss: 0.0633 - accuracy: 0.9806 - val_loss: 0.0343
- val_accuracy: 0.9901
Epoch 4/12
60000/60000 - 2s - loss: 0.0536 - accuracy: 0.9836 - val_loss: 0.0353
- val_accuracy: 0.9894
Epoch 5/12
60000/60000 - 2s - loss: 0.0458 - accuracy: 0.9859 - val_loss: 0.0308

...

60000/60000 - 2s - loss: 0.0225 - accuracy: 0.9923 - val_loss: 0.0287
- val_accuracy: 0.9920
Test loss: 0.028666581494135413
Test accuracy: 0.9919999837875366
Elapsed time: 0:00:26.85
```

6.2.14 Evaluate Accuracy - DIGITS

Note, if you are using a GPU you might get the **ResourceExhaustedError**. This occurs because the GPU might not have enough ram to predict the entire data set at once.

Code

```
# Predict using either GPU or CPU, send the entire dataset.
This might not work on the GPU.
# Set the desired TensorFlow output level for this example
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}'.format(score[0]))
```

```
print('Test accuracy: {}'.format(score[1]))
```

Output

```
Test loss: 0.028666581494135413
Test accuracy: 0.9919999837875366
```

GPUs are most often used for training rather than prediction. For prediction either disable the GPU or just predict on a smaller sample. If your GPU has enough memory, the above prediction code may work just fine. If not, just prediction on a sample with the following code:

Code

```
from sklearn import metrics

# For GPU just grab the first 100 images
small_x = x_test[1:100]
small_y = y_test[1:100]
small_y2 = np.argmax(small_y,axis=1)
pred = model.predict(small_x)
pred = np.argmax(pred,axis=1)
score = metrics.accuracy_score(small_y2, pred)
print('Accuracy: {}'.format(score))
```

Output

```
Accuracy: 1.0
```

6.2.15 MNIST Fashion

Code

```
import tensorflow.keras
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.datasets import fashion_mnist

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
print("Shape of x_train: {}".format(x_train.shape))
print("Shape of y_train: {}".format(y_train.shape))
print()
print("Shape of x_test: {}".format(x_test.shape))
print("Shape of y_test: {}".format(y_test.shape))
```

Output

```
Shape of x_train: (60000, 28, 28)
Shape of y_train: (60000,)
Shape of x_test: (10000, 28, 28)
Shape of y_test: (10000,)
```

6.2.16 Display the Apparel

The following code shows what the Fashion MNIST files contain.

Code

```
# Display as text
from IPython.display import display
import pandas as pd

print("Shape for dataset: {}".format(x_train.shape))
print("Labels: {}".format(y_train))

# Single MNIST digit
single = x_train[0]
print("Shape for single: {}".format(single.shape))

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.DataFrame(single.reshape(28,28))
```

Output

	0	1	2	...	25	26	27
0	0.000000	0.0	0.000000	...	0.000000	0.000000	0.000000
1	0.000000	0.0	0.000000	...	0.000000	0.000000	0.000000
2	0.000000	0.0	0.000000	...	0.000000	0.000000	0.000000
3	0.000000	0.0	0.000000	...	0.003922	0.003922	0.000000
4	0.000000	0.0	0.000000	...	0.000000	0.000000	0.011765
...	...	...	...	...	...	...	...
23	0.000000	0.0	0.290196	...	0.847059	0.666667	0.000000
24	0.007843	0.0	0.000000	...	0.227451	0.000000	0.000000
25	0.000000	0.0	0.000000	...	0.000000	0.000000	0.000000
26	0.000000	0.0	0.000000	...	0.000000	0.000000	0.000000
27	0.000000	0.0	0.000000	...	0.000000	0.000000	0.000000

```
Shape for dataset: (60000, 28, 28, 1)
Labels: [[0. 0. 0. ... 0. 0. 1.]
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]
```

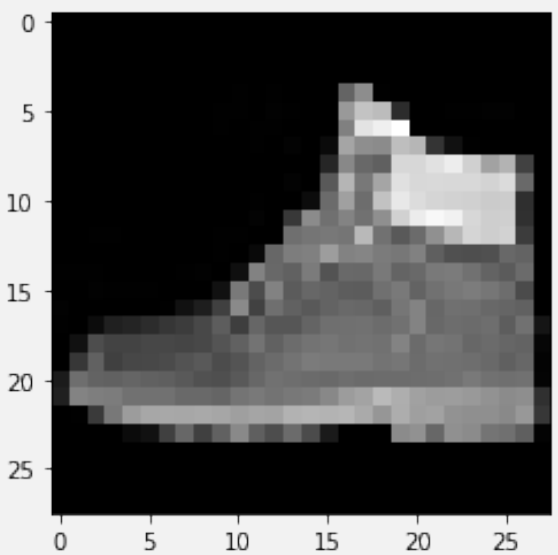
```
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [1. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]
Shape for single: (28, 28, 1)
```

Code

```
# Display as image
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
digit = 90 # Change to choose new article
a = x_train[digit]
plt.imshow(a, cmap='gray', interpolation='nearest')
print("Image (#{}): Which is digit '{}'".format(digit,y_train[digit]))
```

Output

```
Image (#90): Which is digit '9'
```



Code

```
import random

ROWS = 6
random_indices = random.sample(range(x_train.shape[0]), ROWS*ROWS)

sample_images = x_train[random_indices, :]
```

```
plt.clf()

fig, axes = plt.subplots(ROWS,ROWS,
                         figsize=(ROWS,ROWS),
                         sharex=True, sharey=True)

for i in range(ROWS*ROWS):
    subplot_row = i//ROWS
    subplot_col = i%ROWS
    ax = axes[subplot_row, subplot_col]

    plottable_image = np.reshape(sample_images[i,:], (28,28))
    ax.imshow(plottable_image, cmap='gray_r')

    ax.set_xbound([0,28])

plt.tight_layout()
plt.show()
```

Output

```
<Figure size 432x288 with 0 Axes>
```

6.2.17 Training/Fitting CNN - Fashion

The following code will train the CNN for 20,000 steps. This can take awhile, you might want to scale the step count back. GPU training can help. My results:

- CPU Training Time: Elapsed time: 1:50:13.10
- GPU Training Time: Elapsed time: 0:13:43.06

Code

```
import tensorflow.keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import backend as K
batch_size = 128
num_classes = 10
epochs = 12
# input image dimensions
img_rows, img_cols = 28, 28
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print("Training samples: {}".format(x_train.shape[0]))
print("Test samples: {}".format(x_test.shape[0]))
# convert class vectors to binary class matrices
y_train = tensorflow.keras.utils.to_categorical(y_train, num_classes)
y_test = tensorflow.keras.utils.to_categorical(y_test, num_classes)
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Output

```
x_train shape: (60000, 28, 28, 1)
Training samples: 60000
Test samples: 10000
```

Code

```
import tensorflow as tf
import time

start_time = time.time()

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=2,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}'.format(score[0]))
print('Test accuracy: {}'.format(score[1]))

elapsed_time = time.time() - start_time
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

Output

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 - 3s - loss: 0.5325 - accuracy: 0.8119 - val_loss: 0.3258
- val_accuracy: 0.8829
Epoch 2/12
60000/60000 - 2s - loss: 0.3398 - accuracy: 0.8780 - val_loss: 0.2857
- val_accuracy: 0.8937
Epoch 3/12
60000/60000 - 2s - loss: 0.2902 - accuracy: 0.8962 - val_loss: 0.2548
- val_accuracy: 0.9065
Epoch 4/12
60000/60000 - 2s - loss: 0.2612 - accuracy: 0.9051 - val_loss: 0.2453
- val_accuracy: 0.9074
Epoch 5/12
60000/60000 - 2s - loss: 0.2354 - accuracy: 0.9131 - val_loss: 0.2327
```

```
...

60000/60000 - 2s - loss: 0.1422 - accuracy: 0.9464 - val_loss: 0.2250
- val_accuracy: 0.9249
Test loss: 0.22503108531683683
Test accuracy: 0.9248999953269958
Elapsed time: 0:00:25.71
```

6.3 Part 6.3: Implementing a ResNet in Keras

Deeper neural networks are more difficult to train. Residual learning was introduced to ease the training of networks that are substantially deeper than those used previously. ResNet explicitly reformulates the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. On the ImageNet dataset this method was evaluated with residual nets with a depth of up to 152 layers---8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. This technique can also be applied to the CIFAR-10 with 100 and 1000 layers.

ResNet was introduced in the following paper:

- K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385,2015.

What is a residual?

- Residual: an internal aftereffect of experience or activity that influences later behavior

To implement a ResNet we need to give Keras the notion of a residual block. This is essentially two dense layers with a "skip connection" (or residual connection). A residual block is shown in Figure 6.8.

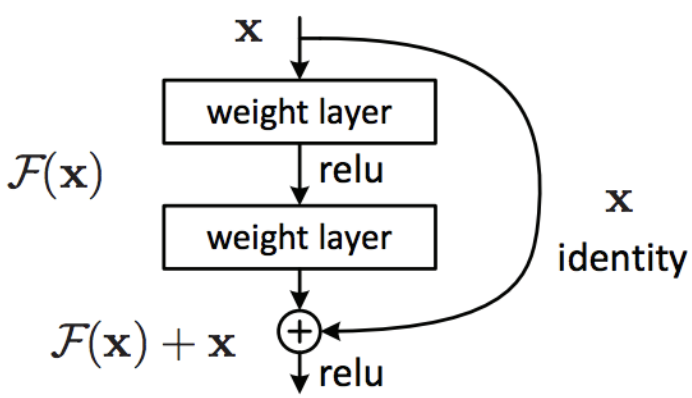


Figure 6.8: Skip Layers

Residual blocks are typically used with convolutional neural networks (CNNs). This allows very deep neural networks of CNNs to be created. Figure 6.9 shows several different ResNets.

6.3.1 Keras Sequence vs Functional Model API

Most of the neural networks create in this course have made use of the Keras sequence object. You might have noticed that we briefly made use of another type of neural network object for the ResNet, the Model. These are the two major means of constructing a neural network in Keras:

- Sequential - Simplified interface to Keras that supports most models where the flow of information is a simple sequence from input to output.
- Keras Functional API - More complex interface that allows neural networks to be constructed of reused layers, multiple input layers, and supports building your own recurrent connections.

It is important to point out that these are not two specific types of neural network. Rather, they are two means of constructing neural networks in Keras. Some types of neural network can be implemented in either, such as dense feedforward neural networks (like we used for the Iris and MPG datasets). However, other types of neural network, like ResNet and GANs can only be used in the Functional Model API.

6.3.2 CIFAR Dataset

The CIFAR-10 and CIFAR-100 datasets are also frequently used by the neural network research community. These datasets were originally part of a competition.

The CIFAR-10 data set contains low-res images that are divided into 10 classes. The CIFAR-100 data set contains 100 classes in a hierarchy.

Code

```
from __future__ import print_function
import tensorflow.keras
from tensorflow.keras.layers import Dense, Conv2D
from tensorflow.keras.layers import BatchNormalization, Activation
from tensorflow.keras.layers import AveragePooling2D, Input, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2
from tensorflow.keras import backend as K
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import cifar10
import numpy as np
import os

# Load the CIFAR10 data.
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Output

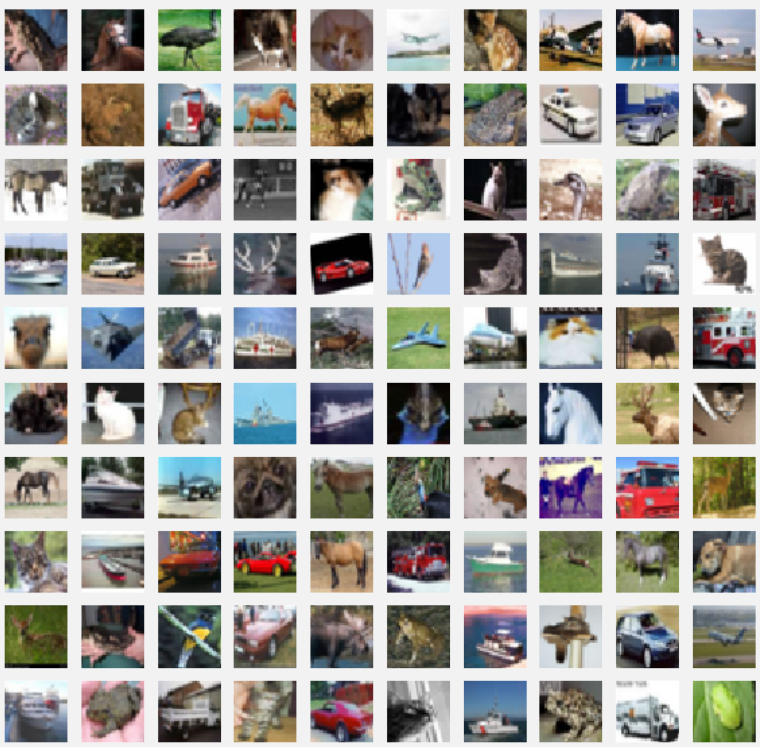
```
Downloading data from
https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 11s 0us/step
```

Samples from the loaded CIFAR dataset can be displayed using the following code.

Code

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from six.moves import cPickle

ROWS = 10

x = x_train.astype("uint8")

fig, axes1 = plt.subplots(ROWS,ROWS, figsize=(10,10))
for j in range(ROWS):
    for k in range(ROWS):
        i = np.random.choice(range(len(x)))
        axes1[j][k].set_axis_off()
        axes1[j][k].imshow(x[i:i+1][0])
```

Output



We will construct a ResNet and train it on the CIFAR-10 dataset. The following block of code defines some constant values that define how the network is constructed.

Code

```
# Training parameters
BATCH_SIZE = 32  # orig paper trained all networks with batch_size=128
EPOCHS = 200 # 200
USE_AUGMENTATION = True
NUM_CLASSES = np.unique(y_train).shape[0] # 10
COLORS = x_train.shape[3]

# Subtracting pixel mean improves accuracy
SUBTRACT_PIXEL_MEAN = True

# Model version
# Orig paper: version = 1 (ResNet v1),
# Improved ResNet: version = 2
# (ResNet v2)
VERSION = 1

# Computed depth from supplied model parameter n
if VERSION == 1:
    DEPTH = COLORS * 6 + 2
elif version == 2:
    DEPTH = COLORS * 9 + 2
```

The following function implements a learning rate decay schedule.

Code

```
def lr_schedule(epoch):
    """Learning Rate Schedule

    Learning rate is scheduled to be reduced after 80, 120, 160, 180 epochs.
    Called automatically every epoch as part of callbacks during training.

    # Arguments
        epoch (int): The number of epochs

    # Returns
        lr (float32): learning rate
    """
    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:
        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1
```

```
    print('Learning rate: ', lr)
    return lr
```

The following code implements a ResNet block. This includes two convolutional layers with a skip connection. Both V1 and V2 of ResNet make use of this type of layer.

Code

```
def resnet_layer(inputs,
                 num_filters=16,
                 kernel_size=3,
                 strides=1,
                 activation='relu',
                 batch_normalization=True,
                 conv_first=True):
    """2D Convolution-Batch Normalization-Activation stack builder

    # Arguments
        inputs (tensor): input tensor from input image or previous layer
        num_filters (int): Conv2D number of filters
        kernel_size (int): Conv2D square kernel dimensions
        strides (int): Conv2D square stride dimensions
        activation (string): activation name
        batch_normalization (bool): whether to include batch normalization
        conv_first (bool): conv-bn-activation (True) or
            bn-activation-conv (False)

    # Returns
        x (tensor): tensor as input to the next layer
    """
    conv = Conv2D(num_filters,
                  kernel_size=kernel_size,
                  strides=strides,
                  padding='same',
                  kernel_initializer='he_normal',
                  kernel_regularizer=l2(1e-4))

    x = inputs
    if conv_first:
        x = conv(x)
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
    else:
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
```

```
        x = conv(x)
    return x
```

6.3.3 ResNet V1

- K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385,2015.

Code

```
def resnet_v1(input_shape, depth, num_classes=10):
    """ResNet Version 1 Model builder [a]

    Stacks of 2 x (3 x 3) Conv2D-BN-ReLU
    Last ReLU is after the shortcut connection.
    At the beginning of each stage, the feature
    map size is halved (downsampled)
    by a convolutional layer with strides=2, while the number of
    filters is
    doubled. Within each stage, the layers have the same number
    filters and the same number of filters.
    Features maps sizes:
    stage 0: 32x32, 16
    stage 1: 16x16, 32
    stage 2:  8x8,  64
    The Number of parameters is approx the same as Table 6 of [a]:
    ResNet20 0.27M
    ResNet32 0.46M
    ResNet44 0.66M
    ResNet56 0.85M
    ResNet110 1.7M

    # Arguments
        input_shape (tensor): shape of input image tensor
        depth (int): number of core convolutional layers
        num_classes (int): number of classes (CIFAR10 has 10)

    # Returns
        model (Model): Keras model instance
    """
    if (depth - 2) % 6 != 0:
        raise ValueError('depth should be 6n+2 (eg 20, 32, 44 in [a]) ')
    # Start model definition.
    num_filters = 16
    num_res_blocks = int((depth - 2) / 6)

    inputs = Input(shape=input_shape)
    x = resnet_layer(inputs=inputs)
```

```
    # Instantiate the stack of residual units
    for stack in range(3):
        for res_block in range(num_res_blocks):
            strides = 1
            # first layer but not first stack
            if stack > 0 and res_block == 0:
                strides = 2  # downsample
            y = resnet_layer(inputs=x,
                             num_filters=num_filters,
                             strides=strides)
            y = resnet_layer(inputs=y,
                             num_filters=num_filters,
                             activation=None)
            # first layer but not first stack
            if stack > 0 and res_block == 0:
                # linear projection residual shortcut connection to match
                # changed dims
                x = resnet_layer(inputs=x,
                                 num_filters=num_filters,
                                 kernel_size=1,
                                 strides=strides,
                                 activation=None,
                                 batch_normalization=False)
            x = tensorflow.keras.layers.add([x, y])
            x = Activation('relu')(x)
        num_filters *= 2

    # Add classifier on top.
    # v1 does not use BN after last shortcut connection-ReLU
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                    activation='softmax',
                    kernel_initializer='he_normal')(y)

    # Instantiate model.
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

6.3.4 ResNet V2

A second version of ResNet was introduced in the following paper. This form of ResNet is commonly referred to as ResNet V2.

- He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European conference on computer vision (pp. 630-645). Springer, Cham.

The following code constructs a ResNet V2 network. The primary difference of the full preactivation 'v2' variant compared to the 'v1' variant is the use of batch normalization before every weight layer.

Code

```
def resnet_v2(input_shape, depth, num_classes=10):
    """ResNet Version 2 Model builder [b]

    Stacks of (1 x 1)-(3 x 3)-(1 x 1) BN-ReLU-Conv2D or also known as
    bottleneck layer
    First shortcut connection per layer is 1 x 1 Conv2D.
    Second and onwards shortcut connection is identity.
    At the beginning of each stage, the feature map size is
    halved (downsampled) by a convolutional layer with
    strides=2, while the number of filter maps is
    doubled. Within each stage, the layers have the same
    number filters and the same filter map sizes.
    Features maps sizes:
    conv1  : 32x32,  16
    stage 0: 32x32,  64
    stage 1: 16x16, 128
    stage 2:  8x8,  256

    # Arguments
        input_shape (tensor): shape of input image tensor
        depth (int): number of core convolutional layers
        num_classes (int): number of classes (CIFAR10 has 10)

    # Returns
        model (Model): Keras model instance
    """
    if (depth - 2) % 9 != 0:
        raise ValueError('depth should be 9n+2 (eg 56 or 110 in [b])')
    # Start model definition.
    num_filters_in = 16
    num_res_blocks = int((depth - 2) / 9)

    inputs = Input(shape=input_shape)
    # v2 performs Conv2D with BN-ReLU on input before splitting into 2 paths
    x = resnet_layer(inputs=inputs,
                     num_filters=num_filters_in,
                     conv_first=True)

    # Instantiate the stack of residual units
    for stage in range(3):
        for res_block in range(num_res_blocks):
            activation = 'relu'
            batch_normalization = True
            strides = 1
            if stage == 0:
                num_filters_out = num_filters_in * 4
                if res_block == 0:  # first layer and first stage
```

```
                activation = None
                batch_normalization = False
        else:
            num_filters_out = num_filters_in * 2
            if res_block == 0:  # first layer but not first stage
                strides = 2    # downsample

        # bottleneck residual unit
        y = resnet_layer(inputs=x,
                         num_filters=num_filters_in,
                         kernel_size=1,
                         strides=strides,
                         activation=activation,
                         batch_normalization=batch_normalization,
                         conv_first=False)
        y = resnet_layer(inputs=y,
                         num_filters=num_filters_in,
                         conv_first=False)
        y = resnet_layer(inputs=y,
                         num_filters=num_filters_out,
                         kernel_size=1,
                         conv_first=False)
        if res_block == 0:
            # linear projection residual shortcut connection to match
            # changed dims
            x = resnet_layer(inputs=x,
                             num_filters=num_filters_out,
                             kernel_size=1,
                             strides=strides,
                             activation=None,
                             batch_normalization=False)
        x = tensorflow.keras.layers.add([x, y])

    num_filters_in = num_filters_out

# Add classifier on top.
# v2 has BN-ReLU before Pooling
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = AveragePooling2D(pool_size=8)(x)
y = Flatten()(x)
outputs = Dense(num_classes,
                activation='softmax',
                kernel_initializer='he_normal')(y)

# Instantiate model.
model = Model(inputs=inputs, outputs=outputs)
return model
```

With all of this defined, we can run the ResNet.

Code

```
# Input image dimensions.
input_shape = x_train.shape[1:]

# Normalize data.
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# If subtract pixel mean is enabled
if SUBTRACT_PIXEL_MEAN:
    x_train_mean = np.mean(x_train, axis=0)
    x_train -= x_train_mean
    x_test -= x_train_mean

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print('y_train shape:', y_train.shape)

# Convert class vectors to binary class matrices.
y_train = tensorflow.keras.utils.to_categorical(y_train, NUM_CLASSES)
y_test = tensorflow.keras.utils.to_categorical(y_test, NUM_CLASSES)

# Create the neural network
if VERSION == 2:
    model = resnet_v2(input_shape=input_shape, depth=DEPTH)
else:
    model = resnet_v1(input_shape=input_shape, depth=DEPTH)

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=lr_schedule(0)),
              metrics=['accuracy'])
model.summary()
```

Output

```
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
y_train shape: (50000, 1)
Learning rate:  0.001
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Connected to
```

```
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]   0
_________________________________________________________________

...

Total params: 274,442
Trainable params: 273,066
Non-trainable params: 1,376
_________________________________________________________________
```

Code

```
import time

start_time = time.time()

# Prepare callbacks for model saving and for learning rate adjustment.
lr_scheduler = LearningRateScheduler(lr_schedule)

lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                               cooldown=0,
                               patience=5,
                               min_lr=0.5e-6)

callbacks = [lr_reducer, lr_scheduler]

# Run training, with or without data augmentation.
if not USE_AUGMENTATION:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=BATCH_SIZE,
              epochs=EPOCHS,
              validation_data=(x_test, y_test),
              shuffle=True,
              callbacks=callbacks)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        # set input mean to 0 over the dataset
        featurewise_center=False,
        # set each sample mean to 0
        samplewise_center=False,
        # divide inputs by std of dataset
```

```
        featurewise_std_normalization=False,
        # divide each input by its std
        samplewise_std_normalization=False,
        # apply ZCA whitening
        zca_whitening=False,
        # epsilon for ZCA whitening
        zca_epsilon=1e-06,
        # randomly rotate images in the range (deg 0 to 180)
        rotation_range=0,
        # randomly shift images horizontally
        width_shift_range=0.1,
        # randomly shift images vertically
        height_shift_range=0.1,
        # set range for random shear
        shear_range=0.,
        # set range for random zoom
        zoom_range=0.,
        # set range for random channel shifts
        channel_shift_range=0.,
        # set mode for filling points outside the input boundaries
        fill_mode='nearest',
        # value used for fill_mode = "constant"
        cval=0.,
        # randomly flip images
        horizontal_flip=True,
        # randomly flip images
        vertical_flip=False,
        # set rescaling factor (applied before any other transformation)
        rescale=None,
        # set function that will be applied on each input
        preprocessing_function=None,
        # image data format, either "channels_first" or "channels_last"
        data_format=None,
        # fraction of images reserved for validation
        # (strictly between 0 and 1)
        validation_split=0.0)

    # Compute quantities required for featurewise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train,
                        batch_size=BATCH_SIZE),
                        validation_data=(x_test, y_test),
                        epochs=EPOCHS, verbose=0, workers=1,
                        callbacks=callbacks,
                        use_multiprocessing=False)
```

```
elapsed_time = time.time() - start_time
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

Output

```
Using real-time data augmentation.
WARNING:tensorflow:From <ipython-input-10-63a1fd529cab>:77:
Model.fit_generator (from tensorflow.python.keras.engine.training) is
deprecated and will be removed in a future version.
Instructions for updating:
Please use Model.fit, which supports generators.
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Learning rate:  0.001
Learning rate:  0.001
Learning rate:  0.001
Learning rate:  0.001
Learning rate:  0.001

...

Learning rate:  5e-07
Learning rate:  5e-07
Learning rate:  5e-07
Learning rate:  5e-07
Elapsed time: 2:22:14.51
```

The trained neural network can now be evaluated.

Code

```
# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

Output

```
10000/10000 [==============================] - 2s 208us/sample - loss:
0.5049 - accuracy: 0.9119
Test loss: 0.5048766820669174
Test accuracy: 0.9119
```

6.4 Part 6.4: Using Your Own Images with Keras

So far we've used image data sets that Keras provides convenience functions for accessing. There are a number of built-in data sets for Keras. While these convenience functions do make it easier to create Keras models for these data sets, these functions also hide the internal workings. You might be wondering how you would train a neural network from your own sets of images.

Consider the convenience functions provided for CIFAR-10:

Code

```
from tensorflow.keras.datasets import cifar10
import numpy as np

# Load the CIFAR10 data.
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

The above code extracts the training and test sets from CIFAR-10. Often these datasets are already pre-split between test and training data. This allows comparison between many researchers who are working on models for this data. Without these splits, it would be difficult to compare accuracy results between two different researchers that were using two different train/test splits. Consider the shape of the dataset.

Code

```
x_train.shape
```

Output

```
(50000, 32, 32, 3)
```

We are provided with 50,000 training elements. Each training element is an image that is 32x32 pixels with 3 color channels. Typically you will either see 1 color channel (grayscale) or 3 color channels (RGB color).

If we look inside of one of the 50,000 elements we can see the structure of each image. It is a matrix of RGB values.

Code

```
x_train[0]
```

Output

```
array([[[ 59,  62,  63],
        [ 43,  46,  45],
        [ 50,  48,  43],
        ...,
        [158, 132, 108],
        [152, 125, 102],
        [148, 124, 103]],
       [[ 16,  20,  20],
```

```
        [  0,   0,   0],
        [ 18,   8,   0],
        ...,
        [123,  88,  55],
        [119,  83,  50],
        [122,  87,  57]],
       [[ 25,  24,  21],

...

        [179, 142,  87],
        ...,
        [216, 184, 140],
        [151, 118,  84],
        [123,  92,  72]]], dtype=uint8)
```

It is also important to note that the data type is uint8, which is unsigned integer 8-bits (1 byte). This corresponds well with image binary data (that is typically 24-bit, 8 bits per 3 color channel = 24 bit). However, while the images may be 8-bit based, neural networks typically expect floating point input. Because of this, some transformation/normalization of the data is needed.

When training, it is usually necessary to handle multiple images at a time. The code presented here will load in multiple images and convert them so that they are all the same size. Processing training data images so that each image is of a uniform height and width is a very common step for computer vision programs.

Code

```
training_data = []

%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML

IMAGE_WIDTH = 200
IMAGE_HEIGHT = 200
IMAGE_CHANNELS = 3

images = [
    "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/ff/"\
    "WashU_Graham_Chapel.JPG",
    "https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg"
```

```
]


def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
        img = img.crop((pad,0,cols,cols))
    else:
        pad = (cols-rows)/2
        img = img.crop((0,pad,rows,rows))

    return img

for url in images:
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = make_square(img)
    img = img.resize((IMAGE_WIDTH,IMAGE_HEIGHT),Image.ANTIALIAS)
    training_data.append(np.asarray(img))
```

The above code contains a function called **make_square** that ensures that each of the images have the same height and width. Square images are particularly easy to deal with because each image will have the same aspect ratio. There are several techniques that can be used to make an image square. Fundamentally the image will either be cropped or padded to make it square. Padding adds extra space to the image to cause a square shape. Cropping removes pixels (and therefore information) from the images to make them square.

The technique above crops the images to make them square. The row and column sizes are analyzed and the image is adjusted based on if the row or column size is smaller. If there are fewer rows and than columns then the extra columns are dropped to cause the row and column count to be equal. Similarly, if there are fewer columns and than rows then the extra columns are rows to cause the row and column count to be equal.

For each image in the set, the image is first adjusted to be square and then resized to the common size that we are forcing all images to be. The resulting images are each added to a list. The training data, at this point, is shown below.

Code

```
training_data
```

Output

```
[array([[[112, 161, 203],
         [112, 161, 203],
         [114, 162, 203],
```

```
          ...,
          [ 86, 121, 162],
          [ 49,  85, 130],
          [ 76, 113, 154]],
         [[106, 158, 203],
          [109, 161, 206],
          [113, 162, 203],
          ...,
          [ 75,  95, 125],
          [ 67,  87, 124],
          [ 24,  38,  73]],
         [[108, 159, 205],

...

          [178, 119,  59],
          ...,
          [207, 151,  82],
          [209, 152,  78],
          [211, 153,  84]]], dtype=uint8)]
```

We now have a 1D list of 3D images (height by width by color depth). We would like this to be a 4D Numpy array. Passing the list to **np.array** performs this conversion/reshaping.

The training data is divided by 127.5 and subtracted by one to normalize to between -1 and 1. This causes the RGB values to be centered around zero and gives greater predictive power to the neural network.

Code

```
training_data = np.array(training_data) / 127.5 - 1.
```

We can display the normalized training data:

Code

```
training_data
```

Output

```
array([[[[-0.12156863,  0.2627451 ,  0.59215686],
         [-0.12156863,  0.2627451 ,  0.59215686],
         [-0.10588235,  0.27058824,  0.59215686],
         ...,
         [-0.3254902 , -0.05098039,  0.27058824],
         [-0.61568627, -0.33333333,  0.01960784],
         [-0.40392157, -0.11372549,  0.20784314]],
        [[-0.16862745,  0.23921569,  0.59215686],
         [-0.14509804,  0.2627451 ,  0.61568627],
         [-0.11372549,  0.27058824,  0.59215686],
         ...,
```

```
          [-0.41176471, -0.25490196, -0.01960784],
          [-0.4745098 , -0.31764706, -0.02745098],
          [-0.81176471, -0.70196078, -0.42745098]],
         [[-0.15294118,  0.24705882,  0.60784314],

...

          [ 0.39607843, -0.06666667, -0.5372549 ],
          ...,
          [ 0.62352941,  0.18431373, -0.35686275],
          [ 0.63921569,  0.19215686, -0.38823529],
          [ 0.65490196,  0.2       , -0.34117647]]])
```

It is sometimes useful to to save a training set. For image and higher dimensional data, as CSV file is not sufficient. Also, Pickle can experience problems with very large datasets. Because of this I prefer to use Numpy's own format for binary data.

Code

```
print("Saving training image binary...")
np.save("training",training_data) # Saves as "training.npy"
print("Done.")
```

Output

```
Saving training image binary...
Done.
```

6.5 Part 6.5: Recognizing Multiple Images with Darknet

Programmers typically design convolutional neural networks to classify a single item centered in an image. However, as humans, we can recognize many items in our field of view in real-time. It is advantageous to be able to recognize multiple items in a single image. One of the most advanced means of doing this is YOLO DarkNet (not to be confused with the Internet Darknet. YOLO[35]is an acronym for You Only Look Once. The fact that YOLO must only look once speaks to the efficiency of the algorithm. In this context, to "look" means to perform one scan over the image. Figure 6.10 shows YOLO tagging in action. It is also possible to run YOLO on live video streams.

As you can see, it is classifying many things in this video. My collection of books behind me is adding considerable "noise," as DarkNet tries to classify every book behind me. If you watch the video, you can see that it is less than perfect. The coffee mug that I pick up gets classified as a cell phone and, at times, a remote. The small yellow object behind me on the desk is a small toolbox (not a remote). However, it gets classified as a book at times and a remote at other times. Currently, this algorithm classifies each frame on its own. The program could achieve greater accuracy if it analyzed multiple images from a video stream. Consider when you see an object coming towards you, if it changes angles, you might form a better opinion of what it was. If that same object now changes to an unfavorable angle, you still know what it is, based on previous information.

6.5.1 How Does DarkNet/YOLO Work?

YOLO begins by resizing the image to a $S \times S$ grid. YOLO runs a single convolutional neural network against this grid that predicts bounding boxes and what might be contained by those boxes. Each bounding box also has a confidence in which item it believes the box contains. YOLO is a regular convolution network, just like we've seen previously. The only difference is that a YOLO CNN outputs multiple prediction bounding boxes. At a high level, Figure 6.11 illustrates this.

The output of the YOLO convolutional neural networks is essentially a multiple regression. YOLO generated the following values for each of the bounding rectangles.

- **x** - The x-coordinate of the center of a bounding rectangle.
- **y** - The y-coordinate of the center of a bounding rectangle.
- **w** - The width of each bounding rectangle.
- **h** - The height of each bounding rectangle.
- **labels** - The relative probabilities of each of the labels (1 value for each label)
- **confidence** - The confidence in this rectangle.

The output layer of a Keras neural network is a Tensor. In the case of YOLO, this output tensor is 3D and is of the following dimensions.

$$S \times S \times (B \cdot 5 + C)$$

The constants in the above expression are:

- S - The dimensions that YOLO overlays across the source image.
- B - The number of potential bounding rectangles generated for each grid cell.
- C - The number of class labels that here are.

The value 5 in the above expression is simply the count of non-label components of each bounding rectangle (x, y, h, w, $confidence$.

Because there are $S^2 \cdot B$ total potential bounding rectangles, the image is nearly full. Because of this, it is essential to drop all rectangles below some threshold of confidence. The image below demonstrates this.

The actual structure of the convolutional neural network behind YOLO is relatively simple, as the following figure illustrates. Because there is only one convolutional neural network, and it "only looks once," the performance is not impacted by how many objects are detected. Figure 6.12 shows the YOLO structure.

6.5.2 Using YOLO in Python

To make use of YOLO in Python, you have several options:

- - The original implementation of YOLO, written in C.
- - An unofficial Python package that implements YOLO in Python, using TensorFlow 2.0.

The code provided in this notebook works equally well when run either locally or from Google CoLab. In either case, the programmer should use TensorFlow 2.0.

6.5.3 Installing YoloV3-TF2

YoloV3-TF2 is not available directly through either PIP or CONDA. Additionally, YoloV3-TF2 is not installed in Google CoLab by default. Therefore, whether you wish to use YoloV3-TF2 through CoLab or run it locally, you need to go through several steps to install it. This section describes the process of installing YoloV3-TF2. The same steps apply to either CoLab or a local install. For CoLab, you must repeat these

steps each time the system restarts your virtual environment. For a local install, you must perform these steps only once for your virtual Python environment. If you are installing locally, make sure to install to the same virtual environment that you created for this course. The following command installs YoloV3-TF2 directly from it's GitHub repository.

Code

```
import sys

!{sys.executable} -m pip install git+https://github.com/
        zzh8829/yolov3-tf2.git@master
```

Output

```
Collecting git+https://github.com/zzh8829/yolov3-tf2.git@master
  Cloning https://github.com/zzh8829/yolov3-tf2.git (to revision
master) to
/private/var/folders/nz/jw1_lq4s10389lbx_j_vjyg4qy7h5_/T/pip-req-
build-plnt65lr
  Running command git clone -q
https://github.com/zzh8829/yolov3-tf2.git
/private/var/folders/nz/jw1_lq4s10389lbx_j_vjyg4qy7h5_/T/pip-req-
build-plnt65lr
Requirement already satisfied (use --upgrade to upgrade):
yolov3-tf2==0.1 from
git+https://github.com/zzh8829/yolov3-tf2.git@master in
/Users/jheaton/miniconda3/envs/tensorflow/lib/python3.7/site-packages
Building wheels for collected packages: yolov3-tf2
  Building wheel for yolov3-tf2 (setup.py) ... done

...

  Stored in directory:
/private/var/folders/nz/jw1_lq4s10389lbx_j_vjyg4qy7h5_/T/pip-ephem-
wheel-cache-6ks2ceq_/wheels/59/1b/97/905ab51e9c0330efe8c3c518aff17de4e
e91100412cd6dd553
Successfully built yolov3-tf2
```

Before you can make use of YoloV3-TF2 there are several files you must obtain:

- **yolov3.weights** - These are the pre-trained weights provided by the author of YOLO.
- **convert.py** - This is a Python script that converts **yolov3.weights** into a TensorFlow compatible weight format.
- **coco.names** - The names of the 80 items that the **yolov3.weights** neural network was trained to recognize.
- **yolov3.tf** - The YOLO weights converted to a format that TensorFlow can use directly.

The code provided below obtains these files. The script stores these files to either your GDrive, if you are using CoLab, or a local folder named "data" if you are running locally.

Researchers have trained YOLO on a variety of different computer image datasets. The version of YOLO weights used in this course is from the dataset Common Objects in Context (COCO).[27]This dataset contains images labeled into 80 different classes. COCO is the source of the file coco.txt that used in this module.

Developers have also adapted YOLO for mobile devices by creating the YOLO Tiny pre-trained weights that use a much smaller convolutional neural network and still achieve acceptable levels of quality. Though YoloV3-TF2 can work with either YOLO Tiny or regular YOLO we are not using the tiny weights for this course.

Code

```
import tensorflow as tf
import os

if COLAB:
  ROOT = '/content/drive/My Drive/projects/t81_558_dlearning/yolo'
else:
  ROOT = os.path.join(os.getcwd(),'data')

filename_darknet_weights = tf.keras.utils.get_file(
    os.path.join(ROOT,'yolov3.weights'),
    origin='https://pjreddie.com/media/files/yolov3.weights')
TINY = False

filename_convert_script = tf.keras.utils.get_file(
    os.path.join(os.getcwd(),'convert.py'),
    origin='https://raw.githubusercontent.com/zzh8829/'\
        "yolov3-tf2/master/convert.py')

filename_classes = tf.keras.utils.get_file(
    os.path.join(ROOT,'coco.names'),
    origin='https://raw.githubusercontent.com/zzh8829/'\
        'yolov3-tf2/master/data/coco.names')
filename_converted_weights = os.path.join(ROOT,'yolov3.tf')
```

Output

```
Downloading data from
https://raw.githubusercontent.com/zzh8829/yolov3-tf2/master/convert.py
8192/1067 [=============================================================
=============================================================
=============================================================
==============================] - 0s 0us/step
```

6.5.4 Transfering Weights

In the course, we transfer already trained weights into our YOLO networks. It can take considerable time to train a YOLO network from scratch. If you would like to train a YOLO network to recognize images other than the COLO provided images, then you may need to train your own YOLO information. If training

from scratch is something you need to do, there is further information on this at the YoloV3-TF2 GitHub repository.

The weights provided by the original authors of YOLO is not directly compatible with TensorFlow. Because of this, it is necessary first to convert the YOLO provided weights into a TensorFlow compatible format. The following code does this conversion. This process does not need to be repeated by the program. Once the conversion script processes YOLO weights the saved to the yolov3.tf YOLO can reuse these converted wights. The following code performs this conversion.

Code

```
import sys
!{sys.executable} "{filename_convert_script}" --weights
    "{filename_darknet_weights}" --output "{filename_converted_weights}"
```

Output

```
2019-12-24 20:08:13.233259: I
tensorflow/core/platform/cpu_feature_guard.cc:145] This TensorFlow
binary is optimized with Intel(R) MKL-DNN to use the following CPU
instructions in performance critical operations:  SSE4.1 SSE4.2 AVX
AVX2 FMA
To enable them in non-MKL-DNN operations, rebuild TensorFlow with the
appropriate compiler flags.
2019-12-24 20:08:13.233578: I
tensorflow/core/common_runtime/process_util.cc:115] Creating new
thread pool with default inter op setting: 8. Tune using
inter_op_parallelism_threads for best performance.
Model: "yolov3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #

...

yolo_output_2/conv2d_74 bias
I1224 20:08:21.439261 140735710913408 convert.py:22] weights loaded
I1224 20:08:22.945503 140735710913408 convert.py:26] sanity check
passed
I1224 20:08:23.474303 140735710913408 convert.py:29] weights saved
```

The conversion script is no longer needed once this script converts the YOLO weights have to a TensorFlow format. Because this executable file resides in the same directory as the course files, we delete it at this point.

Code

```
import os
os.remove(filename_convert_script)
```

Now that we have all of the files needed for YOLO, we are ready to use it to recognize components of an image.

6.5.5 Running DarkFlow (YOLO)

The YoloV3-TF2 library can easily integrate with Python applications. The initialization of the library consists of three steps. First, it is essential to import all of the needed packages for the library. Next, the Python program must define all of the YOLO configurations through the Keras flags architecture. The Keras flag system primarily works from the command line; however, it also allows configuration programmatically in an application. For this example, we configure the package programmatically. Finally, we must scan available devices so that our application takes advantage of any GPUs. The following code performs all three of these steps.

Code

```
import time
from absl import app, flags, logging
from absl.flags import FLAGS
import cv2
import numpy as np
import tensorflow as tf
from yolov3_tf2.models import (YoloV3, YoloV3Tiny)
from yolov3_tf2.dataset import transform_images, load_tfrecord_dataset
from yolov3_tf2.utils import draw_outputs
import sys
from PIL import Image, ImageFile
import requests

# Flags are used to define several options for YOLO.
flags.DEFINE_string('classes', filename_classes, 'path to classes file')
flags.DEFINE_string('weights', filename_converted_weights, \
                    'path to weights file')
flags.DEFINE_boolean('tiny', False, 'yolov3 or yolov3-tiny')
flags.DEFINE_integer('size', 416, 'resize images to')
flags.DEFINE_string('tfrecord', None, 'tfrecord instead of image')
flags.DEFINE_integer('num_classes', 80, 'number of classes in the model')
FLAGS([sys.argv[0]])

# Locate devices to run YOLO on (e.g. GPU)
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) > 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

It is important to understand that Keras flags can only be defined once. If you are going to classify more than one image, make sure that you do not define the flags additional times.

The following code initializes a YoloV3-TF2 classification object. The weights are loaded, and the object is ready for use as the **yolo** variable. It is not necessary to reload the weights and obtain a new **yolo** variable for each classification.

Code

```
# This example does not use the "Tiny version"
if FLAGS.tiny:
    yolo = YoloV3Tiny(classes=FLAGS.num_classes)
else:
    yolo = YoloV3(classes=FLAGS.num_classes)

# Load weights and classes
yolo.load_weights(FLAGS.weights).expect_partial()
print('weights loaded')

class_names = [c.strip() for c in open(FLAGS.classes).readlines()]
print('classes loaded')
```

Output

```
weights loaded
classes loaded
```

Next, we obtain an image to classify. For this example, the program loads the image from a URL. YoloV3-TF2 expects that the image is in the format of a Numpy array. An image file, such as JPEG or PNG, is converted into this raw Numpy format by calling the TensorFlow **decode_image** function. YoloV3-TF2 can obtain images from other sources, so long as the program first decodes them to raw Numpy format. The following code obtains the image in this format.

Code

```
# Read image to classify
url = "https://raw.githubusercontent.com/jeffheaton/"\
    "t81_558_deep_learning/master/images/cook.jpg"
response = requests.get(url)
img_raw = tf.image.decode_image(response.content, channels=3)
```

At this point, we can classify the image that was just loaded. The program should preprocess the image so that it is the size expected by YoloV3-TF2. Your program also sets the confidence threshold at this point. Any sub-image recognized with confidence below this value is not returned by YOLO.

Code

```
# Preprocess image
img = tf.expand_dims(img_raw, 0)
img = transform_images(img, FLAGS.size)

# Desired threshold (any sub-image below this confidence
# level will be ignored.)
FLAGS.yolo_score_threshold = 0.5

# Recognize and report results
```

```
t1 = time.time()
boxes, scores, classes, nums = yolo(img)
t2 = time.time()
print(f"Prediction time: {hms_string(t2 - t1)}")
```

Output

```
Prediction time: 0:00:02.18
```

It is important to note that the **yolo** class instantiated here is a callable object, which means that it can fill the role of both an object and a function. Acting as a function, *yolo* returns three arrays named **boxes**, **scores**, and **classes** that are of the same length. The function returns all sub-images found with a score above the minimum threshold. Additionally, the **yolo** function returns an array named called **nums**. The first element of the **nums** array specifies how many sub-images YOLO found to be above the score threshold.

- **boxes** - The bounding boxes for each of the sub-images detected in the image sent to YOLO.
- **scores** - The confidence for each of the sub-images detected.
- **classes** - The string class names for each of the items. These are COCO names such as "person" or "dog."
- **nums** - The number of images above the threshold.

Your program should use these values to perform whatever actions you wish as a result of the input image. The following code simply displays the images detected above the threshold.

Code

```
print('detections:')
for i in range(nums[0]):
    cls = class_names[int(classes[0][i])]
    score = np.array(scores[0][i])
    box = np.array(boxes[0][i])
    print(f"\t{cls}, {score}, {box}")
```

Output

```
detections:
        person, 0.9995919466018677, [0.31659657 0.10725152 0.68426734
0.74258995]
        dog, 0.9896982312202454, [0.51111007 0.55769485 0.933974
0.8187981 ]
        microwave, 0.9839581847190857, [0.0069517  0.08101548
0.27909756 0.28820014]
        oven, 0.9383126497268677, [0.00773552 0.32521245 0.42321444
0.83368266]
        bottle, 0.8538918495178223, [0.73093545 0.23399045 0.76463544
0.32874537]
        bottle, 0.5538208484649658, [0.790116   0.26327905 0.8189085
```

```
0.3274593 ]
```

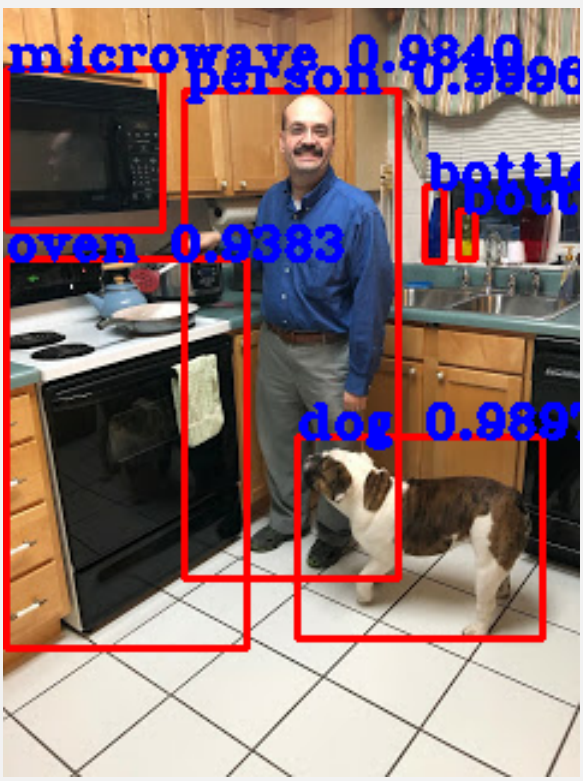
Your program should use these values to perform whatever actions you wish as a result of the input image. The following code simply displays the images detected above the threshold.

YoloV3-TF2 includes a function named **draw_outputs** that allows the sub-image detections to visualized. The following image shows the output of the draw_outputs function. You might have first seen YOLO demonstrated as an image with boxes and labels around the sub-images. A program can produce this output with the arrays returned by the **yolo** function.

Code

```
# Display image using YOLO library's built in function
img = img_raw.numpy()
img = draw_outputs(img, (boxes, scores, classes, nums), class_names)
#cv2.imwrite(FLAGS.output, img) # Save the image
display(Image.fromarray(img, 'RGB')) # Display the image
```

Output



6.5.6 Module 6 Assignment

You can find the first assignment here: assignment 6

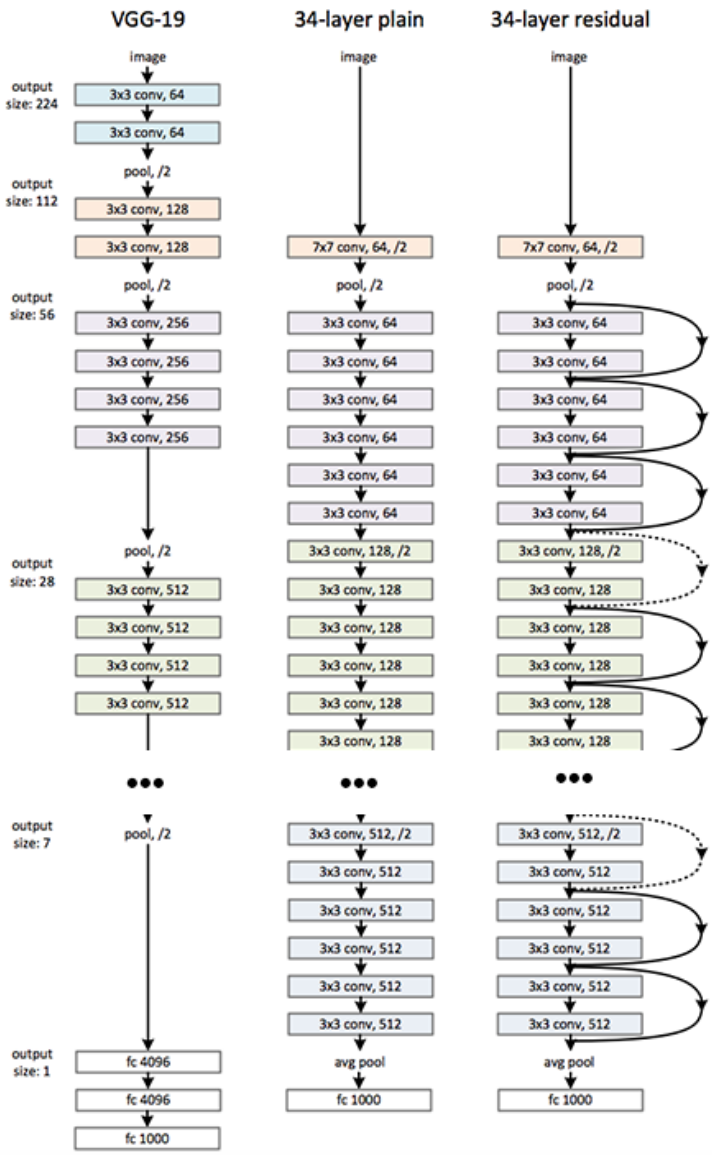


Figure 6.9: ResNets

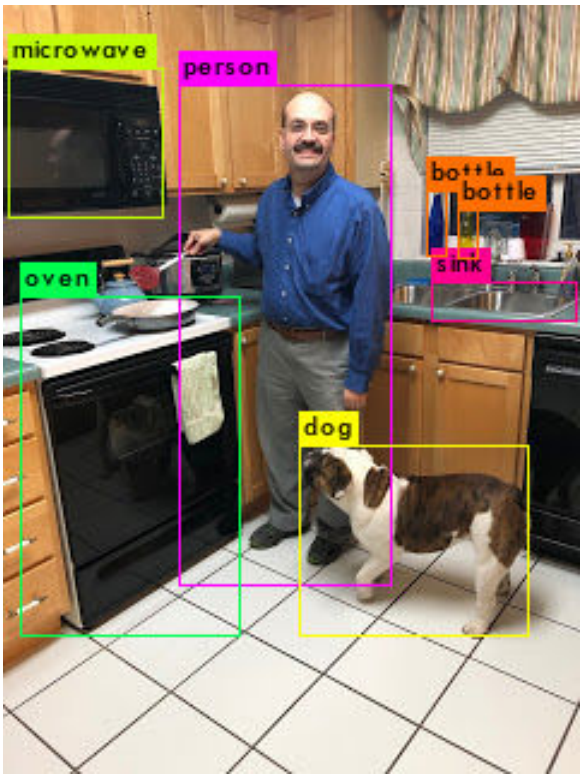


Figure 6.10: YOLO Tagging

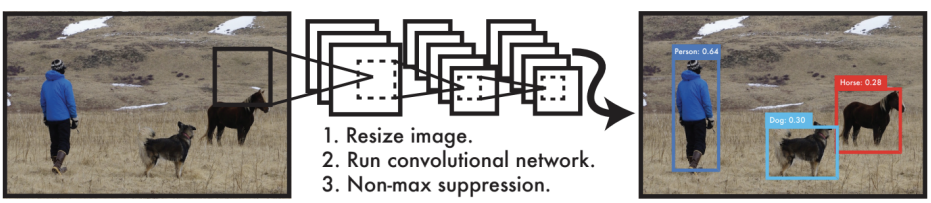


Figure 6.11: The YOLO Detection System

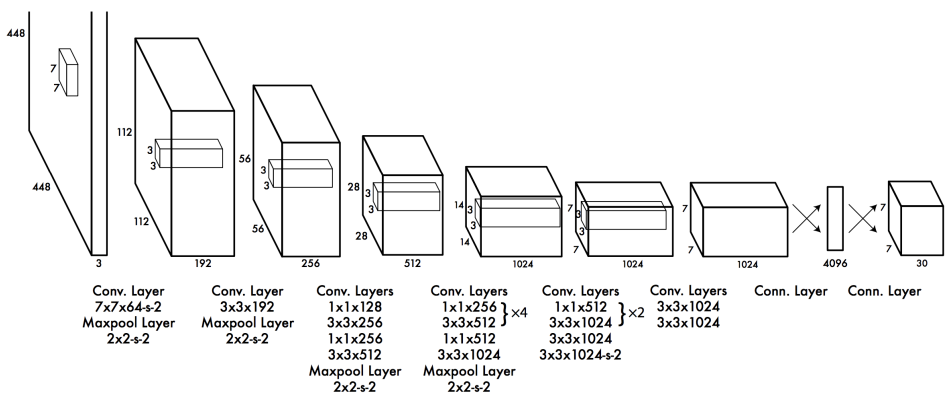


Figure 6.12: YOLO Structure

Chapter 7

Generative Adversarial Networks

7.1 Part 7.1: Introduction to GANS for Image and Data Generation

A generative adversarial network (GAN) is a class of machine learning systems invented by Ian Goodfellow in 2014.[12]Two neural networks contest with each other in a game. Given a training set, this technique learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers, having many realistic characteristics. Though originally proposed as a form of generative model for unsupervised learning, GANs have also proven useful for semi-supervised learning, fully supervised learning, and reinforcement learning.

This paper used neural networks to automatically generate images for several datasets that we've seen previously: MINST and CIFAR. However, it also included the Toronto Face Dataset (a private dataset used by some researchers). These generated images are given in Figure 7.1.

Only sub-figure D made use of convolutional neural networks. Figures A-C make use of fully connected neural networks. As we will see in this module, the role of convolutional neural networks with GANs was greatly increased.

A GAN is called a generative model because it generates new data. The overall process of a GAN is given by the following diagram in Figure 7.2.

7.2 Part 7.2: Implementing DCGANs in Keras

Paper that described the type of DCGAN that we will create in this module.[34]This paper implements a DCGAN as follows:

- No pre-processing was applied to training images besides scaling to the range of the tanh activation function [-1, 1].
- All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128.
- All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02.
- In the LeakyReLU, the slope of the leak was set to 0.2 in all models.
- we used the Adam optimizer(Kingma & Ba, 2014) with tuned hyperparameters. We found the suggested learning rate of 0.001, to be too high, using 0.0002 instead.
- Additionally, we found leaving the momentum term $\beta 1$ at the suggested value of 0.9 resulted in training oscillation and instability while reducing it to 0.5 helped stabilize training.

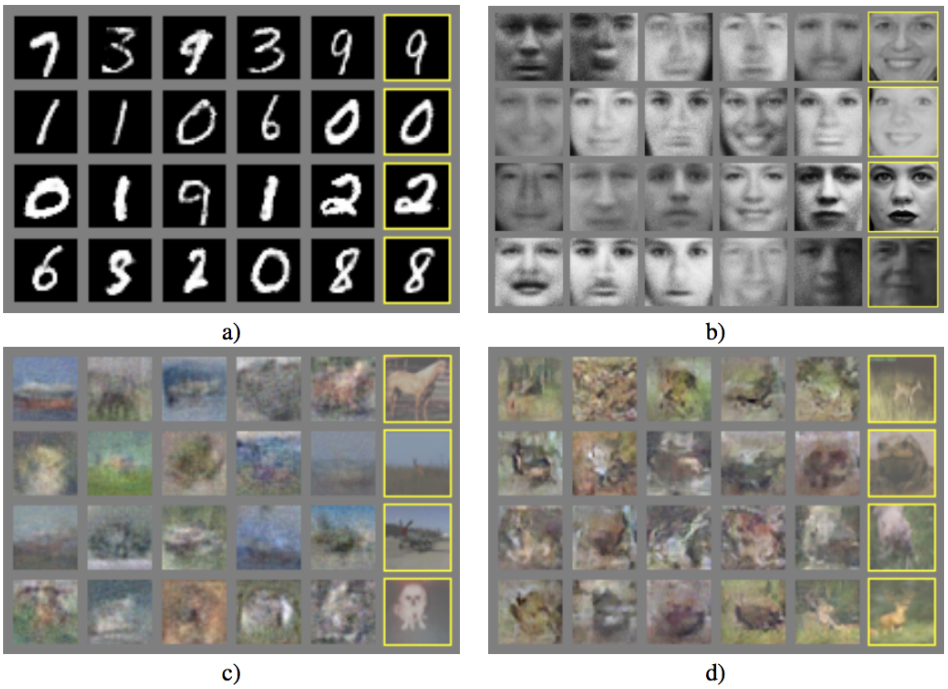


Figure 7.1: GAN Generated Images

The paper also provides the following architecture guidelines for stable Deep Convolutional GANs:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

While creating the material for this module I used a number of Internet resources, some of the most helpful were:

- Deep Convolutional Generative Adversarial Network (TensorFlow 2.0 example code)
- Keep Calm and train a GAN. Pitfalls and Tips on training Generative Adversarial Networks
- Collection of Keras implementations of Generative Adversarial Networks GANs
- dcgan-facegenerator, Semi-Paywalled Article by GitHub Author

The program created next will generate faces similar to these. While these faces are not perfect, they demonstrate how we can construct and train a GAN on or own. Later we will see how to import very advanced weights from nVidia to produce high resolution, realistic looking faces. Figure 7.3 shows images from GAN training.

As discussed in the previous module, the GAN is made up of two different neural networks: the discriminator and the generator. The generator generates the images, while the discriminator detects if a face is real or was generated. These two neural networks work as shown in Figure 7.4:

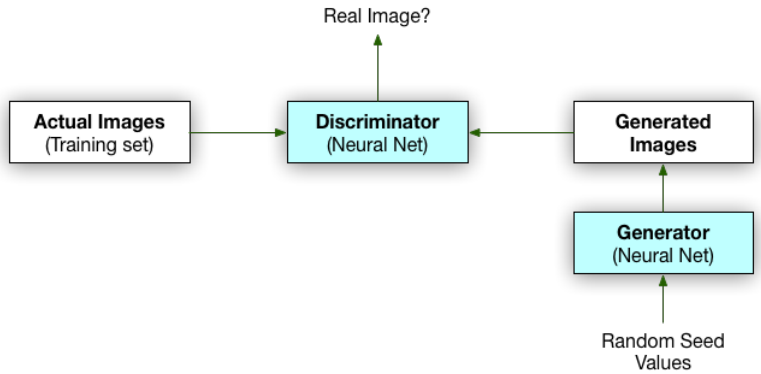


Figure 7.2: GAN Structure

The discriminator accepts an image as its input and produces number that is the probability of the input image being real. The generator accepts a random seed vector and generates an image from that random vector seed. An unlimited number of new images can be created by providing additional seeds.

I suggest running this code with a GPU, it will be very slow on a CPU alone. The following code mounts your Google drive for use with Google CoLab. If you are not using CoLab, the following code will not work.

Code

```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Output

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth
?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleus
ercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&respons
e_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdo
cs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2
f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2f
www.googleapis.com%2fauth%2fpeopleapi.readonly
Enter your authorization code:
Mounted at /content/drive
Note: using Google CoLab
TensorFlow 2.x selected.
```

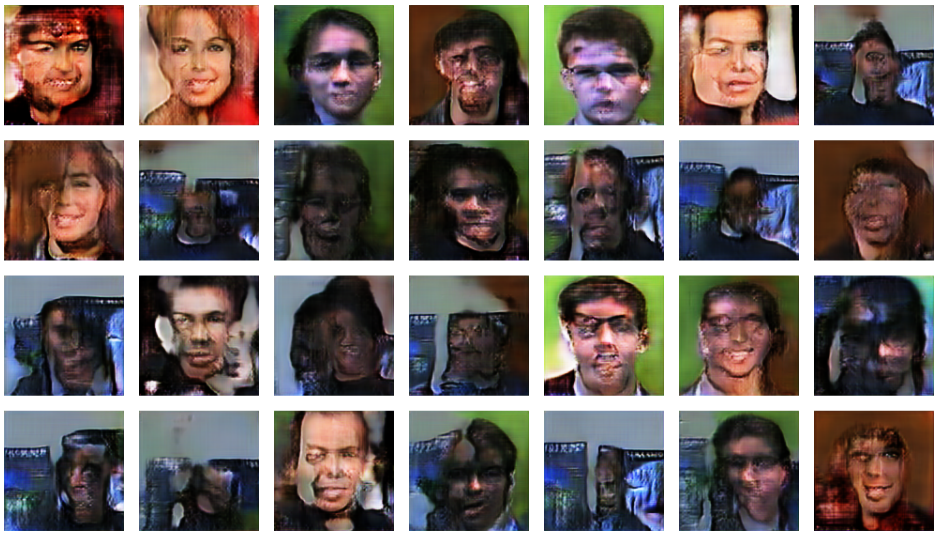


Figure 7.3: GAN Neural Network Training

The following packages will be used to implement a basic GAN system in Python/Keras.

Code

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Reshape, Dropout, Dense
from tensorflow.keras.layers import Flatten, BatchNormalization
from tensorflow.keras.layers import Activation, ZeroPadding2D
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.layers import UpSampling2D, Conv2D
from tensorflow.keras.models import Sequential, Model, load_model
from tensorflow.keras.optimizers import Adam
import numpy as np
from PIL import Image
from tqdm import tqdm
import os
import time
import matplotlib.pyplot as plt
```

These are the constants that define how the GANs will be created for this example. The higher the resolution, the more memory that will be needed. Higher resolution will also result in longer run times. For Google CoLab (with GPU) 128x128 resolution is as high as can be used (due to memory). Note that the resolution is specified as a multiple of 32. So **GENERATE_RES** of 1 is 32, 2 is 64, etc.

To run this you will need training data. The training data can be any collection of images. I suggest using training data from the following two locations. Simply unzip and combine to a common directory. This directory should be uploaded to Google Drive (if you are using CoLab). The constant **DATA_PATH** defines where these images are stored.

The source data (faces) used in this module can be found here:

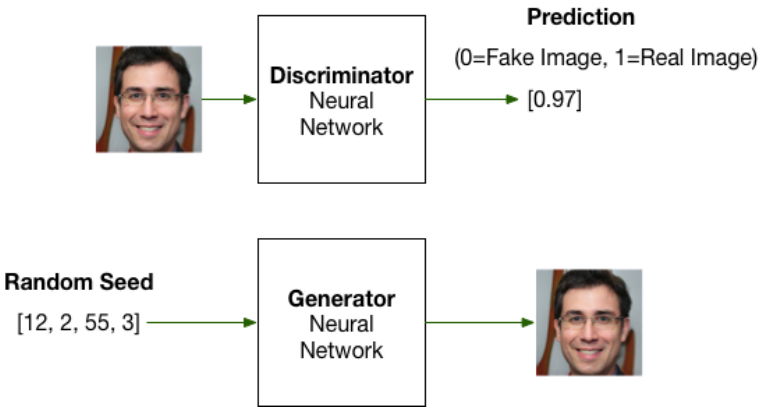


Figure 7.4: Evaluating GANs

- Kaggle Faces Data New
- Kaggle Lag Dataset: Dataset of faces, from more than 1k different subjects

Code

```
# Generation resolution - Must be square
# Training data is also scaled to this.
# Note GENERATE_RES 4 or higher
# will blow Google CoLab's memory and have not
# been tested extensivly.
GENERATE_RES = 3 # Generation resolution factor
# (1=32, 2=64, 3=96, 4=128, etc.)
GENERATE_SQUARE = 32 * GENERATE_RES # rows/cols (should be square)
IMAGE_CHANNELS = 3

# Preview image
PREVIEW_ROWS = 4
PREVIEW_COLS = 7
PREVIEW_MARGIN = 16

# Size vector to generate images from
SEED_SIZE = 100

# Configuration
DATA_PATH = '/content/drive/My Drive/projects/faces'
EPOCHS = 50
BATCH_SIZE = 32
BUFFER_SIZE = 60000

print(f"Will generate {GENERATE_SQUARE}px square images.")
```

Output

```
Will generate 96px square images.
```

Next we will load and preprocess the images. This can take awhile. Google CoLab took around an hour to process. Because of this we store the processed file as a binary. This way we can simply reload the processed training data and quickly use it. It is most efficient to only perform this operation once. The dimensions of the image are encoded into the filename of the binary file because we need to regenerate it if these change.

Code

```
# Image set has 11,682 images.  Can take over an hour
# for initial preprocessing.
# Because of this time needed, save a Numpy preprocessed file.
# Note, that file is large enough to cause problems for
# sume verisons of Pickle,
# so Numpy binary files are used.
training_binary_path = os.path.join(DATA_PATH,
        f'training_data_{GENERATE_SQUARE}_{GENERATE_SQUARE}.npy')

print(f"Looking for file: {training_binary_path}")

if not os.path.isfile(training_binary_path):
  start = time.time()
  print("Loading training images...")

  training_data = []
  faces_path = os.path.join(DATA_PATH, 'face_images')
  for filename in tqdm(os.listdir(faces_path)):
      path = os.path.join(faces_path,filename)
      image = Image.open(path).resize((GENERATE_SQUARE,
            GENERATE_SQUARE),Image.ANTIALIAS)
      training_data.append(np.asarray(image))
  training_data = np.reshape(training_data,(-1,GENERATE_SQUARE,
            GENERATE_SQUARE,IMAGE_CHANNELS))
  training_data = training_data.astype(np.float32)
  training_data = training_data / 127.5 - 1.


  print("Saving training image binary...")
  np.save(training_binary_path,training_data)
  elapsed = time.time()-start
  print (f'Image preprocess time: {hms_string(elapsed)}')
else:
  print("Loading previous training pickle...")
  training_data = np.load(training_binary_path)
```

Output

```
Looking for file: /content/drive/My
```

```
Drive/projects/faces/training_data_96_96.npy
Loading previous training pickle...
```

We will use a TensorFlow **Dataset** object to actually hold the images. This allows the data to be quickly shuffled int divided into the appropriate batch sizes for training.

Code

```
# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(training_data) \
    .shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

The code below creates the generator and discriminator.

Next we actually build the discriminator and the generator. Both will be trained with the Adam optimizer.

Code

```
def build_generator(seed_size, channels):
    model = Sequential()

    model.add(Dense(4*4*256,activation="relu",input_dim=seed_size))
    model.add(Reshape((4,4,256)))

    model.add(UpSampling2D())
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    model.add(UpSampling2D())
    model.add(Conv2D(256,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    # Output resolution, additional upsampling
    model.add(UpSampling2D())
    model.add(Conv2D(128,kernel_size=3,padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))

    if GENERATE_RES>1:
      model.add(UpSampling2D(size=(GENERATE_RES,GENERATE_RES)))
      model.add(Conv2D(128,kernel_size=3,padding="same"))
      model.add(BatchNormalization(momentum=0.8))
      model.add(Activation("relu"))

    # Final CNN layer
    model.add(Conv2D(channels,kernel_size=3,padding="same"))
    model.add(Activation("tanh"))
```

```
    return model


def build_discriminator(image_shape):
    model = Sequential()

    model.add(Conv2D(32, kernel_size=3, strides=2, input_shape=image_shape,
                     padding="same"))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
    model.add(ZeroPadding2D(padding=((0,1),(0,1))))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Conv2D(512, kernel_size=3, strides=1, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))

    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    return model
```

As we progress through training images will be produced to show the progress. These images will contain a number of rendered faces that show how good the generator has become. These faces will be

Code

```
def save_images(cnt,noise):
  image_array = np.full((
      PREVIEW_MARGIN + (PREVIEW_ROWS * (GENERATE_SQUARE+PREVIEW_MARGIN)),
      PREVIEW_MARGIN + (PREVIEW_COLS * (GENERATE_SQUARE+PREVIEW_MARGIN)), 3),
      255, dtype=np.uint8)

  generated_images = generator.predict(noise)
```

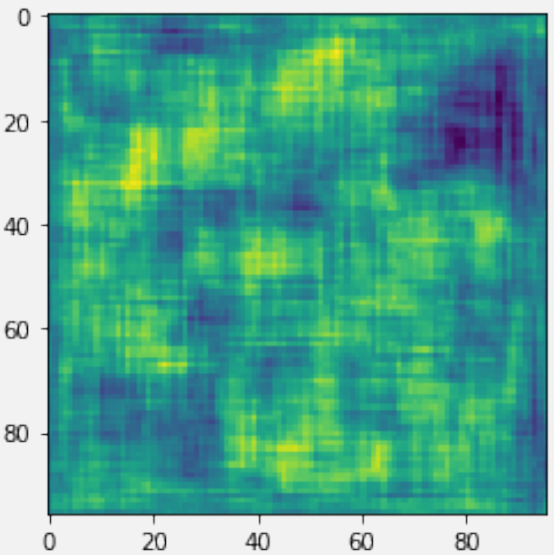
```
  generated_images = 0.5 * generated_images + 0.5

  image_count = 0
  for row in range(PREVIEW_ROWS):
      for col in range(PREVIEW_COLS):
        r = row * (GENERATE_SQUARE+16) + PREVIEW_MARGIN
        c = col * (GENERATE_SQUARE+16) + PREVIEW_MARGIN
        image_array[r:r+GENERATE_SQUARE,c:c+GENERATE_SQUARE] \
            = generated_images[image_count] * 255
        image_count += 1


  output_path = os.path.join(DATA_PATH, 'output')
  if not os.path.exists(output_path):
    os.makedirs(output_path)

  filename = os.path.join(output_path,f"train-{cnt}.png")
  im = Image.fromarray(image_array)
  im.save(filename)
```

Code

```
generator = build_generator(SEED_SIZE, IMAGE_CHANNELS)

noise = tf.random.normal([1, SEED_SIZE])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0])
```

Output

```
<matplotlib.image.AxesImage at 0x7fd92007ad68>
```



Code

```
image_shape = (GENERATE_SQUARE,GENERATE_SQUARE,IMAGE_CHANNELS)

discriminator = build_discriminator(image_shape)
decision = discriminator(generated_image)
print (decision)
```

Output

```
tf.Tensor([[0.50032634]], shape=(1, 1), dtype=float32)
```

Loss functions must be developed that allow the generator and discriminator to be trained in an adversarial way. Because these two neural networks are being trained independently they must be trained in two separate passes. This requires two separate loss functions and also two separate updates to the gradients. When the discriminator's gradients are applied to decrease the discriminator's loss it is important that only the discriminator's weights are update. It is not fair, nor will it produce good results, to adversarially damage the weights of the generator to help the discriminator. A simple backpropagation would do this. It would simultaneously affect the weights of both generator and discriminator to lower whatever loss it was assigned to lower.

Figure 7.5 shows how the discriminator is trained.

Here a training set is generated with an equal number of real and fake images. The real images are randomly sampled (chosen) from the training data. An equal number of random images are generated from random seeds. For the discriminator training set, the x contains the input images and the y contains a value of 1 for real images and 0 for generated ones.

Likewise, the Figure 7.6 shows how the generator is trained.

For the generator training set, the x contains the random seeds to generate images and the y always contains the value of 1, because the optimal is for the generator to have generated such good images that the discriminiator was fooled into assigning them a probability near 1.

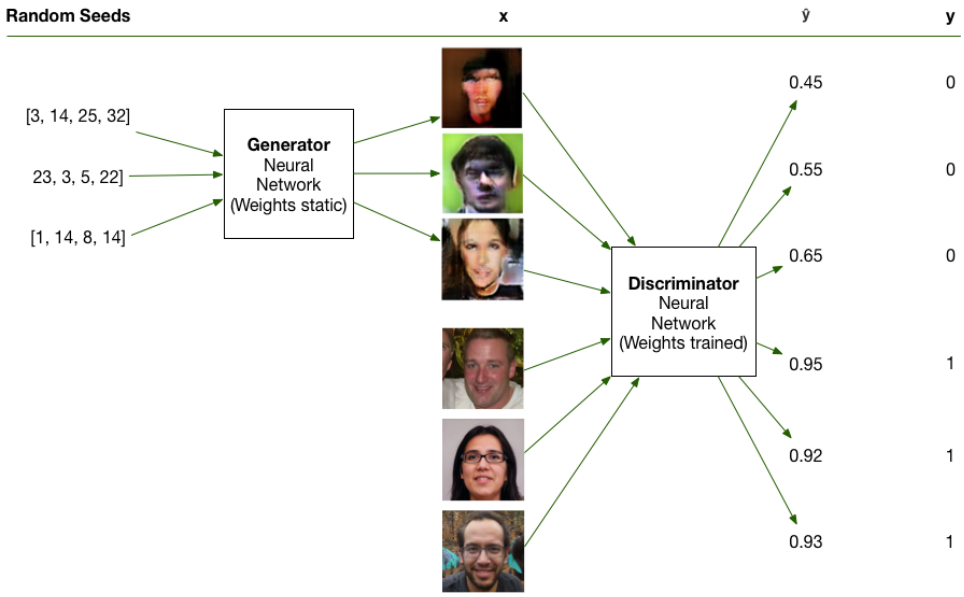


Figure 7.5: Training the Discriminator

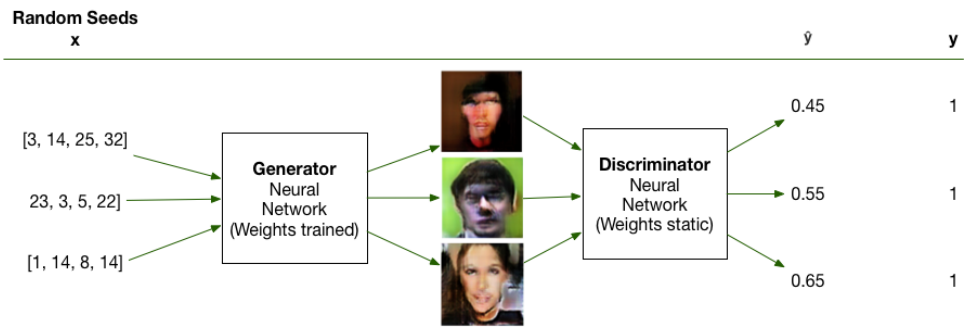


Figure 7.6: Training the Generator

Code

```
# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Both the generator and discriminator use Adam and the same learning rate and momentum. This does not need to be the case. If you use a **GENERATE_RES** greater than 3 you may need to tune these learning rates, as well as other training and hyperparameters.

Code

```
generator_optimizer = tf.keras.optimizers.Adam(1.5e-4,0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1.5e-4,0.5)
```

The following function is where most of the training takes place for both the discriminator and the generator. This function was based on the GAN provided by the TensorFlow Keras exmples documentation. The first thing you should notice about this function is that it is annotated with the **tf.function** annotation. This causes the function to be precompiled and improves performance.

This function trans differently than the code we previously saw for training. This code makes use of **GradientTape** to allow the discriminator and generator to be trained together, yet separately.

Code

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
  seed = tf.random.normal([BATCH_SIZE, SEED_SIZE])

  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    generated_images = generator(seed, training=True)

    real_output = discriminator(images, training=True)
    fake_output = discriminator(generated_images, training=True)

    gen_loss = generator_loss(fake_output)
    disc_loss = discriminator_loss(real_output, fake_output)


    gradients_of_generator = gen_tape.gradient(\
        gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(\
        disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(
        gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(
        gradients_of_discriminator,
        discriminator.trainable_variables))
  return gen_loss,disc_loss
```

Code

```
def train(dataset, epochs):
  fixed_seed = np.random.normal(0, 1, (PREVIEW_ROWS * PREVIEW_COLS,
```

```
                                                SEED_SIZE))
  start = time.time()

  for epoch in range(epochs):
    epoch_start = time.time()

    gen_loss_list = []
    disc_loss_list = []

    for image_batch in dataset:
      t = train_step(image_batch)
      gen_loss_list.append(t[0])
      disc_loss_list.append(t[1])

    g_loss = sum(gen_loss_list) / len(gen_loss_list)
    d_loss = sum(disc_loss_list) / len(disc_loss_list)

    epoch_elapsed = time.time()-epoch_start
    print (f'Epoch {epoch+1}, gen loss={g_loss},disc loss={d_loss},'\
           ' {hms_string(epoch_elapsed)}')
    save_images(epoch,fixed_seed)

  elapsed = time.time()-start
  print (f'Training time: {hms_string(elapsed)}')
```

Code

```
train(train_dataset, EPOCHS)
```

Output

```
Epoch 1, gen loss=0.6737478375434875,disc loss=1.2876468896865845,
0:01:33.86
Epoch 2, gen loss=0.6754337549209595,disc loss=1.2804691791534424,
0:01:26.96
Epoch 3, gen loss=0.6792119741439819,disc loss=1.2002451419830322,
0:01:27.23
Epoch 4, gen loss=0.6704637408256531,disc loss=1.1011184453964233,
0:01:26.74
Epoch 5, gen loss=0.6726831197738647,disc loss=1.0789912939071655,
0:01:26.87
Epoch 6, gen loss=0.6760282516479492,disc loss=1.086405873298645,
0:01:26.64
Epoch 7, gen loss=0.6668657660484314,disc loss=1.0990564823150635,
0:01:26.75
Epoch 8, gen loss=0.6794514060020447,disc loss=1.0572694540023804,
```

```
...

Epoch 49, gen loss=0.6844978332519531,disc loss=1.034816861152649,
0:01:26.71
Epoch 50, gen loss=0.6809002757072449,disc loss=1.0432080030441284,
0:01:26.67
Training time: 1:12:52.22
```

Code

```
generator.save(os.path.join(DATA_PATH,"face_generator.h5"))
```

7.3 Part 7.3: Face Generation with StyleGAN and Python

GANs have appeared frequently in the media, showcasing their ability to generate extremely photorealistic faces. One significant step forward for realistic face generation was nVidia StyleGAN.[21]In this module we will explore StyleGAN2, which is the second interation of this technology by nVidia.[22]We will also preload weights that nVidia trained on. This will allow us to generate high resolution photorealistic looking faces, such seen in Figure 7.7.

Figure 7.7: StyleGAN2 Generated Faces

The above images were generated with StyleGAN2, using Google CoLab. Following the instructions in this section, you will be able to create faces like this of your own. StyleGAN2 images are usually 1,024 x 1,024 in resolution. An example of a full resolution StyleGAN image can be found here.

While the above images look much more realistic than images generated earlier in this course, they are not perfect. Look at Figure 7.8. There are usually a number of tell-tail signs that you are looking at a computer generated image. One of the most obvious is usually the surreal, dream-like backgrounds. The background does not look obviously fake, at first glance; however, upon closer inspection you usually can't quite discern exactly what a GAN generated background actually is. Also look at the image character's left eye. It is slightly unrealistic looking, especially near the eyelashes.

Look at the following GAN face. Can you spot any imperfections?

- Image A demonstrates the very abstract backgrounds usually associated with a GAN generated image.
- Image B exhibits issues that earrings often present for GANs. GANs sometimes have problems with symmetry, particularly earrings.
- Image C contains an abstract background, as well as a highly distorted secondary image.
- Image D also contains a highly distorted secondary image that might be a hand.

There are a number of websites that allow you to generate GANs of your own without any software.

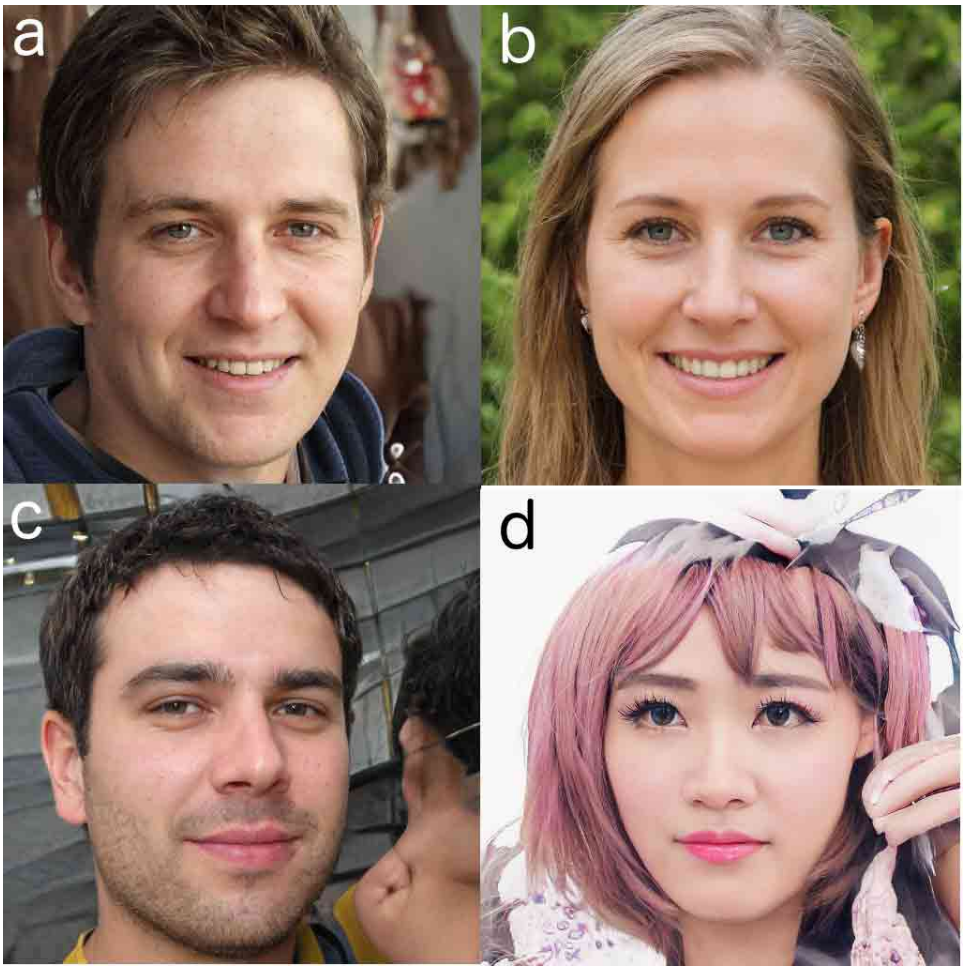


Figure 7.8: StyleGAN2 Face

- This Person Does not Exist
- Which Face is Real

The first site generates high resolution images of human faces. The second site presents a quiz to see if you can detect the difference between a real and fake human faceimage.

In this module you will learn to create your own StyleGAN2 pictures using Python.

7.3.1 Keras Sequence vs Functional Model API

Most of the neural networks create in this course have made use of the Keras sequence object. You might have noticed that we briefly made use of another type of neural network object for the ResNet, the Model. These are the two major means of constructing a neural network in Keras:

- Sequential - Simplified interface to Keras that supports most models where the flow of information is a simple sequence from input to output.
- Keras Functional API - More complex interface that allows neural networks to be constructed of reused layers, multiple input layers, and supports building your own recurrent connections.

It is important to point out that these are not two specific types of neural network. Rather, they are two means of constructing neural networks in Keras. Some types of neural network can be implemented in either, such as dense feedforward neural networks (like we used for the Iris and MPG datasets). However, other types of neural network, like ResNet and GANs can only be used in the Functional Model API.

7.3.2 Generating High Rez GAN Faces with Google CoLab

This notebook demonstrates how to run NVidia StyleGAN2 inside of a Google CoLab notebook. I suggest you use this to generate GAN faces from a pretrained model. If you try to train your own, you will run into compute limitations of Google CoLab.

Make sure to run this code on a GPU instance. GPU is assumed.

First, map your G-Drive, this is where your GANs will be written to.

Code

```
# Run this for Google CoLab (use TensorFlow 1.x)
%tensorflow_version 1.x
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

Output

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth
?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleus
ercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&respons
e_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdo
cs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2
f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2f
www.googleapis.com%2fauth%2fpeopleapi.readonly
Enter your authorization code:
Mounted at /content/drive
```

Next, clone StyleGAN2 from GitHub.

Code

```
!git clone https://github.com/NVlabs/stylegan2.git
```

Output

```
Cloning into 'stylegan2'...
remote: Enumerating objects: 88, done.
remote: Total 88 (delta 0), reused 0 (delta 0), pack-reused 88
Unpacking objects: 100% (88/88), done.
```

Verify that StyleGAN has been cloned.

Code

```
!ls /content/stylegan2/
```

Output

```
dataset_tool.py  LICENSE.txt              README.md
run_training.py
dnnlib           metrics                  run_generator.py
test_nvcc.cu
Dockerfile       pretrained_networks.py   run_metrics.py    training
docs             projector.py             run_projector.py
```

7.3.3 Run StyleGan2 From Command Line

Add the StyleGAN folder to Python so that you can import it. The code below is based on code from NVidia. This actually generates your images.

Code

```
!python /content/stylegan2/run_generator.py generate-images \
    --network=gdrive:networks/stylegan2-ffhq-config-f.pkl \
  --seeds=6600-6625 --truncation-psi=0.5
```

Output

```
Local submit - run_dir: results/00000-generate-images
dnnlib: Running run_generator.generate_images() on localhost...
Loading networks from "gdrive:networks/stylegan2-ffhq-config-f.pkl"...
Downloading http://d36zk2xti64re0.cloudfront.net/stylegan2/networks/st
ylegan2-ffhq-config-f.pkl ... done
Setting up TensorFlow plugin "fused_bias_act.cu": Preprocessing...
Compiling... Loading... Done.
Setting up TensorFlow plugin "upfirdn_2d.cu": Preprocessing...
Compiling... Loading... Done.
Generating image for seed 6600 (0/26) ...
Generating image for seed 6601 (1/26) ...
Generating image for seed 6602 (2/26) ...
Generating image for seed 6603 (3/26) ...
Generating image for seed 6604 (4/26) ...
Generating image for seed 6605 (5/26) ...

...

Generating image for seed 6622 (22/26) ...
Generating image for seed 6623 (23/26) ...
Generating image for seed 6624 (24/26) ...
```

```
Generating image for seed 6625 (25/26) ...
dnnlib: Finished run_generator.generate_images() in 1m 28s.
```

Code

```
!ls /content/results/00000-generate-images
```

Output

```
_finished.txt   seed6604.png  seed6611.png  seed6618.png  seed6625.png
log.txt         seed6605.png  seed6612.png  seed6619.png
submit_config.pkl
run.txt         seed6606.png  seed6613.png  seed6620.png
submit_config.txt
seed6600.png    seed6607.png  seed6614.png  seed6621.png
seed6601.png    seed6608.png  seed6615.png  seed6622.png
seed6602.png    seed6609.png  seed6616.png  seed6623.png
seed6603.png    seed6610.png  seed6617.png  seed6624.png
```

Code

```
cp /content/results/00000-generate-images/* \
    /content/drive/My\ Drive/projects/stylegan2
```

Code

```
!python /content/stylegan2/run_generator.py style-mixing-example \
    --network=gdrive:networks/stylegan2-ffhq-config-f.pkl \
  --row-seeds=85,100,75,458,1500 --col-seeds=55,821,1789,293 \
    --truncation-psi=1.0
```

Output

```
Local submit - run_dir: results/00001-style-mixing-example
dnnlib: Running run_generator.style_mixing_example() on localhost...
Loading networks from "gdrive:networks/stylegan2-ffhq-config-f.pkl"...
Setting up TensorFlow plugin "fused_bias_act.cu": Preprocessing...
Loading... Done.
Setting up TensorFlow plugin "upfirdn_2d.cu": Preprocessing...
Loading... Done.
Generating W vectors...
Generating images...
Generating style-mixed images...
Saving images...
Saving image grid...
dnnlib: Finished run_generator.style_mixing_example() in 46s.
```

7.3.4 Run StyleGAN2 From Python Code

Add the StyleGAN folder to Python so that you can import it. The code below is based on code from NVidia. This actually generates your images.

Code

```
import sys
sys.path.insert(0, "/content/stylegan2")

import dnnlib
```

Code

```
# Copyright (c) 2019, NVIDIA Corporation. All rights reserved.
#
# This work is made available under the Nvidia Source Code License-NC.
# To view a copy of this license, visit
# https://nvlabs.github.io/stylegan2/license.html

import argparse
import numpy as np
import PIL.Image
import dnnlib
import dnnlib.tflib as tflib
import re
import sys

import pretrained_networks

#----------------------------------------------------------------------------

def expand_seed(seeds, vector_size):
  result = []

  for seed in seeds:
    rnd = np.random.RandomState(seed)
    result.append( rnd.randn(1, vector_size) )
  return result

def generate_images(Gs, seeds, truncation_psi):
    noise_vars = [var for name, var in \
                  Gs.components.synthesis.vars.items() \
                  if name.startswith('noise')]

    Gs_kwargs = dnnlib.EasyDict()
    Gs_kwargs.output_transform = dict(func= \
        tflib.convert_images_to_uint8, nchw_to_nhwc=True)
    Gs_kwargs.randomize_noise = False
```

```
    if truncation_psi is not None:
        Gs_kwargs.truncation_psi = truncation_psi

    for seed_idx, seed in enumerate(seeds):
        print('Generating image for seed %d/%d ...' % (seed_idx, len(seeds)))
        rnd = np.random.RandomState()
        tflib.set_vars({var: rnd.randn(*var.shape.as_list()) \
                        for var in noise_vars}) # [height, width]
        images = Gs.run(seed, None, **Gs_kwargs)
        # [minibatch, height, width, channel]
        path = f"/content/drive/My Drive/projects/"+\
                f"stylegan2/image{seed_idx}.png"
        PIL.Image.fromarray(images[0], 'RGB').save(path)

def main():
    sc = dnnlib.SubmitConfig()
    sc.num_gpus = 1
    sc.submit_target = dnnlib.SubmitTarget.LOCAL
    sc.local.do_not_copy_source_files = True
    sc.run_dir_root = "/content/drive/My Drive/projects/stylegan2"
    sc.run_desc = 'generate-images'
    network_pkl = 'gdrive:networks/stylegan2-ffhq-config-f.pkl'

    print('Loading networks from "%s"...' % network_pkl)
    _G, _D, Gs = pretrained_networks.load_networks(network_pkl)
    vector_size = Gs.input_shape[1:][0]
    seeds = expand_seed( range(8000,8020), vector_size)
    generate_images(Gs, seeds,truncation_psi=0.5)

#----------------------------------------------------------------------

if __name__ == "__main__":
    main()

#----------------------------------------------------------------------
```

Output

```
Loading networks from "gdrive:networks/stylegan2-ffhq-config-f.pkl"...
Generating image for seed 0/20 ...
Generating image for seed 1/20 ...
Generating image for seed 2/20 ...
Generating image for seed 3/20 ...
Generating image for seed 4/20 ...
Generating image for seed 5/20 ...
Generating image for seed 6/20 ...
Generating image for seed 7/20 ...
Generating image for seed 8/20 ...
```

```
Generating image for seed 9/20 ...
Generating image for seed 10/20 ...
Generating image for seed 11/20 ...
Generating image for seed 12/20 ...
Generating image for seed 13/20 ...

...

Generating image for seed 15/20 ...
Generating image for seed 16/20 ...
Generating image for seed 17/20 ...
Generating image for seed 18/20 ...
Generating image for seed 19/20 ...
```

7.3.5 Examining the Latent Vector

Figure 7.9 shows the effects of transforming the latent vector between two images.

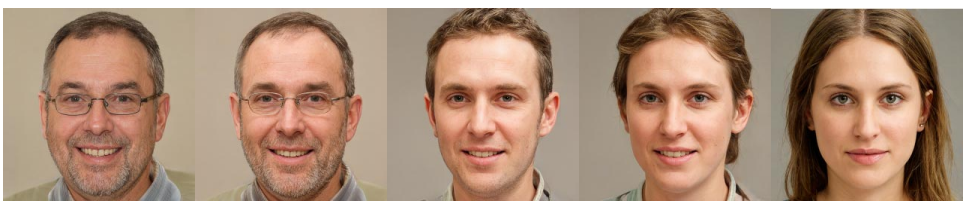


Figure 7.9: Transforming the Latent Vector

Code

```
sc = dnnlib.SubmitConfig()
sc.num_gpus = 1
sc.submit_target = dnnlib.SubmitTarget.LOCAL
sc.local.do_not_copy_source_files = True
sc.run_dir_root = "/content/drive/My Drive/projects/stylegan2"
sc.run_desc = 'generate-images'
network_pkl = 'gdrive:networks/stylegan2-ffhq-config-f.pkl'

print('Loading networks from "%s"...' % network_pkl)
_G, _D, Gs = pretrained_networks.load_networks(network_pkl)
vector_size = Gs.input_shape[1:][0]
# range(8192,8300)
seeds = expand_seed( [8192+1,8192+9], vector_size)
#generate_images(Gs, seeds,truncation_psi=0.5)
print(seeds[0].shape)
```

Output

```
Loading networks from "gdrive:networks/stylegan2-ffhq-config-f.pkl"...
(1, 512)
```

Code

```
# 8192+1,8192+9

STEPS = 300
diff = seeds[1] - seeds[0]
step = diff / STEPS
current = seeds[0].copy()

seeds2 = []
for i in range(STEPS):
  seeds2.append(current)
  current = current + step

generate_images(Gs, seeds2,truncation_psi=0.5)
```

Output

```
Generating image for seed 0/300 ...
Generating image for seed 1/300 ...
Generating image for seed 2/300 ...
Generating image for seed 3/300 ...
Generating image for seed 4/300 ...
Generating image for seed 5/300 ...
Generating image for seed 6/300 ...
Generating image for seed 7/300 ...
Generating image for seed 8/300 ...
Generating image for seed 9/300 ...
Generating image for seed 10/300 ...
Generating image for seed 11/300 ...
Generating image for seed 12/300 ...
Generating image for seed 13/300 ...
Generating image for seed 14/300 ...

...

Generating image for seed 295/300 ...
Generating image for seed 296/300 ...
Generating image for seed 297/300 ...
Generating image for seed 298/300 ...
Generating image for seed 299/300 ...
```

You might wish to view these generate images as a video file. The command line utility **ffmpeg** can be used to combine these images into a video. The following command could be used to create this video. You

will usually have to install **ffmpeg** for the particular operating system that you are running. Refer to the ffmpeg website for more details on installation.

```
ffmpeg -r 30 -i image%d.png -vcodec mpeg4 -y movie.mp4
```

7.4 Part 7.4: GANS for Semi-Supervised Training in Keras

GANs can also be used to implement semi-supervised learning/training. Normally GANs implement unsupervised training. This is because there are no y's (expected outcomes) provided in the dataset. The y-values are usually called labels. For the face generating GANs, there is typically no y-value, only images. This is unsupervised training. Supervised training occurs when we are training a model to predict specified y-values. These techniques are summarized in Figure 7.10.

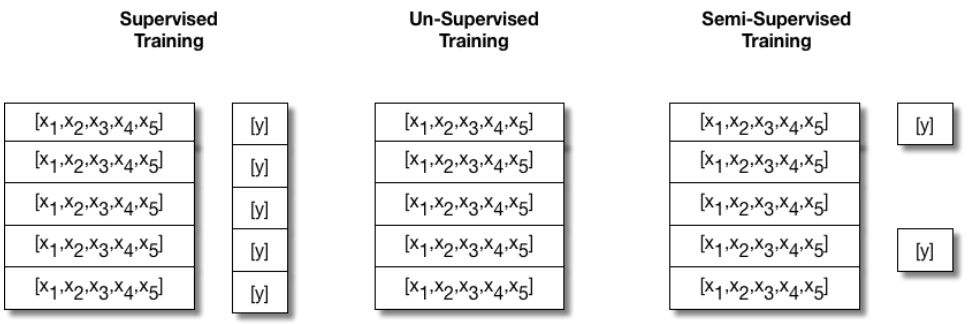


Figure 7.10: Supervised, UnSupervised and Semi-Supervised

This module will look at how to implement semi-supervised training with a GAN.[32]

As you can see, supervised learning is where all data have labels. Supervised learning attempts to learn the labels from the training data to predict these labels for new data. Un-supervised learning has no labels and usually simply clusters the data or in the case of a GAN, learns to produce new data that resembles the training data. Semi-supervised training has a small number of labels for mostly unlabeled data. Semi-supervised learning is usually similar to supervised learning in that the goal is ultimately to predict labels for new data.

Traditionally, unlabeled data would simply be discarded if the overall goal was to create a supervised model. However, the unlabeled data is not without value. Semi-supervised training attempts to use this unlabeled data to help learn additional insights about what labels we do have. There are limits, however. Even semi-supervised training cannot learn entirely new labels that were not in the training set. This would include new classes for classification or learning to predict values outside of the range of the y-values.

Semi-supervised GANs can perform either classification or regression. Previously, we made use of the generator and discarded the discriminator. We simply wanted to create new photo-realistic faces, so we just needed the generator. Semi-supervised learning flips this, as we now discard the generator and make use of the discriminator as our final model.

7.4.1 Semi-Supervised Classification Training

Figure 7.11 shows how to apply GANs for semi-supervised classification training.

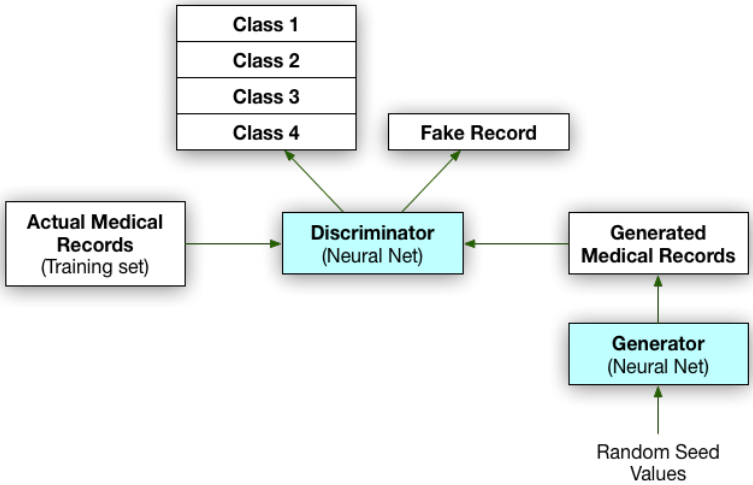


Figure 7.11: GAN for Semisupervised Training

Semi-supervised classification training is laid exactly the same as a regular GAN. The only differences is that it is not a simple true/false classifier as was the case for image GANs that simply classified if the generated image was a real or fake. The additional classes are also added. Later in this module I will provide a link to an example of The Street View House Numbers (SVHN) Dataset. This dataset contains house numbers, as seen in the following image.

Perhaps all of the digits are not labeled. The GAN is setup to classify a real or fake digit, just as we did with the faces. However, we also expand upon the real digits to include classes 0-9. The GAN discriminator is classifying between the 0-9 digits and also fake digits. A semi-supervised GAN classifier always classifies to the number of classes plus one. The additional class indicates a fake classification.

7.4.2 Semi-Supervised Regression Training

Figure 7.13 shows how to apply GANs for semi-supervised regression training.

Neural networks can perform both classification and regression simultaneously, it is simply a matter of how the output neurons are mapped. A hybrid classification-regression neural network simply maps groups of output neurons to be each of the groups of classes to be predicted, along with individual neurons to perform any regression predictions needed.

A regression semi-supervised GAN is one such hybrid. The discriminator has two output neurons. The first output neuron performs the requested regression prediction. The second predicts the probability that the input was fake.

7.4.3 Application of Semi-Supervised Regression

An example of using Keras for Semi-Supervised classification is provided here.

- Semi-supervised learning with Generative Adversarial Networks (GANs)
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- The Street View House Numbers (SVHN) Dataset

Figure 7.12: Semi-Supervised Training for the SVHN Data Set

7.5 Part 7.5: An Overview of GAN Research

- Keras Implementations of Generative Adversarial Networks
- Curated List of Awesome GAN Applications and Demo

7.5.1 Select Projects

- Few-Shot Adversarial Learning of Realistic Neural Talking Head Models, YouTube of Talking Heads
- Pose Guided Person Image Generation
- Deep Fake

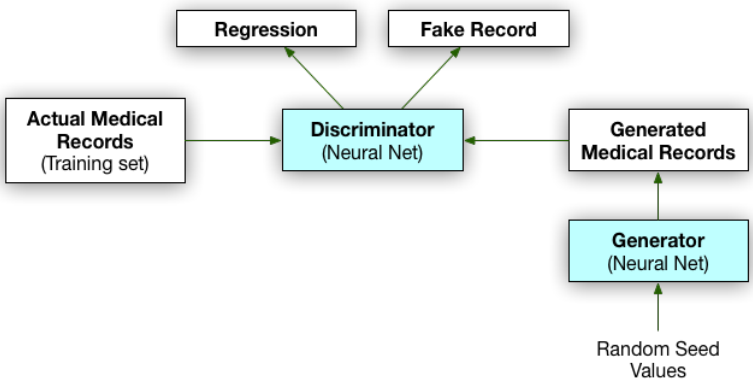


Figure 7.13: Semisupervised Structure.

Chapter 8

Kaggle Data Sets

8.1 Part 8.1: Introduction to Kaggle

Kaggle runs competitions where data scientists compete to provide the best model to fit the data. A simple project to get started with Kaggle is the Titanic data set. Most Kaggle competitions end on a specific date. Website organizers have currently scheduled the Titanic competition to end on December 31, 20xx (with the year usually rolling forward). However, they have already extended the deadline several times, and an extension beyond 2014 is also possible. Second, the Titanic data set is considered a tutorial data set. There is no prize, and your score in the competition does not count towards becoming a Kaggle Master.

8.1.1 Kaggle Ranks

You achieve Kaggle ranks by earning gold, silver, and bronze medals.

- Kaggle Top Users
- Current Top Kaggle User's Profile Page
- Jeff Heaton's (your instructor) Kaggle Profile
- Current Kaggle Ranking System

8.1.2 Typical Kaggle Competition

A typical Kaggle competition will have several components. Consider the Titanic tutorial:

- Competition Summary Page
- Data Page
- Evaluation Description Page
- Leaderboard

8.1.3 How Kaggle Competition Scoring

Kaggle is provided with a data set by the competition sponsor, as seen in Figure 8.1. Kaggle divides this data set as follows:

- **Complete Data Set** - This is the complete data set.
  - **Training Data Set** - This dataset provides both the inputs and the outcomes for the training portion of the data set.

- **Test Data Set** - This dataset provides the complete test data; however, it does not give the outcomes. Your submission file should contain the predicted results for this data set.
  - **Public Leaderboard** - Kaggle does not tell you what part of the test data set contributes to the public leaderboard. Your public score is calculated based on this part of the data set.
  - **Private Leaderboard** - Likewise, Kaggle does not tell you what part of the test data set contributes to the public leaderboard. Your final score/rank is calculated based on this part. You do not see your private leaderboard score until the end.

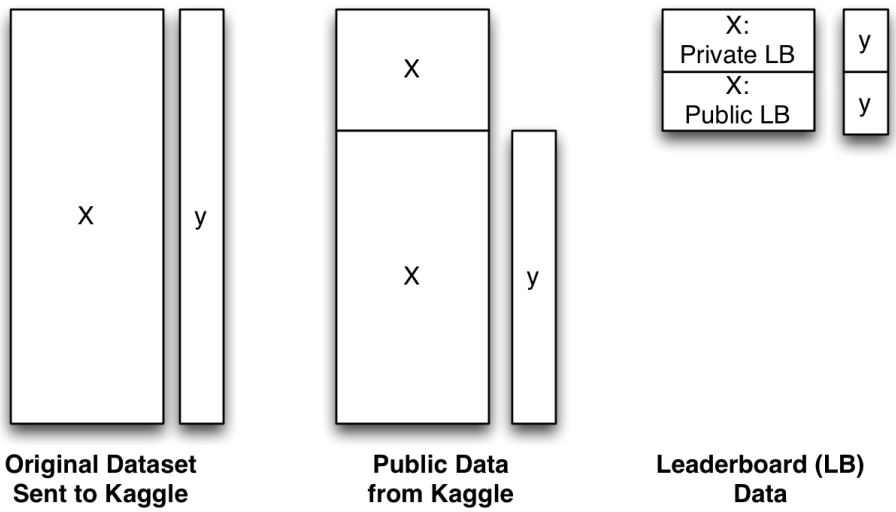


Figure 8.1: How Kaggle Competition Scoring

8.1.4 Preparing a Kaggle Submission

You do not submit the code to your solution to Kaggle. For competitions, you are scored entirely on the accuracy of your submission file. A Kaggle submission file is always a CSV file that contains the **Id** of the row you are predicting and the answer. For the titanic competition, a submission file looks something like this:

```
PassengerId,Survived
892,0
893,1
894,1
895,0
896,0
897,1
...
```

The above file states the prediction for each of the various passengers. You should only predict on ID's that are in the test file. Likewise, you should render a prediction for every row in the test file. Some competitions will have different formats for their answers. For example, a multi-classification will usually have a column for each class and your predictions for each class.

8.1.5 Select Kaggle Competitions

There have been many exciting competitions on Kaggle; these are some of my favorites. Some select predictive modeling competitions, which use tabular data include:

- Otto Group Product Classification Challenge
- Galaxy Zoo - The Galaxy Challenge
- Practice Fusion Diabetes Classification
- Predicting a Biological Response

Many Kaggle competitions include computer vision datasets, such as:

- Diabetic Retinopathy Detection
- Cats vs Dogs
- State Farm Distracted Driver Detection

8.1.6 Module 8 Assignment

You can find the first assignment here: assignment 8

8.2 Part 8.2: Building Ensembles with Scikit-Learn and Keras

8.2.1 Evaluating Feature Importance

Feature importance tells us how important each of the features (from the feature/import vector is to the prediction of a neural network or another model. There are many different ways to evaluate the feature importance for neural networks. The following paper presents an excellent (and readable) overview of the various means of assessing the significance of neural network inputs/features.

- An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data[33]. *Ecological Modelling*, 178(3), 389-397.

In summary, the following methods are available to neural networks:

- Connection Weights Algorithm
- Partial Derivatives
- Input Perturbation
- Sensitivity Analysis
- Forward Stepwise Addition
- Improved Stepwise Selection 1
- Backward Stepwise Elimination
- Improved Stepwise Selection

For this class, we will use the **Input Perturbation** feature ranking algorithm. This algorithm will work with any regression or classification network. I provide an implementation of the input perturbation algorithm for scikit-learn in the next section. This code implements a function below that will work with any scikit-learn model.

Leo Breiman provided this algorithm in his seminal paper on random forests. [Citebreiman2001random:] Although he presented this algorithm in conjunction with random forests, it is model-independent and appropriate for any supervised learning model. This algorithm, known as the input perturbation algorithm, works by evaluating a trained model's accuracy with each of the inputs individually shuffled from a data

set. Shuffling an input causes it to become useless---effectively removing it from the model. More important inputs will produce a less accurate score when they are removed by shuffling them. This process makes sense because important features will contribute to the accuracy of the model. I first presented the TensorFlow implementation of this algorithm in the following paper.

- Early stabilizing feature importance for TensorFlow deep neural networks[15]

This algorithm will use log loss to evaluate a classification problem and RMSE for regression.

Code

```
from sklearn import metrics
import scipy as sp
import numpy as np
import math
from sklearn import metrics

def perturbation_rank(model, x, y, names, regression):
    errors = []

    for i in range(x.shape[1]):
        hold = np.array(x[:, i])
        np.random.shuffle(x[:, i])

        if regression:
            pred = model.predict(x)
            error = metrics.mean_squared_error(y, pred)
        else:
            pred = model.predict_proba(x)
            error = metrics.log_loss(y, pred)

        errors.append(error)
        x[:, i] = hold

    max_error = np.max(errors)
    importance = [e/max_error for e in errors]

    data = {'name':names,'error':errors,'importance':importance}
    result = pd.DataFrame(data, columns = ['name','error','importance'])
    result.sort_values(by=['importance'], ascending=[0], inplace=True)
    result.reset_index(inplace=True, drop=True)
    return result
```

8.2.2 Classification and Input Perturbation Ranking

We now look at the code to perform perturbation ranking for a classification neural network. The implementation technique is slightly different for classification vs regression, so I must provide two different implementations. The primary difference between classification and regression is how we evaluate the accuracy of the neural network in each of these two network types. For regression neural networks, we will use the Root Mean Square (RMSE) error calculation; whereas, we will use log loss for classification.

The code presented below creates a classification neural network that will predict for the classic iris dataset.

Code

```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x_train,y_train,verbose=2,epochs=100)
```

Output

```
Train on 112 samples
Epoch 1/100
112/112 - 0s - loss: 1.1435
Epoch 2/100
112/112 - 0s - loss: 1.0499
Epoch 3/100
112/112 - 0s - loss: 0.9736
Epoch 4/100
112/112 - 0s - loss: 0.9220
Epoch 5/100
```

```
112/112 - 0s - loss: 0.8958
Epoch 6/100
112/112 - 0s - loss: 0.8628
Epoch 7/100
112/112 - 0s - loss: 0.8333

...

Epoch 99/100
112/112 - 0s - loss: 0.1187
Epoch 100/100
112/112 - 0s - loss: 0.1148
<tensorflow.python.keras.callbacks.History at 0x249cb0fba08>
```

Next, we evaluate the accuracy of the trained model. Here we see that the neural network is performing great, with an accuracy of 1.0. For a more complex dataset, we might fear overfitting with such high accuracy. However, for this example, we are more interested in determining the importance of each column.

Code

```
from sklearn.metrics import accuracy_score

pred = model.predict(x_test)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y_test,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

Output

```
Accuracy: 1.0
```

We are now ready to call the input perturbation algorithm. First, we extract the column names and remove the target column. The target column does not have importance, as it is the objective, not one of the inputs. In supervised learning, the target is of the utmost importance.

We can see the importance displayed in the following table. The most important column is always 1.0, and lessor columns will continue in a downward trend. The least important column will have the lowest rank.

Code

```
# Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("species") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank)
```

Output

	name	error	importance
0	petal_l	2.457038	1.000000
1	petal_w	0.564843	0.229888
2	sepal_l	0.206589	0.084081
3	sepal_w	0.186969	0.076095

8.2.3 Regression and Input Perturbation Ranking

We now see how to use input perturbation ranking for a regression neural network. We will use the MPG dataset as a demonstration. The code below loads the MPG dataset and creates a regression neural network for this dataset. The code trains the neural network and calculates an RMSE evaluation.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
        'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
```

```
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x_train,y_train,verbose=2,epochs=100)

# Predict
pred = model.predict(x)
```

Output

```
Train on 298 samples
Epoch 1/100
298/298 - 0s - loss: 40307.5655
Epoch 2/100
298/298 - 0s - loss: 12333.8158
Epoch 3/100
298/298 - 0s - loss: 1765.7875
Epoch 4/100
298/298 - 0s - loss: 1373.6327
Epoch 5/100
298/298 - 0s - loss: 726.2777
Epoch 6/100
298/298 - 0s - loss: 245.1450
Epoch 7/100
298/298 - 0s - loss: 135.2508

...

298/298 - 0s - loss: 37.1556
Epoch 99/100
298/298 - 0s - loss: 35.1785
Epoch 100/100
298/298 - 0s - loss: 33.2472
```

Just as before, we extract the column names and discard the target. We can now create a ranking of the importance of each of the input features. The feature with a ranking of 1.0 is the most important.

Code

```
# Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("name")
names.remove("mpg") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, True)
display(rank)
```

Output

	name	error	importance
0	displacement	287.936080	1.000000
1	weight	198.281957	0.688632
2	horsepower	32.233535	0.111947
3	year	30.184242	0.104830
4	cylinders	26.632769	0.092495
5	origin	26.346470	0.091501
6	acceleration	26.298675	0.091335

8.2.4 Biological Response with Neural Network

The following sections will demonstrate how to use feature importance ranking and ensembling with a more complex dataset. Ensembling is the process where you combine multiple models for greater accuracy. Kaggle competition winners frequently make use of ensembling for high ranking solutions.

We will use the biological response dataset, a Kaggle dataset, where there is an unusually high number of columns. Because of the large number of columns, it is essential to use feature ranking to determine the importance of these columns. We begin by loading the dataset and preprocessing. This Kaggle dataset is a binary classification problem. You must predict if certain conditions will cause a biological response.

- Predicting a Biological Response

Code

```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import KFold
from IPython.display import HTML, display

if COLAB:
    path = "/content/drive/My Drive/data/"
else:
    path = "./data/"

filename_train = os.path.join(path,"bio_train.csv")
filename_test = os.path.join(path,"bio_test.csv")
filename_submit = os.path.join(path,"bio_submit.csv")

df_train = pd.read_csv(filename_train,na_values=['NA','?'])
df_test = pd.read_csv(filename_test,na_values=['NA','?'])

activity_classes = df_train['Activity']
```

A large number of columns is evident when we display the shape of the dataset.

Code

```
print(df_train.shape)
```

Output

```
(3751, 1777)
```

The following code constructs a classification neural network and trains it for the biological response dataset. Once trained, the accuracy is measured.

Code

```
import os
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
import sklearn

# Encode feature vector
# Convert to numpy - Classification
x_columns = df_train.columns.drop('Activity')
x = df_train[x_columns].values
y = df_train['Activity'].values # Classification
x_submit = df_test[x_columns].values.astype(np.float32)


# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

print("Fitting/Training...")
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu'))
model.add(Dense(10))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto')
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
print("Fitting done...")

# Predict
```

```
pred = model.predict(x_test).flatten()


# Clip so that min is never exactly 0, max never 1
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
print("Validation logloss: {}".format(
    sklearn.metrics.log_loss(y_test,pred)))

# Evaluate success using accuracy
pred = pred>0.5 # If greater than 0.5 probability, then true
score = metrics.accuracy_score(y_test, pred)
print("Validation accuracy score: {}".format(score))

# Build real submit file
pred_submit = model.predict(x_submit)

# Clip so that min is never exactly 0, max never 1 (would be a NaN score)
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
submit_df = pd.DataFrame({'MoleculeId':[x+1 for x \
        in range(len(pred_submit))],'PredictedProbability':\
                            pred_submit.flatten()})
submit_df.to_csv(filename_submit, index=False)
```

Output

```
Fitting/Training...
Epoch 00008: early stopping
Fitting done...
Validation logloss: 0.5671799306256637
Validation accuracy score: 0.7643923240938166
```

8.2.5 What Features/Columns are Important

The following uses perturbation ranking to evaluate the neural network.

Code

```
# Rank the features
from IPython.display import display, HTML

names = list(df_train.columns) # x+y column names
names.remove("Activity") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank[0:10])
```

Output

	name	error	importance
0	D27	0.637687	1.000000
1	D1049	0.576325	0.903773
2	D1071	0.574874	0.901498
3	D961	0.574213	0.900462
4	D1407	0.573172	0.898829
5	D1333	0.572745	0.898159
6	D958	0.572173	0.897263
7	D1271	0.571364	0.895994
8	D1167	0.571352	0.895975
9	D51	0.571237	0.895795

8.2.6 Neural Network Ensemble

A neural network ensemble combines neural network predictions with other models. The program determines the exact blend of all of these models by logistic regression. The following code performs this blend for a classification. If you present the final predictions from the ensemble to Kaggle, you will see that the result is very accurate.

Code

```
import numpy as np
import os
import pandas as pd
import math
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

PATH = "./data/"
SHUFFLE = False
FOLDS = 10

def build_ann(input_size,classes,neurons):
    model = Sequential()
    model.add(Dense(neurons, input_dim=input_size, activation='relu'))
    model.add(Dense(1))
    model.add(Dense(classes,activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    return model

def mlogloss(y_test, preds):
    epsilon = 1e-15
    sum = 0
```

```
    for row in zip(preds,y_test):
        x = row[0][row[1]]
        x = max(epsilon,x)
        x = min(1-epsilon,x)
        sum+=math.log(x)
    return( (-1/len(preds))*sum)

def stretch(y):
    return (y - y.min()) / (y.max() - y.min())


def blend_ensemble(x, y, x_submit):
    kf = StratifiedKFold(FOLDS)
    folds = list(kf.split(x,y))

    models = [
        KerasClassifier(build_fn=build_ann,neurons=20,
                        input_size=x.shape[1],classes=2),
        KNeighborsClassifier(n_neighbors=3),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='gini'),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='entropy'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='gini'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='entropy'),
        GradientBoostingClassifier(learning_rate=0.05,
                subsample=0.5, max_depth=6, n_estimators=50)]

    dataset_blend_train = np.zeros((x.shape[0], len(models)))
    dataset_blend_test = np.zeros((x_submit.shape[0], len(models)))

    for j, model in enumerate(models):
        print("Model: {}: {}".format(j, model) )
        fold_sums = np.zeros((x_submit.shape[0], len(folds)))
        total_loss = 0
        for i, (train, test) in enumerate(folds):
            x_train = x[train]
            y_train = y[train]
            x_test = x[test]
            y_test = y[test]
            model.fit(x_train, y_train)
            pred = np.array(model.predict_proba(x_test))
            # pred = model.predict_proba(x_test)
            dataset_blend_train[test, j] = pred[:, 1]
            pred2 = np.array(model.predict_proba(x_submit))
            #fold_sums[:, i] = model.predict_proba(x_submit)[:, 1]
```

```
            fold_sums[:, i] = pred2[:, 1]
            loss = mlogloss(y_test, pred)
            total_loss+=loss
            print("Fold #{}: loss={}".format(i,loss))
        print("{}: Mean loss={}".format(model.__class__.__name__,
                                        total_loss/len(folds)))
        dataset_blend_test[:, j] = fold_sums.mean(1)

    print()
    print("Blending models.")
    blend = LogisticRegression(solver='lbfgs')
    blend.fit(dataset_blend_train, y)
    return blend.predict_proba(dataset_blend_test)

if __name__ == '__main__':

    np.random.seed(42)  # seed to shuffle the train set

    print("Loading data...")
    filename_train = os.path.join(PATH, "bio_train.csv")
    df_train = pd.read_csv(filename_train, na_values=['NA', '?'])

    filename_submit = os.path.join(PATH, "bio_test.csv")
    df_submit = pd.read_csv(filename_submit, na_values=['NA', '?'])

    predictors = list(df_train.columns.values)
    predictors.remove('Activity')
    x = df_train[predictors].values
    y = df_train['Activity']
    x_submit = df_submit.values

    if SHUFFLE:
        idx = np.random.permutation(y.size)
        x = x[idx]
        y = y[idx]

    submit_data = blend_ensemble(x, y, x_submit)
    submit_data = stretch(submit_data)

    ######################
    # Build submit file
    ######################
    ids = [id+1 for id in range(submit_data.shape[0])]
    submit_filename = os.path.join(PATH, "bio_submit.csv")
    submit_df = pd.DataFrame({'MoleculeId': ids,
                              'PredictedProbability':
                              submit_data[:, 1]},
                             columns=['MoleculeId',
```

```
                                'PredictedProbability'])
    submit_df.to_csv(submit_filename, index=False)
```

Output

```
Loading data...
Model: 0 :
<tensorflow.python.keras.wrappers.scikit_learn.KerasClassifier object
at 0x0000024F619B5E48>
Train on 3375 samples
3375/3375 [==============================] - 0s 110us/sample - loss:
0.6152
Fold #0: loss=0.5670192397249542
Train on 3376 samples
3376/3376 [==============================] - 0s 110us/sample - loss:
0.5876
Fold #1: loss=0.5181313306906605
Train on 3376 samples
3376/3376 [==============================] - 0s 108us/sample - loss:
0.5864

...

Fold #7: loss=0.4594068991947806
Fold #8: loss=0.463832905944738
Fold #9: loss=0.46507605175072847
GradientBoostingClassifier: Mean loss=0.46642776737562003
Blending models.
```

8.3 Part 8.3: Architecting Network: Hyperparameters

You have probably noticed several hyperparameters introduced previously in this course that you need to choose for your neural network. The number of layers, neuron counts per layers, layer types, and activation functions are all choices you must make to optimize your neural network. Some of the categories of hyperparameters for you to choose from come from the following list:

- Number of Hidden Layers and Neuron Counts
- Activation Functions
- Advanced Activation Functions
- Regularization: L1, L2, Dropout
- Batch Normalization
- Training Parameters

The following sections will introduce each of these categories for Keras. While I will provide you with some general guidelines for hyperparameter selection; no two tasks are the same. You will benefit from experimentation with these values to determine what works best for your neural network. In the next part, we will see how machine learning can select some of these values on its own.

8.3.1 Number of Hidden Layers and Neuron Counts

The structure of Keras layers is perhaps the hyperparameters that most become aware of first. How many layers should you have? How many neurons on each layer? What activation function and layer type should you use? These are all questions that come up when designing a neural network. There are many different types of layer in Keras, listed here:

- **Activation** - You can also add activation functions as layers. Making use of the activation layer is the same as specifying the activation function as part of a Dense (or other) layer type.
- **ActivityRegularization** Used to add L1/L2 regularization outside of a layer. You can specify L1 and L2 as part of a Dense (or other) layer type.
- **Dense** - The original neural network layer type. In this layer type, every neuron connects to the next layer. The input vector is one-dimensional, and placing specific inputs next to each other does not affect.
- **Dropout** - Dropout consists of randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting. Dropout only occurs during training.
- **Flatten** - Flattens the input to 1D and does not affect the batch size.
- **Input** - A Keras tensor is a tensor object from the underlying back end (Theano, TensorFlow, or CNTK), which we augment with specific attributes to build a Keras model just by knowing the inputs and outputs of the model.
- **Lambda** - Wraps arbitrary expression as a Layer object.
- **Masking** - Masks a sequence by using a mask value to skip timesteps.
- **Permute** - Permutes the dimensions of the input according to a given pattern. Useful for tasks such as connecting RNNs and convolutional networks.
- **RepeatVector** - Repeats the input n times.
- **Reshape** - Similar to Numpy reshapes.
- **SpatialDropout1D** - This version performs the same function as Dropout; however, it drops entire 1D feature maps instead of individual elements.
- **SpatialDropout2D** - This version performs the same function as Dropout; however, it drops entire 2D feature maps instead of individual elements
- **SpatialDropout3D** - This version performs the same function as Dropout; however, it drops entire 3D feature maps instead of individual elements.

There is always trial and error for choosing a good number of neurons and hidden layers. Generally, the number of neurons on each layer will be larger closer to the hidden layer and smaller towards the output layer. This configuration gives the neural network a somewhat triangular or trapezoid appearance.

8.3.2 Activation Functions

Activation functions are a choice that you must make for each layer. Generally, you can follow this guideline:

- Hidden Layers - RELU
- Output Layer - Softmax for classification, linear for regression.

Some of the common activation functions in Keras are listed here:

- **softmax** - Used for multi-class classification. Ensures all output neurons behave as probabilities and sum to 1.0.
- **elu** - Exponential linear unit. Exponential Linear Unit or its widely known name ELU is a function that tend to converge cost to zero faster and produce more accurate results. Can produce negative outputs.

- **selu** - Scaled Exponential Linear Unit (SELU), essentially **elu** multiplied by a scaling constant.
- **softplus** - Softplus activation function. $log(exp(x) + 1)$ Introduced in 2001.
- **softsign** Softsign activation function. $x/(abs(x) + 1)$ Similar to tanh, but not widely used.
- **relu** - Very popular neural network activation function. Used for hidden layers, cannot output negative values. No trainable parameters.
- **tanh** Classic neural network activation function, though often replaced by relu family on modern networks.
- **sigmoid** - Classic neural network activation. Often used on output layer of a binary classifier.
- **hard_sigmoid** - Less computationally expensive variant of sigmoid.
- **exponential** - Exponential (base e) activation function.
- **linear** - Pass through activation function. Usually used on the output layer of a regression neural network.

For more information about Keras activation functions refer to the following:

- Keras Activation Functions
- Activation Function Cheat Sheets

8.3.3 Advanced Activation Functions

Hyperparameters are not changed when the neural network trains. You, the network designer, must define the hyperparameters. The neural network learns regular parameters during neural network training. Neural network weights are the most common type of regular parameter. The "advanced activation functions," as Keras call them, also contain parameters that the network will learn during training. These activation functions may give you better performance than RELU.

- **LeakyReLU** - Leaky version of a Rectified Linear Unit. It allows a small gradient when the unit is not active, controlled by alpha hyperparameter.
- **PReLU** - Parametric Rectified Linear Unit, learns the alpha hyperparameter.

8.3.4 Regularization: L1, L2, Dropout

- Keras Regularization
- Keras Dropout

8.3.5 Batch Normalization

- Keras Batch Normalization
- Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1. Can allow learning rate to be larger.

8.3.6 Training Parameters

- Keras Optimizers
- **Batch Size** - Usually small, such as 32 or so.
- **Learning Rate** - Usually small, 1e-3 or so.

8.3.7 Experimenting with Hyperparameters

Code

```
import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Code

```
import pandas as pd
import os
import numpy as np
import time
import tensorflow.keras.initializers
import statistics
import tensorflow.keras
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit
from tensorflow.keras.layers import LeakyReLU,PReLU
from tensorflow.keras.optimizers import Adam



def evaluate_network(dropout,lr,neuronPct,neuronShrink):
    SPLITS = 2

    # Bootstrap
    boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1)

    # Track progress
    mean_benchmark = []
    epochs_needed = []
    num = 0
    neuronCount = int(neuronPct * 5000)

    # Loop through samples
    for train, test in boot.split(x,df['product']):
        start_time = time.time()
        num+=1

        # Split train and test
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]

        # Construct neural network
        # kernel_initializer =
        # tensorflow.keras.initializers.he_uniform(seed=None)
        model = Sequential()

        layer = 0
        while neuronCount>25 and layer<10:
            #print(neuronCount)
            if layer==0:
                model.add(Dense(neuronCount,
                    input_dim=x.shape[1],
                    activation=PReLU()))
            else:
                model.add(Dense(neuronCount, activation=PReLU()))
            model.add(Dropout(dropout))
```

```
            neuronCount = neuronCount * neuronShrink

        model.add(Dense(y.shape[1],activation='softmax')) # Output
        model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=lr))
        monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
            patience=100, verbose=0, mode='auto', restore_best_weights=True)

        # Train on the bootstrap sample
        model.fit(x_train,y_train,validation_data=(x_test,y_test),
                  callbacks=[monitor],verbose=0,epochs=1000)
        epochs = monitor.stopped_epoch
        epochs_needed.append(epochs)

        # Predict on the out of boot (validation)
        pred = model.predict(x_test)

        # Measure this bootstrap's log loss
        y_compare = np.argmax(y_test,axis=1) # For log loss calculation
        score = metrics.log_loss(y_compare, pred)
        mean_benchmark.append(score)
        m1 = statistics.mean(mean_benchmark)
        m2 = statistics.mean(epochs_needed)
        mdev = statistics.pstdev(mean_benchmark)

        # Record this iteration
        time_took = time.time() - start_time

    tensorflow.keras.backend.clear_session()
    return (-m1)

print(evaluate_network(
    dropout=0.2,
    lr=1e-3,
    neuronPct=0.2,
    neuronShrink=0.2))
```

Output

```
-0.8013879325822928
```

8.4 Part 8.4: Bayesian Hyperparameter Optimization for Keras

Bayesian Hyperparameter Optimization is a method of finding hyperparameters in a more efficient way than a grid search. Because each candidate set of hyperparameters requires a retraining of the neural network, it is best to keep the number of candidate sets to a minimum. Bayesian Hyperparameter Optimization achieves

this by training a model to predict good candidate sets of hyperparameters.[36]

- bayesian-optimization
- hyperopt
- spearmint

Code

```
# Ignore useless W0819 warnings generated by TensorFlow 2.0.
# Hopefully can remove this ignore in the future.
# See https://github.com/tensorflow/tensorflow/issues/31308
import logging, os
logging.disable(logging.WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Now that we've preprocessed the data, we can begin the hyperparameter optimization. We start by creating a function that generates the model based on just three parameters. Bayesian optimization works on a vector of numbers, not on a problematic notion like how many layers and neurons are on each layer. To represent this complex neuron structure as a vector, we use several numbers to describe this structure.

- **dropout** - The dropout percent for each layer.
- **neuronPct** - What percent of our fixed 5,000 maximum number of neurons do we wish to use? This parameter specifies the total count of neurons in the entire network.
- **neuronShrink** - Neural networks usually start with more neurons on the first hidden layer and then decrease this count for additional layers. This percent specifies how much to shrink subsequent layers based on the previous layer. Once we run out of neurons (with the count specified by neuronPft), we stop adding more layers.

These three numbers define the structure of the neural network. The commends in the below code show exactly how the program constructs the network.

Code

```
import pandas as pd
import os
import numpy as np
import time
import tensorflow.keras.initializers
import statistics
import tensorflow.keras
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout, InputLayer
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit
from tensorflow.keras.layers import LeakyReLU,PReLU
from tensorflow.keras.optimizers import Adam

def generate_model(dropout, neuronPct, neuronShrink):
    # We start with some percent of 5000 starting neurons on
    # the first hidden layer.
    neuronCount = int(neuronPct * 5000)

    # Construct neural network
    model = Sequential()

    # So long as there would have been at least 25 neurons and
    # fewer than 10
    # layers, create a new layer.
    layer = 0
    while neuronCount>25 and layer<10:
        # The first (0th) layer needs an input input_dim(neuronCount)
        if layer==0:
            model.add(Dense(neuronCount,
```

```
                input_dim=x.shape[1],
                activation=PReLU()))
        else:
            model.add(Dense(neuronCount, activation=PReLU()))
        layer += 1

        # Add dropout after each hidden layer
        model.add(Dropout(dropout))

        # Shrink neuron count for each layer
        neuronCount = neuronCount * neuronShrink

    model.add(Dense(y.shape[1],activation='softmax')) # Output
    return model
```

We can test this code to see how it creates a neural network based on three such parameters.

Code

```
# Generate a model and see what the resulting structure looks like.
model = generate_model(dropout=0.2, neuronPct=0.1, neuronShrink=0.25)
model.summary()
```

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 500)               24500
_________________________________________________________________
dropout (Dropout)            (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 125)               62750
_________________________________________________________________
dropout_1 (Dropout)          (None, 125)               0
_________________________________________________________________
dense_2 (Dense)              (None, 31)                3937
_________________________________________________________________
dropout_2 (Dropout)          (None, 31)                0

...

=================================================================
Total params: 91,411
Trainable params: 91,411
Non-trainable params: 0
_________________________________________________________________
```

Now we create a function to evaluate the neural network, using three such parameters. We use bootstrapping because one single training run might simply have "bad luck" with the random weights assigned. We use this function to train and then evaluate the neural network.

Code

```
def evaluate_network(dropout,lr,neuronPct,neuronShrink):
    SPLITS = 2

    # Bootstrap
    boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1)

    # Track progress
    mean_benchmark = []
    epochs_needed = []
    num = 0


    # Loop through samples
    for train, test in boot.split(x,df['product']):
        start_time = time.time()
        num+=1

        # Split train and test
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]

        model = generate_model(dropout, neuronPct, neuronShrink)
        model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=lr))
        monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=100, verbose=0, mode='auto', restore_best_weights=True)

        # Train on the bootstrap sample
        model.fit(x_train,y_train,validation_data=(x_test,y_test),
                  callbacks=[monitor],verbose=0,epochs=1000)
        epochs = monitor.stopped_epoch
        epochs_needed.append(epochs)

        # Predict on the out of boot (validation)
        pred = model.predict(x_test)

        # Measure this bootstrap's log loss
        y_compare = np.argmax(y_test,axis=1) # For log loss calculation
        score = metrics.log_loss(y_compare, pred)
        mean_benchmark.append(score)
        m1 = statistics.mean(mean_benchmark)
        m2 = statistics.mean(epochs_needed)
        mdev = statistics.pstdev(mean_benchmark)
```

```
        # Record this iteration
        time_took = time.time() - start_time

    tensorflow.keras.backend.clear_session()
    return (-m1)
```

Output

```
-0.6720430161559489
```

You can try any combination of our three hyperparameters, plus the learning rate, to see how effective these four numbers are. Of course, our goal is not to manually choose different combinations of these four hyperparameters; we seek to automate.

Code

```
print(evaluate_network(
    dropout=0.2,
    lr=1e-3,
    neuronPct=0.2,
    neuronShrink=0.2))
```

We will now automat this process. We define the bounds for each of these four hyperparameters and begin the Bayesian optimization. Once the program completes, the best combination of hyperparameters found is displayed.

Code

```
from bayes_opt import BayesianOptimization
import time

# Supress NaN warnings
import warnings
warnings.filterwarnings("ignore",category =RuntimeWarning)

# Bounded region of parameter space
pbounds = {'dropout': (0.0, 0.499),
           'lr': (0.0, 0.1),
           'neuronPct': (0.01, 1),
           'neuronShrink': (0.01, 1)
          }

optimizer = BayesianOptimization(
    f=evaluate_network,
    pbounds=pbounds,
    verbose=2,  # verbose = 1 prints only when a maximum
    # is observed, verbose = 0 is silent
```

```
    random_state=1,
)

start_time = time.time()
optimizer.maximize(init_points=10, n_iter=100,)
time_took = time.time() - start_time

print(f"Total runtime: {hms_string(time_took)}")
print(optimizer.max)
```

Output

```
|   iter    |  target   |  dropout  |    lr     | neuronPct |
neuron... |
-------------------------------------------------------------------------
---
|  1        | -0.7124   |  0.2081   |  0.07203  |  0.01011  |  0.3093
|
|  2        | -0.7159   |  0.07323  |  0.009234 |  0.1944   |  0.3521
|
|  3        | -13.85    |  0.198    |  0.05388  |  0.425    |  0.6884
|
|  4        | -0.8022   |  0.102    |  0.08781  |  0.03711  |  0.6738
|
|  5        | -0.8295   |  0.2082   |  0.05587  |  0.149    |  0.2061
|
|  6        | -16.32    |  0.3996   |  0.09683  |  0.3203   |  0.6954

...

=========
Total runtime: 3:31:16.15
{'target': -0.6134732591234944, 'params': {'dropout':
0.07250303494641018, 'lr': 0.010879552981584925, 'neuronPct':
0.19549892758598567, 'neuronShrink': 0.3480265521049257}}
```

{'target': -0.6500334282952827, 'params': {'dropout': 0.12771198428037775, 'lr': 0.0074010841641111965, 'neuronPct': 0.10774655638231533, 'neuronShrink': 0.278478867649825 7}}

8.5 Part 8.5: Current Semester's Kaggle

Kaggke competition site for current semester (Fall 2020):

- Fall 2020 Kaggle Assignment

Previous Kaggle competition sites for this class (NOT this semester's assignment, feel free to use code):

- Spring 2020 Kaggle Assignment

- Fall 2019 Kaggle Assignment
- Spring 2019 Kaggle Assignment
- Fall 2018 Kaggle Assignment
- Spring 2018 Kaggle Assignment
- Fall 2017 Kaggle Assignment
- Spring 2017 Kaggle Assignment
- Fall 2016 Kaggle Assignment

8.5.1 Iris as a Kaggle Competition

If the Iris data were used as a Kaggle, you would be given the following three files:

- kaggle_iris_test.csv - The data that Kaggle will evaluate you on. Contains only input, you must provide answers. (contains x)
- kaggle_iris_train.csv - The data that you will use to train. (contains x and y)
- kaggle_iris_sample.csv - A sample submission for Kaggle. (contains x and y)

Important features of the Kaggle iris files (that differ from how we've previously seen files):

- The iris species is already index encoded.
- Your training data is in a separate file.
- You will load the test data to generate a submission file.

The following program generates a submission file for "Iris Kaggle". You can use it as a starting point for assignment 3.

Code

```
import os
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df_train = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_iris_train.csv", na_values=['NA','?'])

# Encode feature vector
df_train.drop('id', axis=1, inplace=True)

num_classes = len(df_train.groupby('species').species.nunique())

print("Number of classes: {}".format(num_classes))

# Convert to numpy - Classification
x = df_train[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df_train['species']) # Classification
```

```
species = dummies.columns
y = dummies.values

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=45)

# Train, with early stopping
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
model.add(Dense(25))
model.add(Dense(y.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto',
                        restore_best_weights=True)

model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
```

Output

```
Number of classes: 3
Restoring model weights from the end of the best epoch.
Epoch 00055: early stopping
<tensorflow.python.keras.callbacks.History at 0x178e5493fc8>
```

Now that we've trained the neural network, we can check its log loss.

Code

```
from sklearn import metrics

# Calculate multi log loss error
pred = model.predict(x_test)
score = metrics.log_loss(y_test, pred)
print("Log loss score: {}".format(score))
```

Output

```
Log loss score: 0.3136451941728592
```

Now we are ready to generate the Kaggle submission file. We will use the iris test data that does not contain a y target value. It is our job to predict this value and submit to Kaggle.

Code

```
# Generate Kaggle submit file

# Encode feature vector
df_test = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_iris_test.csv", na_values=['NA','?'])

# Convert to numpy - Classification
ids = df_test['id']
df_test.drop('id', axis=1, inplace=True)
x = df_test[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
y = dummies.values

# Generate predictions
pred = model.predict(x)
#pred

# Create submission data set

df_submit = pd.DataFrame(pred)
df_submit.insert(0,'id',ids)
df_submit.columns = ['id','species-0','species-1','species-2']

# Write submit file locally
df_submit.to_csv("iris_submit.csv", index=False)

print(df_submit)
```

Output

```
id  species-0  species-1  species-2
0   100   0.022236   0.533230   0.444534
1   101   0.003699   0.394908   0.601393
2   102   0.004600   0.420394   0.575007
3   103   0.956168   0.040161   0.003672
4   104   0.975333   0.022761   0.001906
5   105   0.966681   0.030938   0.002381
6   106   0.992637   0.007049   0.000314
7   107   0.002810   0.358485   0.638705
8   108   0.026152   0.557480   0.416368
9   109   0.001194   0.350682   0.648124
10  110   0.000649   0.268023   0.731328
11  111   0.994907   0.004923   0.000170
12  112   0.072954   0.587299   0.339747
13  113   0.000571   0.258208   0.741221
```

```
...

46  146   0.130497   0.579122   0.290381
47  147   0.023003   0.499443   0.477553
48  148   0.022195   0.527769   0.450036
49  149   0.983695   0.015325   0.000980
50  150   0.942703   0.052154   0.005144
```

8.5.2 MPG as a Kaggle Competition (Regression)

If the Auto MPG data were used as a Kaggle, you would be given the following three files:

- kaggle_mpg_test.csv - The data that Kaggle will evaluate you on. Contains only input, you must provide answers. (contains x)
- kaggle_mpg_train.csv - The data that you will use to train. (contains x and y)
- kaggle_mpg_sample.csv - A sample submission for Kaggle. (contains x and y)

Important features of the Kaggle iris files (that differ from how we've previously seen files):

The following program generates a submission file for "MPG Kaggle".

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_auto_train.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression
```

```
# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          verbose=2,callbacks=[monitor],epochs=1000)

# Predict
pred = model.predict(x_test)
```

Output

```
Train on 261 samples, validate on 88 samples
Epoch 1/1000
261/261 - 0s - loss: 382597.1196 - val_loss: 246687.4858
Epoch 2/1000
261/261 - 0s - loss: 192257.0072 - val_loss: 98804.3558
Epoch 3/1000
261/261 - 0s - loss: 67605.7908 - val_loss: 28617.0703
Epoch 4/1000
261/261 - 0s - loss: 15922.8367 - val_loss: 3325.1682
Epoch 5/1000
261/261 - 0s - loss: 1270.3832 - val_loss: 512.5387
Epoch 6/1000
261/261 - 0s - loss: 1118.9636 - val_loss: 1651.5679
Epoch 7/1000
261/261 - 0s - loss: 1703.0441 - val_loss: 1161.2368

...

261/261 - 0s - loss: 25.1895 - val_loss: 20.4868
Epoch 295/1000
Restoring model weights from the end of the best epoch.
261/261 - 0s - loss: 23.6036 - val_loss: 20.3795
Epoch 00295: early stopping
```

Now that we've trained the neural network, we can check its RMSE error.

Code

```
import numpy as np

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Final score (RMSE): {}".format(score))
```

Output

```
Final score (RMSE): 4.5134384517538795
```

Now we are ready to generate the Kaggle submission file. We will use the MPG test data that does not contain a y target value. It is our job to predict this value and submit to Kaggle.

Code

```
import pandas as pd

# Generate Kaggle submit file

# Encode feature vector
df_test = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_auto_test.csv", na_values=['NA','?'])

# Convert to numpy - regression
ids = df_test['id']
df_test.drop('id', axis=1, inplace=True)

# Handle missing value
df_test['horsepower'] = df_test['horsepower'].\
    fillna(df['horsepower'].median())

x = df_test[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values




# Generate predictions
pred = model.predict(x)
#pred

# Create submission data set

df_submit = pd.DataFrame(pred)
df_submit.insert(0,'id',ids)
```

```
df_submit.columns = ['id','mpg']

# Write submit file locally
df_submit.to_csv("auto_submit.csv", index=False)

print(df_submit)
```

Output

```
id         mpg
0    350  29.112602
1    351  27.803200
2    352  27.981804
3    353  30.487831
4    354  27.227440
5    355  26.438324
6    356  27.886986
7    357  29.103935
8    358  26.447609
9    359  30.027260
10   360  30.312553
11   361  30.712151
12   362  23.952263
13   363  24.858467

...

44   394  26.575218
45   395  33.546684
46   396  24.233910
47   397  28.609993
48   398  28.913261
```

8.5.3 Module 8 Assignment

You can find the first assignment here: assignment 8

Chapter 9

Transfer Learning

9.1 Part 9.1: Introduction to Keras Transfer Learning

Human beings learn new skills throughout their entire lives. However, this learning is rarely from scratch. No matter what task a human is learning, they are most likely drawing on experiences early in life to learn this new skill. In this way, humans learn much differently than most deep learning projects.

At some point, a human being learns to tell the difference between a cat and a dog. To teach a neural network, the difference you would obtain a large number of cat pictures and dog pictures. The neural network would iterate over all of these pictures and train on the differences. The human child that learned to distinguish between the two animals would probably need to see a few examples when parents told them the name of each type of animal. The human child would use previous knowledge of looking at different living and non-living objects to help make this classification. The child would already know the physical appearance of sub-objects, such as fur, eyes, ears, noses, tails, and teeth.

Transfer learning attempts to teach a neural network by similar means. Rather than training your neural network from scratch, you begin training with a preloaded set of weights. Usually, you will remove the topmost layers of the pretrained neural network and retrain it with new uppermost layers. The layers remaining from the previous neural network will be locked so that training does not change these weights. Only the newly added layers will be trained. Figure 9.1 summarizes this process.

It can take a great deal of computing power to train a neural network for a large image dataset. Google, Facebook, Microsoft, and other tech companies have utilized GPU arrays for training high-quality neural networks for a variety of applications. Transferring these weights into your neural network can save you considerable effort and compute time. It is unlikely that a pretrained model will exactly fit the application that you seek to implement. Finding the closest pretrained model and using transfer learning is an essential skill for a deep learning engineer.

The imagenet dataset has several high-quality neural networks that fit it.[5]In the next parts of this module, we will take a much closer look at this data set. For many image recognition tasks, an imagenet trained neural network can be a great starting point for your neural networks.

9.1.1 Transfer Learning Example

Let's look at an example of where we could use transfer learning to build upon an imagenet neural network. Microsoft released a website that accepts a picture of a dog and attempts to classify them by breed. We provide the Microsoft dog breed website here:

What breed is that dog?

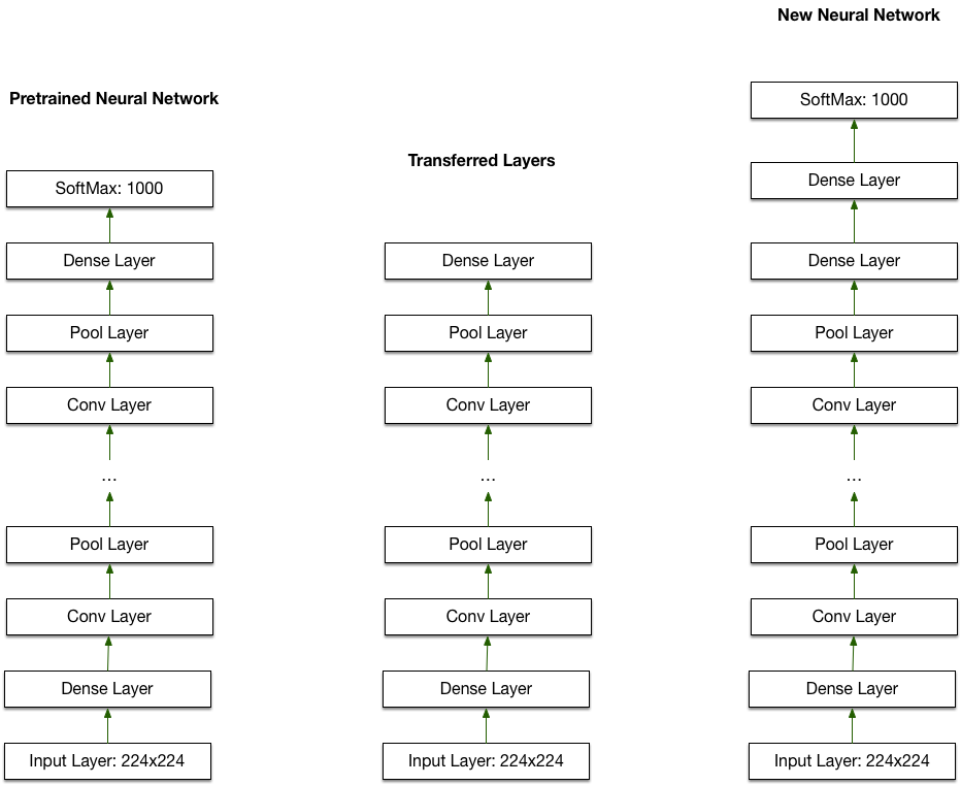


Figure 9.1: Transfer Learning

Let's see how we can classify a specific dog breed. For this task, it is necessary to obtain pictures of dogs labeled according to breed. Such a network could be trained entirely from scratch. However, it would require a large quantity of breed-labeled images. Transfer learning with imagenet could be very beneficial for a neural network project such as this. A neural network pre-trained on imagenet would already contain neurons that can recognize many subcomponents of the various dog breeds that the neural network had previously seen on the other animal images in imagenet.

Finally, we evaluate the accuracy of the predictions.

Code

```
import pandas as pd
import io
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/iris.csv",
```

```
    na_values=['NA', '?'])

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values


# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output

model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)
```

Output

```
Train on 150 samples
Epoch 1/100
150/150 - 1s - loss: 1.0437
Epoch 2/100
150/150 - 0s - loss: 0.9085
Epoch 3/100
150/150 - 0s - loss: 0.8192
Epoch 4/100
150/150 - 0s - loss: 0.7486
Epoch 5/100
150/150 - 0s - loss: 0.6859
Epoch 6/100
150/150 - 0s - loss: 0.6365
Epoch 7/100
150/150 - 0s - loss: 0.5906

...

Epoch 99/100
150/150 - 0s - loss: 0.0794
Epoch 100/100
150/150 - 0s - loss: 0.0848
<tensorflow.python.keras.callbacks.History at 0x1d1fb8db988>
```

To keep this example simple, we are not setting aside a validation set. The goal of this example is to show how to create a multi-layer neural network, where we transfer the weights to another network. We begin by evaluating the accuracy of the network on the training set.

Code

```
from sklearn.metrics import accuracy_score
pred = model.predict(x)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Training Accuracy: {correct}")
```

Output

```
Training Accuracy: 0.9866666666666667
```

Viewing the model summary is as expected; we can see the three layers previously defined.

Code

```
model.summary()
```

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                250
_________________________________________________________________
dense_1 (Dense)              (None, 25)                1275
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 78
=================================================================
Total params: 1,603
Trainable params: 1,603
Non-trainable params: 0
_________________________________________________________________
```

Now that we've trained a neural network on the iris dataset, we can transfer the knowledge of this neural network to other neural networks. It is possible to create a new neural network from some or all of the layers of this neural network. To demonstrate the technique, we will create a new neural network that is essentially a clone of the first neural network. We now transfer all of the layers from the original neural network into the new one.

Code

```
model2 = Sequential()
for layer in model.layers:
    model2.add(layer)
model2.summary()
```

Output

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                250
_________________________________________________________________
dense_1 (Dense)              (None, 25)                1275
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 78
=================================================================
Total params: 1,603
Trainable params: 1,603
Non-trainable params: 0
_________________________________________________________________
```

As a sanity check, we would like to calculate the accuracy of the newly created model. The in-sample accuracy should be the same as the previous model that the new model transferred.

Code

```
from sklearn.metrics import accuracy_score
pred = model2.predict(x)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Training Accuracy: {correct}")
```

Output

```
Training Accuracy: 0.9866666666666667
```

The in-sample accuracy of the newly created neural network is the same as the first neural network. We've successfully transferred all of the layers from the original neural network.

For this example, we will train a neural network to classify three new hypothetical flowers that are uncreatively named:

- Fake Flower 1
- Fake Flower 2
- Fake Flower 3
- Fake Flower 4

We have measurements for samples of these flowers that conform to the predictors contained in the original iris dataset: sepal width, sepal length, petal width, and petal length. For transfer learning to be effective, the input for the newly trained neural network most closely conforms to the first neural network we transfer.

We will strip away the last output layer that contains the softmax activation function that performs this final classification. We will create a new output layer that will classify the four new flowers. We will only

train the weights in this new layer. We will mark the first two layers as non-trainable. The hope is that the first few layers have learned to abstract the raw input data in a way that is also helpful to the new neural network.

This process is accomplished by looping over the first few layers and copying them to the new neural network. We output a summary of the new neural network to verify that Keras stripped the previous output layer.

Code

```
model3 = Sequential()
for i in range(2):
    layer = model.layers[i]
    layer.trainable = False
    model3.add(layer)
model3.summary()
```

Output

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                250
_________________________________________________________________
dense_1 (Dense)              (None, 25)                1275
=================================================================
Total params: 1,525
Trainable params: 0
Non-trainable params: 1,525
_________________________________________________________________
```

To complete the new neural network, we add a 4-neuron classification layer and compile for classification.

Code

```
model3.add(Dense(4,activation='softmax')) # Output

model3.compile(loss='categorical_crossentropy', optimizer='adam')
model3.summary()
```

Output

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                250
_________________________________________________________________
```

```
dense_1 (Dense)              (None, 25)                1275
_________________________________________________________________
dense_3 (Dense)              (None, 4)                 104
=================================================================
Total params: 1,629
Trainable params: 104
Non-trainable params: 1,525
_________________________________________________________________
```

Next we generate some training data for the 4 fake flowers, and train the neural network.

Code

```
x = np.array([
    [2.1,0.9,0.8,1.1], # 1
    [2.5,1.2,0.8,1.2],
    [1.1,3.1,1.1,1.1], # 2
    [0.8,2.2,0.7,1.2],
    [1.2,0.7,3.1,1.1], # 3
    [1.0,1.1,2.4,0.9],
    [0.1,1.1,4.1,1.2], # 4
    [1.2,0.8,3.1,0.1],
])

y = np.array([
    [0,0,0,1],
    [0,0,0,1],
    [0,0,1,0],
    [0,0,1,0],
    [0,1,0,0],
    [0,1,0,0],
    [1,0,0,0],
    [1,0,0,0],
])

model3.fit(x,y,verbose=0,epochs=1000)
```

Output

```
<tensorflow.python.keras.callbacks.History at 0x1d1ff923e08>
```

We can evaluate the in-sample accuracy for the new model containing transferred layers from the previous model.

Code

```
from sklearn.metrics import accuracy_score
pred = model3.predict(x)
predict_classes = np.argmax(pred,axis=1)
```

```
expected_classes = np.argmax(y,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Training Accuracy: {correct}")
```

Output

```
Training Accuracy: 1.0
```

9.1.2 Module 9 Assignment

You can find the first assignment here: assignment 9

9.2 Part 9.2: Popular Pretrained Neural Networks for Keras

The following two sites, among others, can be great starting points to find pretrained models for use in your projects:

- TensorFlow Model Zoo
- Papers with Code

Keras contains built-in support for several pretrained models. You can find the complete list in the Keras documentation.

9.2.1 DenseNet

The Dense Convolutional Network (DenseNet) model is provided by keras. Recent work has shown that convolutional networks can be substantially deeper, more accurate, and efficient to train if they contain shorter connections between layers close to the input and those close to the output. In this paper, we embrace this observation and introduce the Dense Convolutional Network (DenseNet), which connects each layer to every other layer in a feed-forward fashion. Whereas traditional convolutional networks with L layers have L connections - one between each layer and its subsequent layer - our network has L(L+1)/2 direct connections. For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters. We evaluate our proposed architecture on four highly competitive object recognition benchmark tasks (CIFAR-10, CIFAR-100, SVHN, and ImageNet). DenseNets obtain significant improvements over the state-of-the-art on most of them, while requiring less computation to achieve high performance. Code and pre-trained models are available in Keras.

9.2.2 InceptionResNetV2 and InceptionV3

The Inception ResNet V2 model is provided by Keras. A convolutional neural network (CNN) achieves a new state of the art in terms of accuracy on the ILSVRC image classification benchmark. Inception-ResNet-v2 is a variation of our earlier Inception V3 model, which borrows some ideas from Microsoft's ResNet papers.

9.2.3 MobileNet

The MobileNet model is [provided by Keras].(https://keras.io/applications/#mobilenetv2). Created for mobile and embedded vision applications. MobileNets are based on a streamlined architecture that uses depth-wise separable convolutions to build lightweight deep neural networks. We introduce two simple global hyper-parameters that efficiently trade-off between latency and accuracy. These hyper-parameters allow the model builder to choose the right-sized model for their application based on the problem's constraints. We present extensive experiments on resource and accuracy tradeoffs and show strong performance compared to other popular models on ImageNet classification. We then demonstrate the effectiveness of MobileNets across a wide range of applications and use cases including object detection, fine-grain classification, face attributes, and large scale geo-localization.

9.2.4 MobileNetV2

The MobileNet V2 model is provided by Keras. Improves the state of the art performance of mobile models on multiple tasks and benchmarks as well as across a spectrum of different model sizes. We also describe efficient ways of applying these mobile models to object detection in a novel framework we call SSDLite. We also demonstrate how to build mobile semantic segmentation models through a reduced form of DeepLabv3, which we call Mobile DeepLabv3.

9.2.5 NASNet

The NASNet model is provided by Keras. Developing neural network image classification models often requires significant architecture engineering. In this paper, we study a method to learn the model architectures directly on the dataset of interest. As this approach is expensive when the dataset is large, we propose searching for an architectural building block on a small dataset and then transferring it to a larger dataset. The key contribution of this work is the design of a new search space (the "NASNet search space"), which enables transferability. In our experiments, we search for the best convolutional layer (or "cell") on the CIFAR-10 dataset and then apply this cell to the ImageNet dataset by stacking together more copies of this cell, each with their own parameters to design a convolutional architecture, named "NASNet architecture." We also introduce a new regularization technique called ScheduledDropPath that significantly improves generalization in the NASNet models. On CIFAR-10 itself, NASNet achieves a 2.4% error rate, which is state-of-the-art. On ImageNet, NASNet achieves, among the published works, state-of-the-art accuracy of 82.7% top-1 and 96.2% top-5 on ImageNet. Our model is 1.2% better in top-1 accuracy than the best human-invented architectures. It has 9 billion fewer FLOPS - a reduction of 28% in computational demand from the previous state-of-the-art model. When evaluated at different levels of computational cost, NASNets' accuracy exceeds those of the state-of-the-art human-designed models. For instance, a small version of NASNet achieves 74% top-1 accuracy, which is 3.1% better than equivalently-sized, state-of-the-art models for mobile platforms. Finally, the learned features by NASNet used with the Faster-RCNN framework surpass state-of-the-art by 4.0% achieving 43.1% mAP on the COCO dataset.

9.2.6 ResNet, ResNetV2, ResNeXt

The ResNet, ResNetV2, and ResNeXt models are provided by Keras. Deep residual networks have emerged as a family of extremely deep architectures showing compelling accuracy and nice convergence behaviors. In this paper, we analyze the propagation formulations behind the residual building blocks, which suggest that the forward and backward signals can be directly propagated from one block to any other block when using identity mappings as the skip connections and after-addition activation. A series of ablation experiments support the importance of these identity mappings. This motivates us to propose a new residual unit, which

makes training easier and improves generalization. We report improved results using a 1001-layer ResNet on CIFAR-10 (4.62% error) and CIFAR-100, and a 200-layer ResNet on ImageNet.

9.2.7 VGG16 and VGG19

The VGG16 and VGG19 models are provided by Keras. In this work, we investigate the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. Our main contribution is a thorough evaluation of networks of increasing depth using an architecture with tiny (3x3) convolution filters. This shows that a significant improvement in the prior-art configurations can be achieved by pushing the depth to 16-19 weight layers. These findings were the basis of our ImageNet Challenge 2014 submission, where our team secured the first and the second places in the localization and classification tracks, respectively. We also show that our representations generalize well to other datasets, where they achieve state-of-the-art results. We have made our two best-performing ConvNet models publicly available to facilitate further research on the use of deep visual representations in computer vision.

9.2.8 Xception

The Xception model is provided by Keras. We present an interpretation of Inception modules in convolutional neural networks as an intermediate step in-between regular convolution and the depthwise separable convolution operation (a depthwise convolution followed by a pointwise convolution). In this light, a depthwise separable convolution can be understood as an Inception module with a maximally large number of towers. This observation leads us to propose a novel deep convolutional neural network architecture inspired by Inception, where Inception modules have been replaced with depthwise separable convolutions. We show that this architecture, dubbed Xception, slightly outperforms Inception V3 on the ImageNet dataset (which Inception V3 was designed for). It significantly outperforms Inception V3 on a larger image classification dataset comprising 350 million images and 17,000 classes. Since the Xception architecture has the same number of parameters as Inception V3, the performance gains are not due to increased capacity but rather to more efficient use of model parameters.

9.3 Part 9.3: Transfer Learning for Computer Vision and Keras

In this part, we will use transfer learning to create a simple neural network that can new images that are not in MobileNet. To keep the example simple, we will only train for older storage technologies, such as floppy disks, tapes, etc. This dataset can be downloaded from the following location: download link.

To keep computation times to a minimum, we will make use of the MobileNet included in Keras. We will begin by loading the entire MobileNet and seeing how well it classifies with several test images. MobileNet can classify 1,000 different images. We will ultimately extend it to classify image types that are not in its dataset, in this example, four media types.

Code

```
import pandas as pd
import numpy as np
import os
import tensorflow.keras
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense,GlobalAveragePooling2D
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.preprocessing import image
```

```
from tensorflow.keras.applications.mobilenet import preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
```

We begin by downloading weights for a MobileNet trained for the imagenet dataset, which will take some time to download the first time you train the network.

Code

```
model = MobileNet(weights='imagenet',include_top=True)
```

The loaded network is a Keras neural network. However, this is a neural network that a third party engineered on advanced hardware. Merely looking at the structure of an advanced state-of-the-art neural network can be educational.

Code

```
model.summary()
```

Output

```
Model: "mobilenet_1.00_224"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
conv1_pad (ZeroPadding2D)    (None, 225, 225, 3)       0
_________________________________________________________________
conv1 (Conv2D)               (None, 112, 112, 32)      864
_________________________________________________________________
conv1_bn (BatchNormalization (None, 112, 112, 32)      128
_________________________________________________________________
conv1_relu (ReLU)            (None, 112, 112, 32)      0
_________________________________________________________________
conv_dw_1 (DepthwiseConv2D)  (None, 112, 112, 32)      288

...

=================================================================
Total params: 4,253,864
Trainable params: 4,231,976
Non-trainable params: 21,888
_________________________________________________________________
```

Just examining the above structure, several clues to neural network architecture become evident.

Notice how some of the layers have zeros in their number of parameters. The summary always displays hyperparameters as zero. The neural network fitting process does not change hyperparameters. The other

layers have learnable parameters that are adjusted as training occurs. The layer types are all hyperparameters; Keras will not change a convolution layer to a max-pooling layer. However, the layers that have parameters are trained/adjusted by the training algorithm. Most of the parameters seen above are the weights of the neural network.

The programmer can configure some of the parameters as non-trainable. The training algorithm cannot adjust these. When we later use transfer learning with this model, we will strip off the final layers that classify 1000 items and replace them with our four media types. Only our new layers will be trainable; we will mark the existing layers as non-trainable.

This neural network makes extensive use of the Relu activation function. Relu is a common choice for activation functions. Also, the neural network makes use of batch and dropout normalization. Many deep neural networks are pyramid-shaped, and this is the case for this one. This neural network uses and expanding pyramid shape as you can see the neuron/filter counts grow from 32 to 64 to 128 to 256 to 512 and max out at 1,024.

We will now use the MobileNet to classify several image URL's below. You can add additional URL's of your own to see how well the MobileNet can classify.

Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML
from tensorflow.keras.applications.mobilenet import decode_predictions

IMAGE_WIDTH = 224
IMAGE_HEIGHT = 224
IMAGE_CHANNELS = 3

ROOT = "https://data.heatonresearch.com/data/t81-558/images/"

def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
        img = img.crop((pad,0,cols,cols))
    else:
        pad = (cols-rows)/2
        img = img.crop((0,pad,rows,rows))

    return img

L = "_____"

def classify_array(images):
    for url in images:
```

```
        x = []
        ImageFile.LOAD_TRUNCATED_IMAGES = False
        response = requests.get(url)
        img = Image.open(BytesIO(response.content))
        img.load()
        img = img.resize((IMAGE_WIDTH,IMAGE_HEIGHT),Image.ANTIALIAS)

        x = image.img_to_array(img)
        x = np.expand_dims(x, axis=0)
        x = preprocess_input(x)
        x = x[:,:,:,:3] # maybe an alpha channel
        pred = model.predict(x)

        display(L)
        display(img)
        print(np.argmax(pred,axis=1))

        lst = decode_predictions(pred, top=5)
        for itm in lst[0]:
            print(itm)

classify_array( [
    ROOT+"soccer_ball.jpg",
    ROOT+"race_truck.jpg"
])
```

Output

```
'------------------------------------------------------------'
```

```
[805]
('n04254680', 'soccer_ball', 0.9999938)
('n03530642', 'honeycomb', 3.862427e-06)
('n03255030', 'dumbbell', 4.4424414e-07)
('n02782093', 'balloon', 3.7039203e-07)
('n04548280', 'wall_clock', 3.143872e-07)
'                                                                    '
```

```
[751]
('n04037443', 'racer', 0.71319467)
('n03100240', 'convertible', 0.1008973)
('n04285008', 'sports_car', 0.07707668)
('n03930630', 'pickup', 0.026352933)
('n02704792', 'amphibian', 0.011636195)
```

Overall, the neural network is doing quite well.

For many applications, MobileNet might be entirely acceptable as an image classifier. However, if you need to classify very specialized images not in the 1,000 image types supported by imagenet, it is necessary to use transfer learning.

9.3.1 Transfer

It is possible to create your image classification network from scratch. This endeavor would take considerable time and resources. Just creating an image classifier would require many labeled pictures. Using a pretrained neural network, you are tapping into knowledge already built into the lower layers of the neural network. The transferred layers likely already have some notion of eyes, ears, feet, and fur. These lower-level concepts help to train the neural network to identify these images.

Next, we reload the MobileNet; however, we set the *include_top* parameter to *False*. This setting instructs Keras not to load the final classification layers. This setting is the common mode of operation for transfer learning. We display a summary to see that the top classification layer is now missing.

Code

```
base_model=MobileNet(weights='imagenet',include_top=False)
#imports the mobilenet model and discards the last 1000 neuron layer.

base_model.summary()
```

Output

```
WARNING:tensorflow:`input_shape` is undefined or non-square, or `rows`
is not in [128, 160, 192, 224]. Weights for input shape (224, 224)
will be loaded as the default.
Model: "mobilenet_1.00_224"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, None, None, 3)]   0
_________________________________________________________________
conv1_pad (ZeroPadding2D)    (None, None, None, 3)     0
_________________________________________________________________
conv1 (Conv2D)               (None, None, None, 32)    864
_________________________________________________________________
conv1_bn (BatchNormalization (None, None, None, 32)    128
_________________________________________________________________

...

=================================================================
Total params: 3,228,864
Trainable params: 3,206,976
Non-trainable params: 21,888
_________________________________________________________________
```

We will add new top layers to the neural network. Our final SoftMax layer includes support for 3 classes.

Code

```
x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x)
x=Dense(1024,activation='relu')(x)
preds=Dense(4,activation='softmax')(x)
```

Next, we mark the original MobileNet layers as non-trainable and our new layers as trainable.

Code

```
model=Model(inputs=base_model.input,outputs=preds)
```

```
for layer in model.layers[:20]:
    layer.trainable=False
for layer in model.layers[20:]:
    layer.trainable=True
```

To train the neural network, we must create a directory structure to hold the images. The Keras command **flow_from_directory** performs this for us. It requires that a folder be laid out as follows. Each class is a folder that contains images of that class. We can also specify a target size; in this case the original MobileNet size of 224x224 is desired.

For this simple example I included four classes, my directories are setup as follows:

- **trans** - The root directory of the dataset.
- **trans/cd** - Pictures of CD's.
- **trans/disk35** - Pictures of 3.5 inch disks.
- **trans/disk525** - Pictures of 5.25 inch disks.
- **trans/tape** - Pictures of tapes.

Code

```
if COLAB:
  PATH = "/content/drive/My Drive/projects/trans/"
else:
  PATH = 'c:\\jth\\data\\trans'

train_datagen=ImageDataGenerator(preprocessing_function=preprocess_input)

train_generator=train_datagen.flow_from_directory(PATH,
                                                  target_size=(224,224),
                                                  color_mode='rgb',
                                                  batch_size=52,
                                                  class_mode='categorical',
                                                  shuffle=True)
```

Output

```
Found 52 images belonging to 4 classes.
```

We are now ready to compile and fit the neural network.

Code

```
model.compile(optimizer='Adam',loss='categorical_crossentropy',
              metrics=['accuracy'])

step_size_train=train_generator.n//train_generator.batch_size
model.fit(train_generator,
                  steps_per_epoch=step_size_train,
                  epochs=50)
```

Output

```
Epoch 1/50
1/1 [==============================] - 0s 2ms/step - loss: 1.6215 -
accuracy: 0.2885
Epoch 2/50
1/1 [==============================] - 0s 1ms/step - loss: 2.0469 -
accuracy: 0.4808
Epoch 3/50
1/1 [==============================] - 0s 1ms/step - loss: 0.0366 -
accuracy: 1.0000
Epoch 4/50
1/1 [==============================] - 0s 1ms/step - loss: 0.9929 -
accuracy: 0.6923
Epoch 5/50
1/1 [==============================] - 0s 2ms/step - loss: 0.0829 -
accuracy: 0.9808

...

- accuracy: 1.0000
Epoch 50/50
1/1 [==============================] - 0s 1ms/step - loss: 1.9898e-06
- accuracy: 1.0000
<tensorflow.python.keras.callbacks.History at 0x7fbd3e91c7f0>
```

To make use of this neural network we will need to know which output neuron corrisponds to each of the training classes/directories we provided to the generator. By calling the **class_indices** property of the generator, we are provided with this information.

Code

```
print(train_generator.class_indices)
```

Output

```
{'cd': 0, 'disk35': 1, 'disk525': 2, 'tape': 3}
```

We are now ready to see how our new model can predict our classes. The URLs in the code some examples. Feel free to add your own. We did not use a large dataset, so it will not be perfect. A larger training set will improve accuracy.

Code

```
%matplotlib inline

def classify_array(images, classes):
    inv_map = {v: k for k, v in classes.items()}
```

```
    for url in images:
        x = []
        ImageFile.LOAD_TRUNCATED_IMAGES = False
        response = requests.get(url)
        img = Image.open(BytesIO(response.content))
        img.load()
        img = img.resize((IMAGE_WIDTH,IMAGE_HEIGHT),Image.ANTIALIAS)

        x = image.img_to_array(img)
        x = np.expand_dims(x, axis=0)
        x = preprocess_input(x)
        x = x[:,:,:,:3] # maybe an alpha channel


        pred = model.predict(x)

        display(L)
        display(img)
        pred2 = int(np.argmax(pred,axis=1))
        print(pred)
        print(inv_map[pred2])
        #print(classes[pred2])
        #print(pred[0])


classify_array( [
    #ROOT+"disk_35.png",
    #ROOT+"disk_525.png",
    #ROOT+"disk_35b.png",
    #ROOT+"IMG_1563.jpg",
    #ROOT+"IMG_1565.jpg",
    ROOT+"IMG_1567.jpg",
    ROOT+"IMG_1570.jpg"
],train_generator.class_indices)
```

Output

```
'------------------------------------------------------------'
```

```
[[3.0885078e-25 1.0000000e+00 4.4679350e-28 0.0000000e+00]]
disk35
'                                                          '
```

```
[[1.8025676e-27 1.2575651e-34 1.0000000e+00 8.7701346e-38]]
disk525
```

This is a very simple example, with a small dataset. The accuracy will not be great; however, it demonstrates how to expand a pretrained model with your own images.

9.4 Part 9.4: Transfer Learning for Languages and Keras

You will commonly use transfer learning in conjunction with Natural Language Processing (NLP). This course has an entire module that covers NLP. However, we will look at how you can load a network into Keras for NLP via transfer learning. The following three sources were helpful for the creation of this section.

- Universal sentence encoder[4]. arXiv preprint arXiv:1803.11175)
- Deep Transfer Learning for Natural Language Processing: Text Classification with Universal Embeddings[19]
- Keras Tutorial: How to Use Google's Universal Sentence Encoder for Spam Classification

These examples make use of TensorFlow Hub, which allows pretrained models to be loaded into TensorFlow easily. To install TensorHub use the following commands.

Code

```
!pip install tensorflow_hub
```

Output

```
Collecting tensorflow_hub
  Downloading tensorflow_hub-0.9.0-py2.py3-none-any.whl (103 kB)
Requirement already satisfied: six>=1.12.0 in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
tensorflow_hub) (1.15.0)
Requirement already satisfied: protobuf>=3.8.0 in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
tensorflow_hub) (3.12.3)
Requirement already satisfied: numpy>=1.12.0 in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
tensorflow_hub) (1.18.5)
Requirement already satisfied: setuptools in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
protobuf>=3.8.0->tensorflow_hub) (49.2.0.post20200714)
Installing collected packages: tensorflow-hub
Successfully installed tensorflow-hub-0.9.0
```

It is also necessary to install TensorFlow Datasets, which you can install \with the following command.

Code

```
!pip install tensorflow_datasets
```

Output

```
Collecting tensorflow_datasets
  Downloading tensorflow_datasets-3.2.1-py3-none-any.whl (3.4 MB)
Requirement already satisfied: requests>=2.19.0 in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
tensorflow_datasets) (2.24.0)
```

```
Requirement already satisfied: wrapt in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
tensorflow_datasets) (1.12.1)
Collecting dill
  Downloading dill-0.3.2.zip (177 kB)
Requirement already satisfied: attrs>=18.1.0 in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from
tensorflow_datasets) (19.3.0)
Requirement already satisfied: absl-py in
c:\users\jeffh\miniconda3\envs\tensorflow\lib\site-packages (from

...

Successfully built dill promise
Installing collected packages: dill, promise, googleapis-common-
protos, tensorflow-metadata, tensorflow-datasets
Successfully installed dill-0.3.2 googleapis-common-protos-1.52.0
promise-2.3 tensorflow-datasets-3.2.1 tensorflow-metadata-0.23.0
```

Load the Internet Movie DataBase (IMDB) reviews data set. This example is based on a TensorFlow example that you can find here.

Code

```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds

train_data, test_data = tfds.load(name="imdb_reviews",
                                  split=["train", "test"],
                                  batch_size=-1, as_supervised=True)

train_examples, train_labels = tfds.as_numpy(train_data)
test_examples, test_labels = tfds.as_numpy(test_data)

# /Users/jheaton/tensorflow_datasets/imdb_reviews/plain_text/0.1.0
```

Output

```
Downloading and preparing dataset imdb_reviews/plain_text/1.0.0
(download: Unknown size, generated: Unknown size, total: Unknown size)
to C:\Users\jeffh\tensorflow_datasets\imdb_reviews\plain_text\1.0.0...
HBox(children=(FloatProgress(value=1.0, bar_style='info',
description='Dl Completed...', max=1.0,
style=ProgreHBox(children=(FloatProgress(value=1.0, bar_style='info',
description='Dl Size...', max=1.0, style=ProgressSty
HBox(children=(FloatProgress(value=1.0, bar_style='info', max=1.0),
HTML(value='')))Shuffling and writing examples to C:\Users\jeffh\tenso
```

```
rflow_datasets\imdb_reviews\plain_text\1.0.0.incomplete7JB40W\imdb_rev
iews-train.tfrecord
HBox(children=(FloatProgress(value=0.0, max=25000.0),
HTML(value='')))HBox(children=(FloatProgress(value=1.0,
bar_style='info', max=1.0), HTML(value='')))Shuffling and writing
examples to C:\Users\jeffh\tensorflow_datasets\imdb_reviews\plain_text

...

\1.0.0.incomplete7JB40W\imdb_reviews-unsupervised.tfrecord
HBox(children=(FloatProgress(value=0.0, max=50000.0),
HTML(value='')))Dataset imdb_reviews downloaded and prepared to
C:\Users\jeffh\tensorflow_datasets\imdb_reviews\plain_text\1.0.0.
Subsequent calls will reuse this data.
```

Load a pretrained embedding model called gnews-swivel-20dim. This network was trained by Google on GNEWS data and can convert RAW text into vectors.

Code

```
model = "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1"
hub_layer = hub.KerasLayer(model, output_shape=[20], input_shape=[],
                           dtype=tf.string, trainable=True)
```

Consider the following three movie reviews.

Code

```
train_examples[:3]
```

Output

```
array([b"This was an absolutely terrible movie. Don't be lured in by
Christopher Walken or Michael Ironside. Both are great actors, but
this must simply be their worst role in history. Even their great
acting could not redeem this movie's ridiculous storyline. This movie
is an early nineties US propaganda piece. The most pathetic scenes
were those when the Columbian rebels were making their cases for
revolutions. Maria Conchita Alonso appeared phony, and her pseudo-love
affair with Walken was nothing but a pathetic emotional plug in a
movie that was devoid of any real meaning. I am disappointed that
there are movies like this, ruining actor's like Christopher Walken's
good name. I could barely sit through it.",
       b'I have been known to fall asleep during films, but this is
usually due to a combination of things including, really tired, being
warm and comfortable on the sette and having just eaten a lot. However
on this occasion I fell asleep because the film was rubbish. The plot

...
```

```
rush. Mr. Mann and company appear to have mistaken Dawson City for
Deadwood, the Canadian North for the American Wild West.<br /><br
/>Canadian viewers be prepared for a Reefer Madness type of enjoyable
howl with this ludicrous plot, or, to shake your head in disgust.'],
      dtype=object)
```

The embedding layer can convert each to 20-number vectors.

Code

```
hub_layer(train_examples[:3])
```

Output

```
<tf.Tensor: shape=(3, 20), dtype=float32, numpy=
array([[ 1.765786  , -3.882232  ,  3.9134233 , -1.5557289 , -3.3362343
,
        -1.7357955 , -1.9954445 ,  1.2989551 ,  5.081598  , -1.1041286
,
        -2.0503852 , -0.72675157, -0.65675956,  0.24436149, -3.7208383
,
         2.0954835 ,  2.2969332 , -2.0689783 , -2.9489717 , -1.1315987
],
       [ 1.8804485 , -2.5852382 ,  3.4066997 ,  1.0982676 , -4.056685
,
        -4.891284  , -2.785554  ,  1.3874227 ,  3.8476458 , -0.9256538
,
        -1.896706  ,  1.2113281 ,  0.11474707,  0.76209456, -4.8791065
,

...

        -2.2268345 ,  0.07446612, -1.4075904 , -0.70645386, -1.907037
,
         1.4419787 ,  1.9551861 , -0.42660055, -2.8022065 ,
0.43727064]],
      dtype=float32)>
```

We add additional layers to attempt to classify the movie reviews as either positive or negative.

Code

```
model = tf.keras.Sequential()
model.add(hub_layer)
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

```
model.summary()
```

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 20)                400020
_________________________________________________________________
dense (Dense)                (None, 16)                336
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 400,373
Trainable params: 400,373
Non-trainable params: 0
_________________________________________________________________
```

Compile the neural network.

Code

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Split and train the neural network.

Code

```
x_val = train_examples[:10000]
partial_x_train = train_examples[10000:]

y_val = train_labels[:10000]
partial_y_train = train_labels[10000:]
```

Code

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=40,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=1)
```

Output

```
Train on 15000 samples, validate on 10000 samples
Epoch 1/40
15000/15000 [==============================] - 3s 189us/sample - loss:
0.6388 - accuracy: 0.6433 - val_loss: 0.5910 - val_accuracy: 0.6937
Epoch 2/40
15000/15000 [==============================] - 2s 143us/sample - loss:
0.5626 - accuracy: 0.7191 - val_loss: 0.5495 - val_accuracy: 0.7295
Epoch 3/40
15000/15000 [==============================] - 2s 143us/sample - loss:
0.5173 - accuracy: 0.7573 - val_loss: 0.5138 - val_accuracy: 0.7585
Epoch 4/40
15000/15000 [==============================] - 2s 145us/sample - loss:
0.4774 - accuracy: 0.7839 - val_loss: 0.4809 - val_accuracy: 0.7832
Epoch 5/40
15000/15000 [==============================] - 2s 146us/sample - loss:

...

15000/15000 [==============================] - 2s 144us/sample - loss:
0.0384 - accuracy: 0.9949 - val_loss: 0.3966 - val_accuracy: 0.8707
Epoch 40/40
15000/15000 [==============================] - 2s 145us/sample - loss:
0.0359 - accuracy: 0.9955 - val_loss: 0.4033 - val_accuracy: 0.8684
```

Evaluate the neural network.

Code

```
results = model.evaluate(test_data, test_labels)

print(results)
```

Output

```
25000/25000 [==============================] - 4s 167us/sample - loss:
0.4406 - accuracy: 0.8523
[0.44060500403404235, 0.85228]
```

Code

```
history_dict = history.history
history_dict.keys()
```

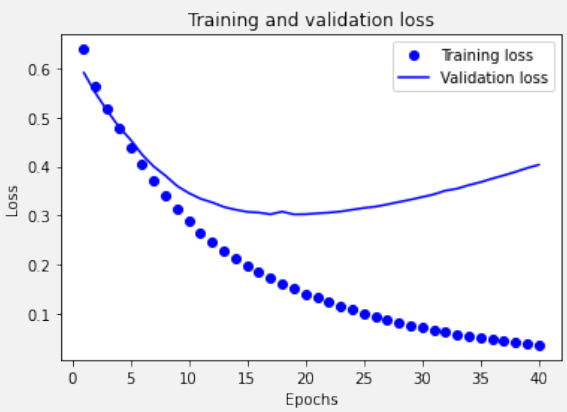
Output

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Code

```
%matplotlib inline
import matplotlib.pyplot as plt

acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Output

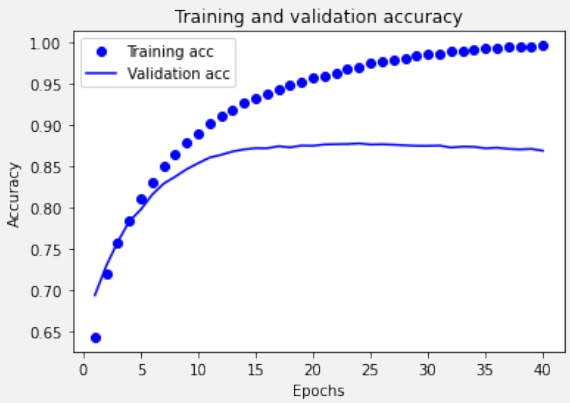


Code

```
plt.clf()   # clear figure

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
```

```
plt.legend()

plt.show()
```

Output



9.5 Part 9.5: Transfer Learning for Keras Feature Engineering

It is also common to use transfer learning to generate features from complex input. We will see this technique in later modules when we make use of Word2Vec. In this section, we will learn that we can take a pretrained neural network and strip off the dense layers. The output from the final convolutional layers becomes vectors of features about the source images. These vectors will contain information about the subcomponents of the images and might be useful as input to other models.

The following code extracts 1,024 sized vectors of features for images.

Code

```
%matplotlib inline
import pandas as pd
import numpy as np
import os
import tensorflow.keras
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense,GlobalAveragePooling2D
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
```

```
from io import BytesIO
from IPython.display import display, HTML
from tensorflow.keras.applications.mobilenet import decode_predictions

model = MobileNet(weights='imagenet',include_top=False)

IMAGE_WIDTH = 224
IMAGE_HEIGHT = 224
IMAGE_CHANNELS = 3

ROOT = "https://data.heatonresearch.com/data/t81-558/images/"

images = [
    ROOT+"soccer_ball.jpg"
]


def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
        img = img.crop((pad,0,cols,cols))
    else:
        pad = (cols-rows)/2
        img = img.crop((0,pad,rows,rows))

    return img

for url in images:
    x = []
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = img.resize((IMAGE_WIDTH,IMAGE_HEIGHT),Image.ANTIALIAS)

    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    pred = model.predict(x)

    display("_____")
    display(img)
    print(pred.shape)
    print(pred)
```

Output

```
'_____'
```

```
(1, 7, 7, 1024)
[[[[0.          0.          0.          ... 0.          0.
    0.        ]
   [0.          0.          0.          ... 0.          0.
    0.        ]
   [0.          0.          0.          ... 0.26740852  0.
    0.        ]
   ...
   [0.          0.          0.          ... 0.          0.
    0.        ]
   [0.          0.          0.          ... 0.          0.
    0.        ]
   [0.          0.          0.          ... 0.          0.
    0.        ]]
  [[0.          0.          0.          ... 0.          0.

...

    0.        ]
   [0.          0.          0.          ... 0.          0.
    0.        ]
   [0.          0.          0.          ... 0.          0.
    0.        ]]]]
```

Code

```
model.summary()
```

Output

```
Model: "mobilenet_1.00_224"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, None, 3)]   0
_________________________________________________________________
conv1_pad (ZeroPadding2D)    (None, None, None, 3)     0
_________________________________________________________________
conv1 (Conv2D)               (None, None, None, 32)    864
_________________________________________________________________
conv1_bn (BatchNormalization (None, None, None, 32)    128
_________________________________________________________________
conv1_relu (ReLU)            (None, None, None, 32)    0
_________________________________________________________________
conv_dw_1 (DepthwiseConv2D)  (None, None, None, 32)    288

...

=================================================================
Total params: 3,228,864
Trainable params: 3,206,976
Non-trainable params: 21,888
_________________________________________________________________
```

Chapter 10

Time Series in Keras

10.1 Part 10.1: Time Series Data Encoding

In this chapter, we will examine time series encoding and recurrent networks, two topics that are logical to put together because they are both methods for dealing with data that spans over time. Time series encoding deals with representing events that occur over time to a neural network. There are many different methods to encode data that occur over time to a neural network. This encoding is necessary because a feedforward neural network will always produce the same output vector for a given input vector. Recurrent neural networks do not require encoding of time series data because they are able to handle data that occur over time automatically.

The variation in temperature during the week is an example of time-series data. For instance, if we know that today's temperature is 25 degrees, and tomorrow's temperature is 27 degrees, the recurrent neural networks and time series encoding provide another option to predict the correct temperature for the week. Conversely, a traditional feedforward neural network will always respond with the same output for a given input. If we train a feedforward neural network to predict tomorrow's temperature, it should return a value of 27 for 25. The fact that it will always output 27 when given 25 might be a hindrance to its predictions. Surely the temperature of 27 will not always follow 25. It would be better for the neural network to consider the temperatures for a series of days before the prediction. Perhaps the temperature over the last week might allow us to predict tomorrow's temperature. Therefore, recurrent neural networks and time series encoding represent two different approaches to representing data over time to a neural network.

Previously we trained neural networks with input (x) and expected output (y). X was a matrix, the rows were training examples, and the columns were values to be predicted. The x value will now contain sequences of data. The definition of the y value will stay the same.

Dimensions of the training set (x):

- Axis 1: Training set elements (sequences) (must be of the same size as y size)
- Axis 2: Members of sequence
- Axis 3: Features in data (like input neurons)

Previously, we might take as input a single stock price, to predict if we should buy (1), sell (-1), or hold (0). The following code illustrates this encoding.

Code

```
#
```

```
x = [
    [32],
    [41],
    [39],
    [20],
    [15]
]

y = [
    1,
    -1,
    0,
    -1,
    1
]

print(x)
print(y)
```

Output

```
[[32], [41], [39], [20], [15]]
[1, -1, 0, -1, 1]
```

The following code builds a CSV file from scratch, to see it as a data frame, use the following:

Code

```
from IPython.display import display, HTML
import pandas as pd
import numpy as np

x = np.array(x)
print(x[:,0])


df = pd.DataFrame({'x':x[:,0], 'y':y})
display(df)
```

Output

	x	y
0	32	1
1	41	-1
2	39	0
3	20	-1
4	15	1

```
[32 41 39 20 15]
```

You might want to put volume in with the stock price. The following code shows how we can add an additional dimension to handle the volume.

Code

```
x = [
    [32,1383],
    [41,2928],
    [39,8823],
    [20,1252],
    [15,1532]
]

y = [
    1,
    -1,
    0,
    -1,
    1
]

print(x)
print(y)
```

Output

```
[[32, 1383], [41, 2928], [39, 8823], [20, 1252], [15, 1532]]
[1, -1, 0, -1, 1]
```

Again, very similar to what we did before. The following shows this as a data frame.

Code

```
from IPython.display import display, HTML
import pandas as pd
import numpy as np

x = np.array(x)
print(x[:,0])


df = pd.DataFrame({'price':x[:,0], 'volume':x[:,1], 'y':y})
display(df)
```

Output

	price	volume	y
0	32	1383	1
1	41	2928	-1
2	39	8823	0
3	20	1252	-1
4	15	1532	1

```
[32 41 39 20 15]
```

Now we get to sequence format. We want to predict something over a sequence, so the data format needs to add a dimension. A maximum sequence length must be specified, but the individual sequences can be of any length.

Code

```
x = [
    [[32,1383],[41,2928],[39,8823],[20,1252],[15,1532]],
    [[35,8272],[32,1383],[41,2928],[39,8823],[20,1252]],
    [[37,2738],[35,8272],[32,1383],[41,2928],[39,8823]],
    [[34,2845],[37,2738],[35,8272],[32,1383],[41,2928]],
    [[32,2345],[34,2845],[37,2738],[35,8272],[32,1383]],
]

y = [
    1,
    -1,
    0,
    -1,
    1
]

print(x)
print(y)
```

Output

```
[[[32, 1383], [41, 2928], [39, 8823], [20, 1252], [15, 1532]], [[35,
8272], [32, 1383], [41, 2928], [39, 8823], [20, 1252]], [[37, 2738],
[35, 8272], [32, 1383], [41, 2928], [39, 8823]], [[34, 2845], [37,
2738], [35, 8272], [32, 1383], [41, 2928]], [[32, 2345], [34, 2845],
[37, 2738], [35, 8272], [32, 1383]]]
[1, -1, 0, -1, 1]
```

Even if there is only one feature (price), the 3rd dimension must be used:

Code

```
x = [
    [[32],[41],[39],[20],[15]],
    [[35],[32],[41],[39],[20]],
    [[37],[35],[32],[41],[39]],
    [[34],[37],[35],[32],[41]],
    [[32],[34],[37],[35],[32]],
]

y = [
    1,
    -1,
    0,
    -1,
    1
]

print(x)
print(y)
```

Output

```
[[[32], [41], [39], [20], [15]], [[35], [32], [41], [39], [20]],
[[37], [35], [32], [41], [39]], [[34], [37], [35], [32], [41]], [[32],
[34], [37], [35], [32]]]
[1, -1, 0, -1, 1]
```

10.1.1 Module 10 Assignment

You can find the first assignment here: assignment 10

10.2 Part 10.2: Programming LSTM with Keras and TensorFlow

So far, the neural networks that we've examined have always had forward connections. Neural networks of this type always begin with an input layer connected to the first hidden layer. Each hidden layer always connects to the next hidden layer. The final hidden layer always connects to the output layer. This manner to connect layers is the reason that these networks are called "feedforward." Recurrent neural networks are not so rigid, as backward connections are also allowed. A recurrent connection links a neuron in a layer to either a previous layer or the neuron itself. Most recurrent neural network architectures maintain state in the recurrent connections. Feedforward neural networks don't maintain any state. A recurrent neural network's state acts as a sort of short-term memory for the neural network. Consequently, a recurrent neural network will not always produce the same output for a given input.

Recurrent neural networks do not force the connections to flow only from one layer to the next, from the input layer to the output layer. A recurrent connection occurs when a connection is formed between a neuron and one of the following other types of neurons:

- The neuron itself
- A neuron on the same level
- A neuron on a previous level

Recurrent connections can never target the input neurons or bias neurons. The processing of recurrent connections can be challenging. Because the recurrent links create endless loops, the neural network must have some way to know when to stop. A neural network that entered an endless loop would not be useful. To prevent endless loops, we can calculate the recurrent connections with the following three approaches:

- Context neurons
- Calculating output over a fixed number of iterations
- Calculating output until neuron output stabilizes

The context neuron is a special neuron type that remembers its input and provides that input as its output the next time that we calculate the network. For example, if we gave a context neuron 0.5 as input, it would output 0. Context neurons always output 0 on their first call. However, if we gave the context neuron a 0.6 as input, the output would be 0.5. We never weigh the input connections to a context neuron, but we can weigh the output from a context neuron just like any other network connection.

Context neurons allow us to calculate a neural network in a single feedforward pass. Context neurons usually occur in layers. A layer of context neurons will always have the same number of context neurons as neurons in its source layer, as demonstrated by Figure 10.1.

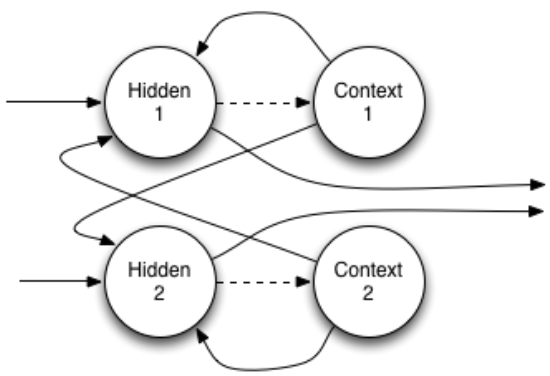


Figure 10.1: Context Layers

As you can see from the above layer, two hidden neurons that are labeled hidden one and hidden two directly connect to the two context neurons. The dashed lines on these connections indicate that these are not weighted connections. These weightless connections are never dense. If these connections were dense, hidden one would be connected to both hidden one and hidden 2. However, the direct connection joins each hidden neuron to its corresponding context neuron. The two context neurons form dense, weighted connections to the two hidden neurons. Finally, the two hidden neurons also form dense connections to the neurons in the next layer. The two context neurons form two connections to a single neuron in the next layer, four connections to two neurons, six connections to three neurons, and so on.

You can combine context neurons with the input, hidden, and output layers of a neural network in many different ways.

10.2.1 Understanding LSTM

Long Short Term Neural Network (LSTM) layers are a type of recurrent unit that you often use with deep neural networks.[18]For TensorFlow, you can think of LSTM as a layer type that you can combine with other layer types, such as dense. LSTM makes use of two transfer function types internally.

The first type of transfer function is the sigmoid. This transfer function type is used form gates inside of the unit. The sigmoid transfer function is given by the following equation:

$S(t) = \frac{1}{1+e^{-t}}$

The second type of transfer function is the hyperbolic tangent (tanh) function, which you to scale the output of the LSTM. This functionality is similar to how we have used other transfer functions in this course.

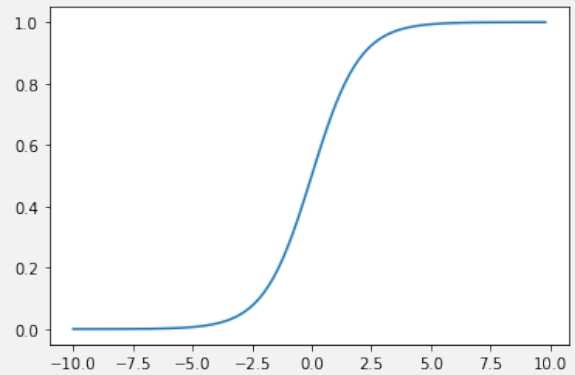
We provide the graphs for these functions here:

Code

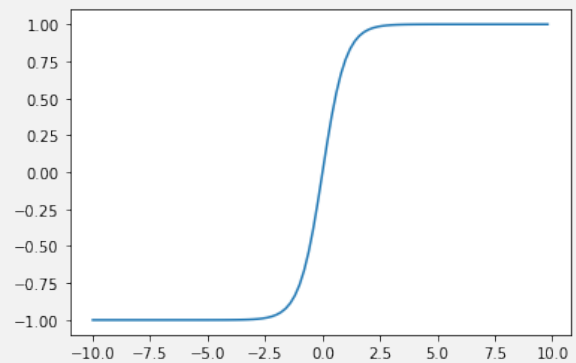
```
%matplotlib inline

import matplotlib
import numpy as np
import matplotlib.pyplot as plt
import math

def sigmoid(x):
    a = []
    for item in x:
        a.append(1/(1+math.exp(-item)))
    return a

def f2(x):
    a = []
    for item in x:
        a.append(math.tanh(item))
    return a

x = np.arange(-10., 10., 0.2)
y1 = sigmoid(x)
y2 = f2(x)

print("Sigmoid")
plt.plot(x,y1)
plt.show()

print("Hyperbolic Tangent(tanh)")
plt.plot(x,y2)
plt.show()
```

Output

```
Sigmoid
```



Hyperbolic Tangent(tanh)



Both of these two functions compress their output to a specific range. For the sigmoid function, this range is 0 to 1. For the hyperbolic tangent function, this range is -1 to 1.

LSTM maintains an internal state and produces an output. The following diagram shows an LSTM unit over three time slices: the current time slice (t), as well as the previous (t-1) and next (t+1) slice, as demonstrated by Figure 10.2.

The values $\hat{y}$ are the output from the unit; the values (x) are the input to the unit, and the values c are the context values. The output and context values always feed their output to the next time slice. The context values allow the network to maintain state between calls. Figure 10.3 shows the internals of a LSTM layer.

A LSTM unit consists of three gates:

- Forget Gate (f_t) - Controls if/when the context is forgotten. (MC)
- Input Gate (i_t) - Controls if/when the context should remember a value. (M+/MS)
- Output Gate (o_t) - Controls if/when the remembered value is allowed to pass from the unit. (RM)

Mathematically, you can think of the above diagram as the following:

These are vector values.

First, calculate the forget gate value. This gate determines if the LSTM unit should forget its short term memory. The value b is a bias, just like the bias neurons we saw before. Except LSTM has a bias for every gate: b_t, b_i, and b_o.

$$f_t = S(W_f \cdot [\hat{y}_{t-1}, x_t] + b_f)$$

$$i_t = S(W_i \cdot [\hat{y}_{t-1}, x_t] + b_i)$$

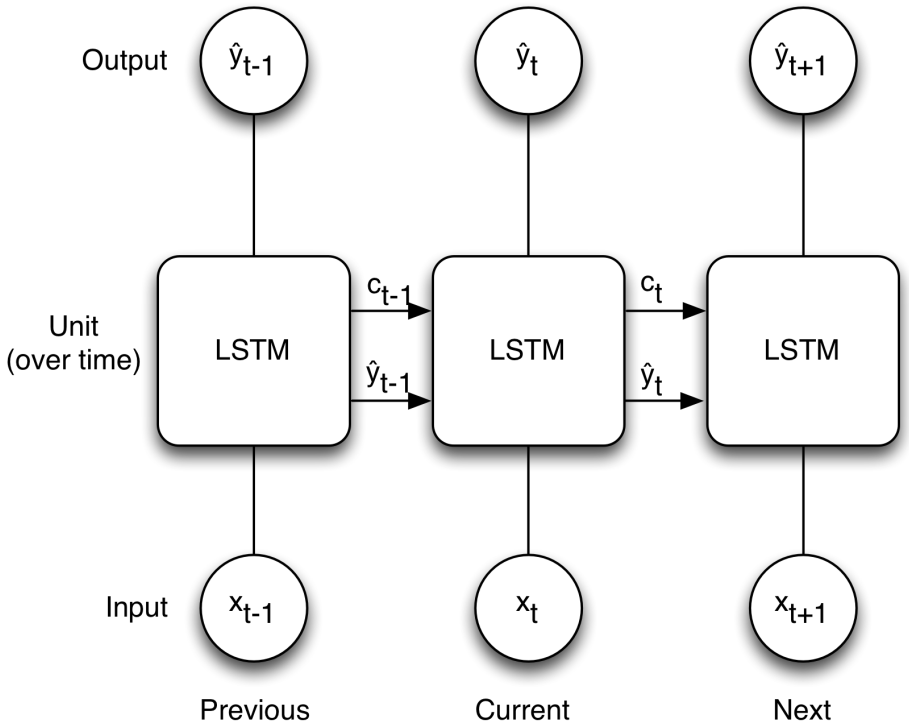


Figure 10.2: LSTM Layers

$\tilde{C}t = \backslash tanh(W_C \backslash cdot [\backslash hat\{y\}t - 1, x_t] + b_C)$

$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$

$o_t = S(W_o \cdot [\hat{y}_{t-1}, x_t] + b_o)$

$\hat{y}_t = o_t \cdot \tanh(C_t)$

10.2.2 Simple TensorFlow LSTM Example

The following code creates the LSTM network, which is an example of a RNN for classification. The following code trains on a data set (x) with a max sequence size of 6 (columns) and six training elements (rows)

Code

```
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding
from tensorflow.keras.layers import LSTM
import numpy as np
```

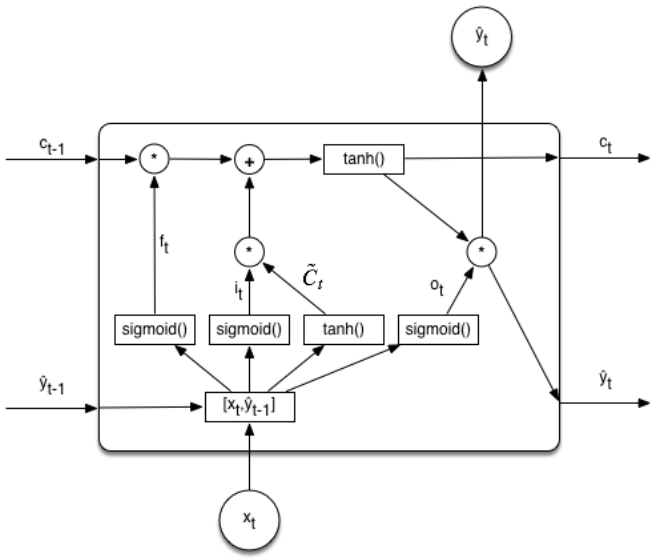


Figure 10.3: Inside a LSTM Layer

```
max_features = 4 # 0,1,2,3 (total of 4)
x = [
    [[0],[1],[1],[0],[0],[0]],
    [[0],[0],[0],[2],[2],[0]],
    [[0],[0],[0],[0],[3],[3]],
    [[0],[2],[2],[0],[0],[0]],
    [[0],[0],[3],[3],[0],[0]],
    [[0],[0],[0],[0],[1],[1]]
]
x = np.array(x,dtype=np.float32)
y = np.array([1,2,3,2,3,1],dtype=np.int32)

# Convert y2 to dummy variables
y2 = np.zeros((y.shape[0], max_features),dtype=np.float32)
```

```
y2[np.arange(y.shape[0]), y] = 1.0
print(y2)

print('Build␣model...')
model = Sequential()
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2, \
               input_shape=(None, 1)))
model.add(Dense(4, activation='sigmoid'))

# try using different optimizers and different optimizer configs
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print('Train...')
model.fit(x,y2,epochs=200)
pred = model.predict(x)
predict_classes = np.argmax(pred,axis=1)
print("Predicted␣classes:␣{}",predict_classes)
print("Expected␣classes:␣{}",predict_classes)
```

Output

```
[[0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]]
Build model...
Train...
Train on 6 samples
Epoch 1/200
6/6 [==============================] - 3s 505ms/sample - loss: 0.6913
- accuracy: 0.5417
Epoch 2/200
6/6 [==============================] - 0s 2ms/sample - loss: 0.6886 -
accuracy: 0.5833

...

Epoch 200/200
6/6 [==============================] - 0s 4ms/sample - loss: 0.2199 -
accuracy: 0.9167
Predicted classes: {} [1 2 3 2 2 1]
Expected classes: {} [1 2 3 2 2 1]
```

Code

```
def runit(model, inp):
    inp = np.array(inp,dtype=np.float32)
    pred = model.predict(inp)
    return np.argmax(pred[0])

print( runit( model, [[[0],[0],[0],[0],[0],[1]]] ))
```

Output

```
1
```

10.2.3 Sun Spots Example

In this section, we see an example of RNN regression to predict sunspots. You can find the data files needed for this example at the following location.

- Sunspot Data Files
- Download Daily Sunspots - 1/1/1818 to now.

The following code loads the sunspot file:

Code

```
import pandas as pd
import os

# Replacce the following path with your own file. It can be downloaded from:
# http://www.sidc.be/silso/INFO/sndtotcsv.php

if COLAB:
    PATH = "/content/drive/My Drive/Colab Notebooks/data/"
else:
    PATH = "./data/"

filename = os.path.join(PATH,"SN_d_tot_V2.0.csv")
names = ['year', 'month', 'day', 'dec_year', 'sn_value' ,
         'sn_error', 'obs_num']
df = pd.read_csv(filename,sep=';',header=None,names=names,
                 na_values=['-1'], index_col=False)

print("Starting file:")
print(df[0:10])

print("Ending file:")
print(df[-10:])
```

Output

```
Starting file:
    year  month  day  dec_year  sn_value  sn_error  obs_num
0   1818      1    1  1818.001        -1       NaN        0
1   1818      1    2  1818.004        -1       NaN        0
2   1818      1    3  1818.007        -1       NaN        0
3   1818      1    4  1818.010        -1       NaN        0
4   1818      1    5  1818.012        -1       NaN        0
5   1818      1    6  1818.015        -1       NaN        0
6   1818      1    7  1818.018        -1       NaN        0
7   1818      1    8  1818.021        65      10.2        1
8   1818      1    9  1818.023        -1       NaN        0
9   1818      1   10  1818.026        -1       NaN        0
Ending file:
       year  month  day  dec_year  sn_value  sn_error  obs_num
73769  2019     12   22  2019.974         0       0.0       17

...

73774  2019     12   27  2019.988         0       0.0       26
73775  2019     12   28  2019.990         0       0.0       26
73776  2019     12   29  2019.993         0       0.0       27
73777  2019     12   30  2019.996         0       0.0       32
73778  2019     12   31  2019.999         0       0.0       19
```

As you can see, there is quite a bit of missing data near the end of the file. We want to find the starting index where the missing data no longer occurs. This technique is somewhat sloppy; it would be better to find a use for the data between missing values. However, the point of this example is to show how to use LSTM with a somewhat simple time-series.

Code

```
start_id = max(df[df['obs_num'] == 0].index.tolist())+1
# Find the last zero and move one beyond
print(start_id)
df = df[start_id:] # Trim the rows that have missing observations
```

Output

```
11314
```

Code

```
df['sn_value'] = df['sn_value'].astype(float)
df_train = df[df['year']<2000]
df_test = df[df['year']>=2000]
```

```
spots_train = df_train['sn_value'].tolist()
spots_test = df_test['sn_value'].tolist()

print("Training set has {} observations.".format(len(spots_train)))
print("Test set has {} observations.".format(len(spots_test)))
```

Output

```
Training set has 55160 observations.
Test set has 7305 observations.
```

Code

```
import numpy as np

def to_sequences(seq_size, obs):
    x = []
    y = []

    for i in range(len(obs)-SEQUENCE_SIZE):
        #print(i)
        window = obs[i:(i+SEQUENCE_SIZE)]
        after_window = obs[i+SEQUENCE_SIZE]
        window = [[x] for x in window]
        #print("{} - {}".format(window,after_window))
        x.append(window)
        y.append(after_window)

    return np.array(x),np.array(y)


SEQUENCE_SIZE = 10
x_train,y_train = to_sequences(SEQUENCE_SIZE,spots_train)
x_test,y_test = to_sequences(SEQUENCE_SIZE,spots_test)

print("Shape of training set: {}".format(x_train.shape))
print("Shape of test set: {}".format(x_test.shape))
```

Output

```
Shape of training set: (55150, 10, 1)
Shape of test set: (7295, 10, 1)
```

Code

```
x_train
```

Output

```
array([[[353.],
        [240.],
        [275.],
        ...,
        [340.],
        [238.],
        [287.]],
       [[240.],
        [275.],
        [352.],
        ...,
        [238.],
        [287.],
        [294.]],
       [[275.],

...

        [123.],
        ...,
        [ 85.],
        [103.],
        [ 66.]]])
```

Code

```
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding
from tensorflow.keras.layers import LSTM
from tensorflow.keras.datasets import imdb
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np

print('Build model...')
model = Sequential()
model.add(LSTM(64, dropout=0.0, recurrent_dropout=0.0,input_shape=(None, 1)))
model.add(Dense(32))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
```

```
                        verbose=1, mode='auto', restore_best_weights=True)
print('Train...')

model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

Output

```
Build model...
Train...
Train on 55150 samples, validate on 7295 samples
Epoch 1/1000
55150/55150 - 13s - loss: 1312.6864 - val_loss: 190.2033
Epoch 2/1000
55150/55150 - 8s - loss: 513.1618 - val_loss: 188.5868
Epoch 3/1000
55150/55150 - 8s - loss: 510.8469 - val_loss: 191.0815
Epoch 4/1000
55150/55150 - 8s - loss: 506.8735 - val_loss: 215.0268
Epoch 5/1000
55150/55150 - 8s - loss: 503.7439 - val_loss: 193.7987
Epoch 6/1000
55150/55150 - 8s - loss: 504.5192 - val_loss: 199.2520
Epoch 7/1000
Restoring model weights from the end of the best epoch.
55150/55150 - 8s - loss: 502.6547 - val_loss: 198.9333
Epoch 00007: early stopping
<tensorflow.python.keras.callbacks.History at 0x2583b832548>
```

Finally, we evaluate the model with RMSE.

Code

```
from sklearn import metrics

pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print("Score (RMSE): {}".format(score))
```

Output

```
Score (RMSE): 13.732691339581104
```

10.2.4 Further Reading for LSTM

- Understanding LSTM Networks

- Recurrent Neural Networks in TensorFlow

10.3 Part 10.3: Text Generation with LSTM

Recurrent neural networks are also known for their ability to generate text. As a result, the output of the neural network can be free-form text. In this section, we will see how to train an LSTM can on a textual document, such as classic literature, and learn to output new text that appears to be of the same form as the training material. If you train your LSTM on Shakespeare, it will learn to crank out new prose similar to what Shakespeare had written.

Don't get your hopes up. You are not going to teach your deep neural network to write the next Pulitzer Prize for Fiction. The prose generated by your neural network will be nonsensical. However, it will usually be nearly grammatically and of a similar style as the source training documents.

A neural network generating nonsensical text based on literature may not seem useful at first glance. However, this technology gets so much interest because it forms the foundation for many more advanced technologies. The fact that the LSTM will typically learn human grammar from the source document opens a wide range of possibilities. You can use similar technology to complete sentences when a user is entering text. Simply the ability to output free-form text becomes the foundation of many other technologies. In the next part, we will use this technique to create a neural network that can write captions for images to describe what is going on in the picture.

10.3.1 Additional Information

The following are some of the articles that I found useful in putting this section together.

- The Unreasonable Effectiveness of Recurrent Neural Networks
- Keras LSTM Generation Example

10.3.2 Character-Level Text Generation

There are several different approaches to teaching a neural network to output free-form text. The most basic question is if you wish the neural network to learn at the word or character level. In many ways, learning at the character level is the more interesting of the two. The LSTM is learning to construct its own words without even being shown what a word is. We will begin with character-level text generation. In the next module, we will see how we can use nearly the same technique to operate at the word level. We will implement word-level automatic captioning in the next module.

We begin by importing the needed Python packages and defining the sequence length, named **maxlen**. Time-series neural networks always accept their input as a fixed-length array. Because you might not use all of the sequence elements, it is common to fill extra elements with zeros. You will divide the text into sequences of this length, and the neural network will train to predict what comes after this sequence.

Code

```
from tensorflow.keras.callbacks import LambdaCallback
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.utils import get_file
import numpy as np
import random
```

```
import sys
import io
import requests
import re
```

For this simple example, we will train the neural network on the classic children's book Treasure Island. We begin by loading this text into a Python string and displaying the first 1,000 characters.

Code

```
r = requests.get("https://data.heatonresearch.com/data/t81-558/text/"\
                 "treasure_island.txt")
raw_text = r.text
print(raw_text[0:1000])
```

Output

```
The Project Gutenberg EBook of Treasure Island, by Robert Louis
Stevenson
This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.net
Title: Treasure Island
Author: Robert Louis Stevenson
Illustrator: Milo Winter
Release Date: January 12, 2009 [EBook #27780]
Language: English
*** START OF THIS PROJECT GUTENBERG EBOOK TREASURE ISLAND ***
Produced by Juliet Sutherland, Stephen Blundell and the
Online Distributed Proofreading Team at http://www.pgdp.net
 THE ILLUSTRATED CHILDREN'S LIBRARY

...

             Milo Winter
            [Illustration]
           GRAMERCY BOOKS
              NEW YORK
 Foreword copyright  1986 by Random House V
```

We will extract all unique characters from the text and sort them. This technique allows us to assign a unique ID to each character. Because we sorted the characters, these IDs should remain the same. If we add new characters to the original text, then the IDs would change. We build two dictionaries. The first **char2idx** is used to convert a character into its ID. The second **idx2char** converts an ID back into its character.

Code

```
processed_text = raw_text.lower()
processed_text = re.sub(r'[^\x00-\x7f]',r'', processed_text)
```

Code

```
print('corpus length:', len(processed_text))

chars = sorted(list(set(processed_text)))
print('total chars:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

Output

```
corpus length: 397400
total chars: 60
```

We are now ready to build the actual sequences. Just like previous neural networks, there will be an x and y. However, for the LSTM, x and y will both be sequences. The x input will specify the sequences where y are the expected output. The following code generates all possible sequences.

Code

```
# cut the text in semi-redundant sequences of maxlen characters
maxlen = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(processed_text) - maxlen, step):
    sentences.append(processed_text[i: i + maxlen])
    next_chars.append(processed_text[i + maxlen])
print('nb sequences:', len(sentences))
```

Output

```
nb sequences: 132454
```

Code

```
sentences
```

Output

```
['the project gutenberg ebook of treasure ',
```

```
 ' project gutenberg ebook of treasure isl',
 'oject gutenberg ebook of treasure island',
 'ct gutenberg ebook of treasure island, b',
 'gutenberg ebook of treasure island, by r',
 'enberg ebook of treasure island, by robe',
 'erg ebook of treasure island, by robert ',
 ' ebook of treasure island, by robert lou',
 'ook of treasure island, by robert louis ',
 ' of treasure island, by robert louis ste',
 ' treasure island, by robert louis steven',
 'easure island, by robert louis stevenson',
 'ure island, by robert louis stevenson\r\n\r',
 ' island, by robert louis stevenson\r\n\r\nth',
 'land, by robert louis stevenson\r\n\r\nthis ',

...

 'st of color plates_                       ',
 'of color plates_                          ',
 'color plates_                             ',
 'or plates_                                ',
 ...]
```

Code

```
print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1
```

Output

```
Vectorization...
```

Code

```
x.shape
```

Output

```
(132454, 40, 60)
```

Code

```
y.shape
```

Output

```
(132454, 60)
```

The dummy variables for y are shown below.

Code

```
y[0:10]
```

Output

```
array([[False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False,  True, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False,  True, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False,

...

        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False,  True, False, False, False, False]])
```

Next, we create the neural network. This neural network's primary feature is the LSTM layer, which allows the sequences to be processed.

Code

```
# build the model: a single LSTM
print('Build model...')
model = Sequential()
```

```
model.add(LSTM(128, input_shape=(maxlen, len(chars))))
model.add(Dense(len(chars), activation='softmax'))

optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Output

```
Build model...
```

Code

```
model.summary()
```

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 128)               96768
_________________________________________________________________
dense (Dense)                (None, 60)                7740
=================================================================
Total params: 104,508
Trainable params: 104,508
Non-trainable params: 0
_________________________________________________________________
```

The LSTM will produce new text character by character. We will need to sample the correct letter from the LSTM predictions each time. The **sample** function accepts the following two parameters:

- **preds** - The output neurons.
- **temperature** - 1.0 is the most conservative, 0.0 is the most confident (willing to make spelling and other errors).

The sample function below is essentially performing a softmax on the neural network predictions. This causes each output neuron to become a probability of its particular letter.

Code

```
def sample(preds, temperature=1.0):
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
```

```
    return np.argmax(probas)
```

Keras calls the following function at the end of each training Epoch. The code generates sample text generations that visually demonstrate the neural network better at text generation. As the neural network trains, the generations should look more realistic.

Code

```
def on_epoch_end(epoch, _):
    # Function invoked at end of each epoch. Prints generated text.
    print("******************************************************")
    print('----- Generating text after Epoch: %d' % epoch)

    start_index = random.randint(0, len(processed_text) - maxlen - 1)
    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('----- temperature:', temperature)

        generated = ''
        sentence = processed_text[start_index: start_index + maxlen]
        generated += sentence
        print('----- Generating with seed: "' + sentence + '"')
        sys.stdout.write(generated)

        for i in range(400):
            x_pred = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(sentence):
                x_pred[0, t, char_indices[char]] = 1.

            preds = model.predict(x_pred, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = indices_char[next_index]

            generated += next_char
            sentence = sentence[1:] + next_char

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()
```

We are now ready to train. It can take up to an hour to train this network, depending on how fast your computer is. If you have a GPU available, please make sure to use it.

Code

```
# Ignore useless W0819 warnings generated by TensorFlow 2.0.
Hopefully can remove this ignore in the future.
# See https://github.com/tensorflow/tensorflow/issues/31308
import logging, os
logging.disable(logging.WARNING)
```

```
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

# Fit the model
print_callback = LambdaCallback(on_epoch_end=on_epoch_end)

model.fit(x, y,
          batch_size=128,
          epochs=60,
          callbacks=[print_callback])
```

Output

```
Train on 132454 samples
Epoch 1/60
   128/132454 [..............................] - ETA:
35:39**************************************************
------ Generating text after Epoch: 0
------ temperature: 0.2
------ Generating with seed: "im shouting.
but you may suppose i pa"
im shouting.
but you may suppose i pa
```

10.4 Part 10.4: Image Captioning with Keras and TensorFlow

Image captioning is a new technology that combines LSTM text generation with the computer vision powers of a convolutional neural network. I first saw this technology in Andrej Karpathy's Dissertation.[20]Figure 10.4 shows images from his work.

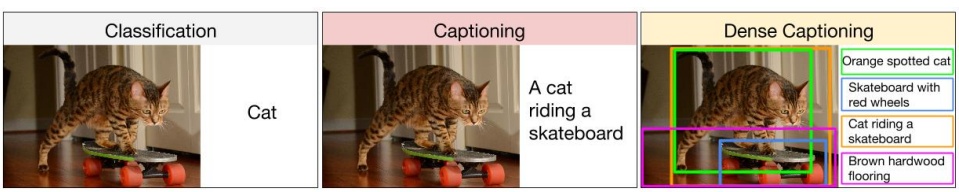


Figure 10.4: Andrej Karpathy's Dissertation

In this part, we will use LSTM and CNN to create a basic image captioning system. We will use transfer learning to utilize these two projects:

- InceptionV3
- Glove

We use inception to extract features from the images. Glove is a set of Natural Language Processing (NLP) vectors for common words. Figure 10.5 gives a high-level overview of captioning.

We begin by importing the needed libraries.

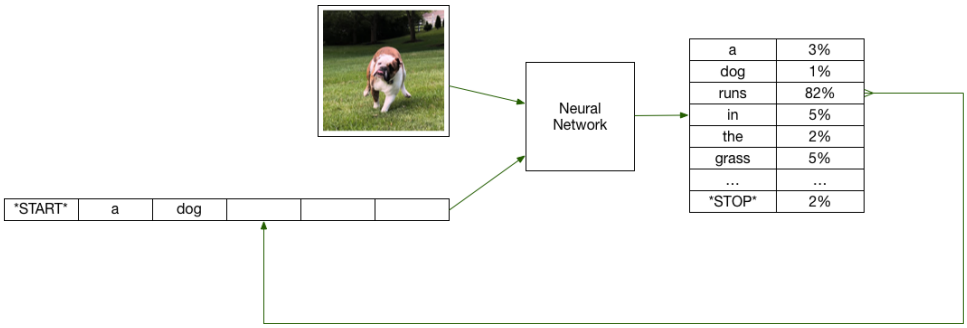


Figure 10.5: Captioning with a Neural Network

Code

```
import os
import string
import glob
from tensorflow.keras.applications import MobileNet
import tensorflow.keras.applications.mobilenet

from tensorflow.keras.applications.inception_v3 import InceptionV3
import tensorflow.keras.applications.inception_v3


from tqdm import tqdm
import tensorflow.keras.preprocessing.image
import pickle
from time import time
import numpy as np
from PIL import Image
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (LSTM, Embedding,
    TimeDistributed, Dense, RepeatVector,
    Activation, Flatten, Reshape, concatenate,
    Dropout, BatchNormalization)
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras import Input, layers
from tensorflow.keras import optimizers

from tensorflow.keras.models import Model

from tensorflow.keras.layers import add
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

START = "startseq"
```

```
STOP = "endseq"
EPOCHS = 10
USE_INCEPTION = True
```

We use the following function to nicely format elapsed times.

Code

```
# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return f"{h}:{m:>02}:{s:>05.2f}"
```

10.4.1 Needed Data

You will need to download the following data and place it in a folder for this example. Point the *root_captioning* string at the folder that you are using for the caption generation. This folder should have the following sub-folders.

- data - Create this directory to hold saved models.
- glove.6B - Glove embeddings.
- Flicker8k_Dataset - Flicker dataset.
- Flicker8k_Text

Note, the original Flickr datasets are no longer available, but you can download them from a location specified by this article.

10.4.2 Running Locally

You will need to modify the following code to reflect the folder's location that you will store files required by this captions example.

Code

```
if COLAB:
    root_captioning = "/content/drive/My Drive/projects/captions"
else:
    root_captioning = "./data/captions"
```

10.4.3 Clean/Build Dataset From Flickr8k

We must pull in the Flickr dataset captions and clean them of extra whitespace, punctuation, and other distractions.

Code

```
null_punct = str.maketrans('', '', string.punctuation)
lookup = dict()

with open( os.path.join(root_captioning,'Flickr8k_text',\
                        'Flickr8k.token.txt'), 'r') as fp:

  max_length = 0
  for line in fp.read().split('\n'):
    tok = line.split()
    if len(line) >= 2:
      id = tok[0].split('.')[0]
      desc = tok[1:]

      # Cleanup description
      desc = [word.lower() for word in desc]
      desc = [w.translate(null_punct) for w in desc]
      desc = [word for word in desc if len(word)>1]
      desc = [word for word in desc if word.isalpha()]
      max_length = max(max_length,len(desc))

      if id not in lookup:
        lookup[id] = list()
      lookup[id].append(' '.join(desc))

lex = set()
for key in lookup:
  [lex.update(d.split()) for d in lookup[key]]
```

The following code displays stats on the data downloaded and processed.

Code

```
print(len(lookup)) # How many unique words
print(len(lex)) # The dictionary
print(max_length) # Maximum length of a caption (in words)
```

Output

```
8092
8763
32
```

Next, we load the Glove embeddings.

Code

```
# Warning, running this too soon on GDrive can sometimes not work.
# Just rerun if len(img) = 0
img = glob.glob(os.path.join(root_captioning,'Flicker8k_Dataset', '*.jpg'))
```

Display the count of how many Glove embeddings we have.

Code

```
len(img)
```

Output

```
8091
```

Read all image names and use the predefined train/test sets.

Code

```
train_images_path = os.path.join(root_captioning,\
            'Flickr8k_text','Flickr_8k.trainImages.txt')
train_images = set(open(train_images_path, 'r').read().strip().split('\n'))
test_images_path = os.path.join(root_captioning,
            'Flickr8k_text','Flickr_8k.testImages.txt')
test_images = set(open(test_images_path, 'r').read().strip().split('\n'))

train_img = []
test_img = []

for i in img:
  f = os.path.split(i)[-1]
  if f in train_images:
    train_img.append(f)
  elif f in test_images:
    test_img.append(f)
```

Display the size of the train and test sets.

Code

```
print(len(train_images))
print(len(test_images))
```

Output

```
6000
1000
```

Build the sequences. We include a **start** and **stop** token at the beginning/end. We will later use the **start** token to begin the process of generating a caption. Encountering the **stop** token in the generated text will let us know the process is complete.

Code

```
train_descriptions = {k:v for k,v in lookup.items() if f'{k}.jpg' \
                      in train_images}
for n,v in train_descriptions.items():
  for d in range(len(v)):
    v[d] = f'{START} {v[d]} {STOP}'
```

See how many discriptions were extracted.

Code

```
len(train_descriptions)
```

Output

```
6000
```

10.4.4 Choosing a Computer Vision Neural Network to Transfer

This example provides two neural networks that we can use via transfer learning. In this example, I use Glove for the text embedding and InceptionV3 to extract features from the images. Both of these transfers serve to extract features from the raw text and the images. Without this prior knowledge transferred in, this example would take considerably more training.

I made it so you can interchange the neural network used for the images. By setting the values **WIDTH**, **HEIGHT**, and **OUTPUT_DIM**, you can interchange images. One characteristic that you are seeking for the image neural network is that it does not have too many outputs (once you strip the 1000-class imagenet classifier, as is common in transfer learning). InceptionV3 has 2,048 features below the classifier, and MobileNet has over 50K. If the additional dimensions truly capture aspects of the images, then they are worthwhile. However, having 50K features increases the processing needed and the complexity of the neural network we are constructing.

Code

```
if USE_INCEPTION:
  encode_model = InceptionV3(weights='imagenet')
  encode_model = Model(encode_model.input, encode_model.layers[-2].output)
  WIDTH = 299
  HEIGHT = 299
  OUTPUT_DIM = 2048
  preprocess_input = \
    tensorflow.keras.applications.inception_v3.preprocess_input
else:
  encode_model = MobileNet(weights='imagenet',include_top=False)
  WIDTH = 224
```

```
HEIGHT = 224
OUTPUT_DIM = 50176
preprocess_input = tensorflow.keras.applications.mobilenet.preprocess_input
```

Output

```
Downloading data from https://github.com/fchollet/deep-learning-models
/releases/download/v0.5/inception_v3_weights_tf_dim_ordering_tf_kernel
s.h5
96116736/96112376 [==============================] - 28s 0us/step
```

The summary of the chosen image neural network to be transferred is displayed.

Code

```
encode_model.summary()
```

Output

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Connected to
=================================================================
input_1 (InputLayer)         [(None, 299, 299, 3)      0
_________________________________________________________________
conv2d (Conv2D)              (None, 149, 149, 32)      864
input_1[0][0]
_________________________________________________________________
batch_normalization (BatchNorma (None, 149, 149, 32)   96

...

Total params: 21,802,784
Trainable params: 21,768,352
Non-trainable params: 34,432
_________________________________________________________________
```

10.4.5 Creating the Training Set

We need to encode the images to create the training set. Later we will encode new images to present them for captioning.

Code

```
def encodeImage(img):
  # Resize all images to a standard size (specified bythe image
  # encoding network)
  img = img.resize((WIDTH, HEIGHT), Image.ANTIALIAS)
  # Convert a PIL image to a numpy array
  x = tensorflow.keras.preprocessing.image.img_to_array(img)
  # Expand to 2D array
  x = np.expand_dims(x, axis=0)
  # Perform any preprocessing needed by InceptionV3 or others
  x = preprocess_input(x)
  # Call InceptionV3 (or other) to extract the smaller feature set for
  # the image.
  x = encode_model.predict(x) # Get the encoding vector for the image
  # Shape to correct form to be accepted by LSTM captioning network.
  x = np.reshape(x, OUTPUT_DIM )
  return x
```

We can how to generate the training set, which will involve looping over every JPG that we provided. Because this can take a while to perform, we will save it to a pickle file. This saved file prevents the considerable time needed to reprocess all of the images again. Because the images are processed differently by different transferred neural networks, the filename contains the output dimensions. We follow this naming convention because if you changed from InceptionV3 to MobileNet, the number of output dimensions would change, and you must reprocess the images.

Code

```
train_path = os.path.join(root_captioning,"data",f'train{OUTPUT_DIM}.pkl')
if not os.path.exists(train_path):
  start = time()
  encoding_train = {}
  for id in tqdm(train_img):
    image_path = os.path.join(root_captioning,'Flicker8k_Dataset', id)
    img = tensorflow.keras.preprocessing.image.load_img(image_path, \
            target_size=(HEIGHT, WIDTH))
    encoding_train[id] = encodeImage(img)
  with open(train_path, "wb") as fp:
    pickle.dump(encoding_train, fp)
  print(f"\nGenerating␣training␣set␣took:␣{hms_string(time()-start)}")
else:
  with open(train_path, "rb") as fp:
    encoding_train = pickle.load(fp)
```

We must also perform a similar process for the test images.

Code

```
test_path = os.path.join(root_captioning,"data",f'test{OUTPUT_DIM}.pkl')
if not os.path.exists(test_path):
  start = time()
  encoding_test = {}
  for id in tqdm(test_img):
    image_path = os.path.join(root_captioning,'Flicker8k_Dataset', id)
    img = tensorflow.keras.preprocessing.image.load_img(image_path, \
                target_size=(HEIGHT, WIDTH))
    encoding_test[id] = encodeImage(img)
  with open(test_path, "wb") as fp:
    pickle.dump(encoding_test, fp)
  print(f"\nGenerating testing set took: {hms_string(time()-start)}")
else:
  with open(test_path, "rb") as fp:
    encoding_test = pickle.load(fp)
```

Next, we separate the captions that we will use for training. There are two sides to this training, the images, and the captions.

Code

```
all_train_captions = []
for key, val in train_descriptions.items():
    for cap in val:
        all_train_captions.append(cap)
len(all_train_captions)
```

Output

```
30000
```

Words that do not occur very often can be misleading to neural network training. It is better to remove such words. Here we remove any words that occur less than ten times. We display the new reduced size of the vocabulary shrunk.

Code

```
word_count_threshold = 10
word_counts = {}
nsents = 0
for sent in all_train_captions:
    nsents += 1
    for w in sent.split(' '):
        word_counts[w] = word_counts.get(w, 0) + 1

vocab = [w for w in word_counts if word_counts[w] >= word_count_threshold]
print('preprocessed words %d ==> %d' % (len(word_counts), len(vocab)))
```

Output

```
preprocessed words 7578 ==> 1651
```

Next, we build two lookup tables for this vocabulary. The table **idxtoword** converts index numbers to actual words to index values. The **wordtoidx** lookup table performs the opposite.

Code

```
idxtoword = {}
wordtoidx = {}

ix = 1
for w in vocab:
    wordtoidx[w] = ix
    idxtoword[ix] = w
    ix += 1

vocab_size = len(idxtoword) + 1
vocab_size
```

Output

```
1652
```

Previously we added a start and stop token to all sentences. We must account for this in the maximum length of captions.

Code

```
max_length +=2
print(max_length)
```

Output

```
34
```

10.4.6 Using a Data Generator

Up to this point, we've always generated training data ahead of time and fit the neural network to it. It is not always practical to create all of the training data ahead of time. The memory demands can be considerable. If we generate the training data as the neural network needs it, it is possible to use a Keras generator. The generator will create new data as it is needed. The generator provided here creates the training data for the caption neural network, as it is needed.

If we were to build all needed training data ahead of time, it would look like Figure 10.6.

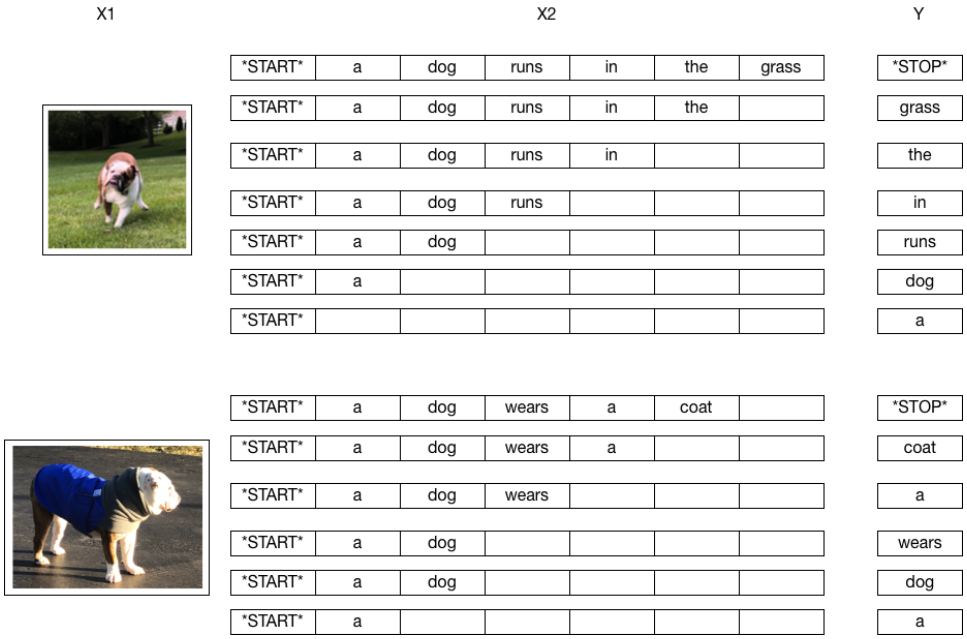


Figure 10.6: Captioning Training Data

Here we are just training on two captions. However, we would have to duplicate the image for each of these partial captions that we have. Additionally, the Flikr8K data set has five captions for each picture. Those would all require duplication of data as well. It is much more efficient to generate the data as needed.

Code

```
def data_generator(descriptions, photos, wordtoidx, \
                   max_length, num_photos_per_batch):
  # x1 - Training data for photos
  # x2 - The caption that goes with each photo
  # y - The predicted rest of the caption
  x1, x2, y = [], [], []
  n=0
  while True:
    for key, desc_list in descriptions.items():
      n+=1
      photo = photos[key+'.jpg']
      # Each photo has 5 descriptions
      for desc in desc_list:
        # Convert each word into a list of sequences.
        seq = [wordtoidx[word] for word in desc.split(' ') \
              if word in wordtoidx]
        # Generate a training case for every possible sequence and outcome
        for i in range(1, len(seq)):
          in_seq, out_seq = seq[:i], seq[i]
          in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
```

```
            out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
            x1.append(photo)
            x2.append(in_seq)
            y.append(out_seq)
        if n==num_photos_per_batch:
          yield ([np.array(x1), np.array(x2)], np.array(y))
          x1, x2, y = [], [], []
          n=0
```

10.4.7 Loading Glove Embeddings

Code

```
glove_dir = os.path.join(root_captioning,'glove.6B')
embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.200d.txt'), encoding="utf-8")

for line in tqdm(f):
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs

f.close()
print(f'Found␣{len(embeddings_index)}␣word␣vectors.')
```

Output

```
Found 400000 word vectors.
```

10.4.8 Building the Neural Network

We build an embedding matrix from Glove. We will directly copy this matrix to the weight matrix of the neural network.

Code

```
embedding_dim = 200

# Get 200-dim dense vector for each of the 10000 words in out vocabulary
embedding_matrix = np.zeros((vocab_size, embedding_dim))

for word, i in wordtoidx.items():
    #if i < max_words:
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
```

```
        # Words not found in the embedding index will be all zeros
        embedding_matrix[i] = embedding_vector
```

The matrix dimensions make sense. It is 1652 (the size of the vocabulary) by 200 (the number of features Glove generates for each word).

Code

```
embedding_matrix.shape
```

Output

```
(1652, 200)
```

Code

```
inputs1 = Input(shape=(OUTPUT_DIM,))
fe1 = Dropout(0.5)(inputs1)
fe2 = Dense(256, activation='relu')(fe1)
inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
se2 = Dropout(0.5)(se1)
se3 = LSTM(256)(se2)
decoder1 = add([fe2, se3])
decoder2 = Dense(256, activation='relu')(decoder1)
outputs = Dense(vocab_size, activation='softmax')(decoder2)
caption_model = Model(inputs=[inputs1, inputs2], outputs=outputs)
```

Code

```
embedding_dim
```

Output

```
200
```

Code

```
caption_model.summary()
```

Output

```
Model: "model_1"
_________________________________________________________________
_________________________________
```

```
Layer (type)                 Output Shape              Param #
Connected to
==================================================================================
==================================
input_3 (InputLayer)         [(None, 34)]              0
__________________________________________________________________________________
__________________________________
input_2 (InputLayer)         [(None, 2048)]            0
__________________________________________________________________________________
__________________________________
embedding (Embedding)        (None, 34, 200)           330400
input_3[0][0]

...

Total params: 1,813,268
Trainable params: 1,813,268
Non-trainable params: 0
__________________________________________________________________________________
__________________________________
```

Code

```
caption_model.layers[2].set_weights([embedding_matrix])
caption_model.layers[2].trainable = False
caption_model.compile(loss='categorical_crossentropy', optimizer='adam')
```

10.4.9 Train the Neural Network

Code

```
number_pics_per_bath = 3
steps = len(train_descriptions)//number_pics_per_bath
```

Code

```
model_path = os.path.join(root_captioning,"data",f'caption-model.hdf5')
if not os.path.exists(model_path):
  for i in tqdm(range(EPOCHS*2)):
      generator = data_generator(train_descriptions, encoding_train,
                    wordtoidx, max_length, number_pics_per_bath)
      caption_model.fit_generator(generator, epochs=1,
                    steps_per_epoch=steps, verbose=1)

  caption_model.optimizer.lr = 1e-4
  number_pics_per_bath = 6
  steps = len(train_descriptions)//number_pics_per_bath
```

```
  for i in range(EPOCHS):
      generator = data_generator(train_descriptions, encoding_train,
                      wordtoidx, max_length, number_pics_per_bath)
      caption_model.fit_generator(generator, epochs=1,
                              steps_per_epoch=steps, verbose=1)
  caption_model.save_weights(model_path)
  print(f"\Training took: {hms_string(time()-start)}")
else:
  caption_model.load_weights(model_path)
```

10.4.10 Generating Captions

It is essential to understand that we do not generate a caption with one single call to the neural network's predict function. Neural networks output a fixed-length tensor. To get a variable-length output, such as free-form text, requires multiple calls to the neural network.

The neural network accepts two objects (which we map to the input neurons). The first input is the photo, and the second input is an ever-growing caption. The caption begins with just the starting token. The neural network's output is the prediction of the next word in the caption. The caption continues to grow until the neural network predicts an end token, or we reach the maximum length of a caption.

Code

```
def generateCaption(photo):
    in_text = START
    for i in range(max_length):
        sequence = [wordtoidx[w] for w in in_text.split() if w in wordtoidx]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = caption_model.predict([photo,sequence], verbose=0)
        yhat = np.argmax(yhat)
        word = idxtoword[yhat]
        in_text += ' ' + word
        if word == STOP:
            break
    final = in_text.split()
    final = final[1:-1]
    final = ' '.join(final)
    return final
```

10.4.11 Evaluate Performance on Test Data from Flicker8k

The caption model performs relatively well on images that are similar to the training set.

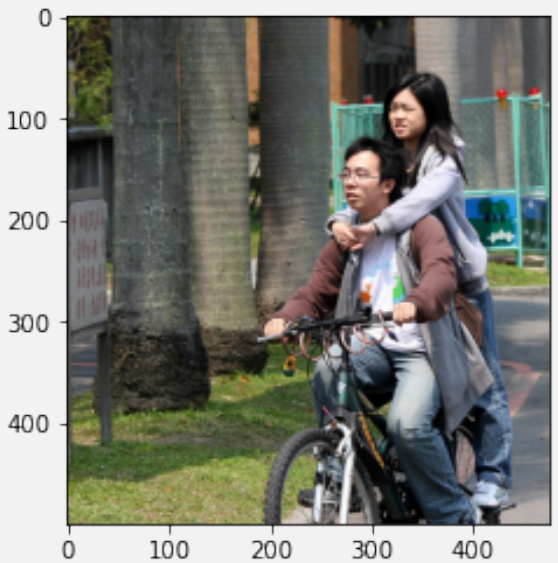
Code

```
for z in range(2): # set higher to see more examples
  pic = list(encoding_test.keys())[z]
  image = encoding_test[pic].reshape((1,OUTPUT_DIM))
```

```
    print(os.path.join(root_captioning,'Flicker8k_Dataset', pic))
    x=plt.imread(os.path.join(root_captioning,'Flicker8k_Dataset', pic))
    plt.imshow(x)
    plt.show()
    print("Caption:",generateCaption(image))
    print("_____________________________________")
```

Output

```
./data/captions\Flicker8k_Dataset\3485425825_c2f3446e73.jpg
```




```
Caption: man in white shirt is standing by woman with blue hat
_____________________________________
./data/captions\Flicker8k_Dataset\3490736665_38710f4b91.jpg
```

```
Caption: dog is chasing ball
_____________________________________
```

Code

```
encoding_test[pic].shape
```

Output

```
(2048,)
```

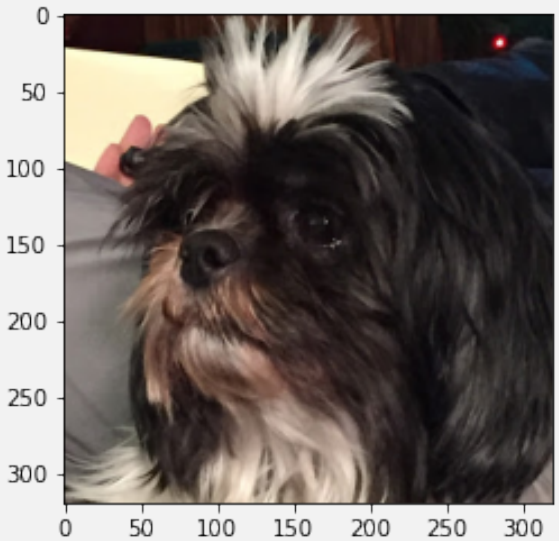
10.4.12 Evaluate Performance on My Photos

In the "photos" folder of this GitHub repository, I keep a collection of personal photos that I like to use for testing neural networks. These are entirely separate from the Flicker8K dataset, and as a result, the caption neural network does not perform nearly as well.

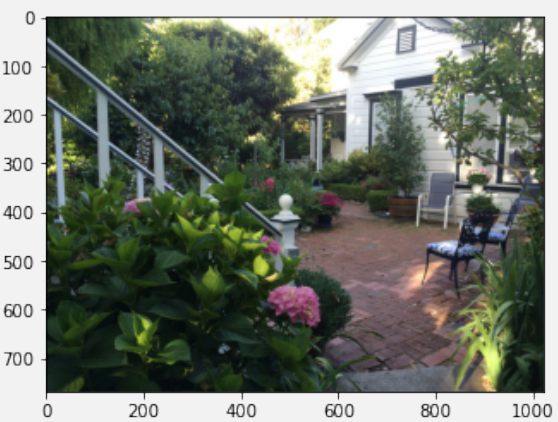
Code

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO
import numpy as np

%matplotlib inline

ROOT = "https://github.com/jeffheaton/" + \
       "t81_558_deep_learning/blob/master/photos/"
urls = [
  ROOT+"annie_dog.jpg?raw=true",
  ROOT+"landscape.jpg?raw=true",
  ROOT+"hickory_coat.jpg?raw=true"
]

for url in urls:
  response = requests.get(url)
  img = Image.open(BytesIO(response.content))
  img.load()

  plt.imshow(img)
  plt.show()

  response = requests.get(url)

  img = encodeImage(img).reshape((1,OUTPUT_DIM))
  print(img.shape)
  print("Caption:",generateCaption(img))
  print("_____")
```

Output




```
(1, 2048)
Caption: two dogs are fighting in the grass
--------------------------------------------------
```




```
(1, 2048)
Caption: two workers are sitting on the curb in front of building
--------------------------------------------------
```

```
(1, 2048)
Caption: two dogs are fighting over bone
```

10.4.13 Module 10 Assignment

You can find the first assignment here: assignment 10

10.5 Part 10.5: Temporal CNN in Keras and TensorFlow

Traditionally, we use Convolutional Neural Networks (CNNs) for image classification problems and Long Short Term Memory (LSTM) networks for time series. However, recent research has shown CNN's to be beneficial at time series problems. We begin similarly to how we started for LSTM by attempting to classify simple numeric sequences. The data preparation is nearly identical to LSTM; however, the neural network consists of a **Conv1D** layer this time around. The following code prepares the data and constructs the neural network.

Code

```
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D
import numpy as np

max_features = 4 # 0,1,2,3 (total of 4)
x = [
    [[0],[1],[1],[0],[0],[0]],
    [[0],[0],[0],[2],[2],[0]],
    [[0],[0],[0],[0],[3],[3]],
    [[0],[2],[2],[0],[0],[0]],
    [[0],[0],[3],[3],[0],[0]],
    [[0],[0],[0],[0],[1],[1]]
]
x = np.array(x,dtype=np.float32)
```

```
y = np.array([1,2,3,2,3,1],dtype=np.int32)

# Convert y2 to dummy variables
y2 = np.zeros((y.shape[0], max_features),dtype=np.float32)
y2[np.arange(y.shape[0]), y] = 1.0
y2 = np.asarray(y2).astype('float32').reshape((-1,1,4))
print(y2)

print('Build model...')
model = Sequential()
model.add(Conv1D(128, kernel_size=x.shape[1], input_shape=(None, 1)))
model.add(Dense(4, activation='sigmoid'))

# try using different optimizers and different optimizer configs
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print('Train...')
model.fit(x,y2,epochs=200)
pred = model.predict(x)
predict_classes = np.argmax(pred,axis=1)
print("Predicted classes: {}",predict_classes)
print("Expected classes: {}",predict_classes)
```

Output

```
[[[0. 1. 0. 0.]]
 [[0. 0. 1. 0.]]
 [[0. 0. 0. 1.]]
 [[0. 0. 1. 0.]]
 [[0. 0. 0. 1.]]
 [[0. 1. 0. 0.]]]
Build model...
Train...
Train on 6 samples
Epoch 1/200
6/6 [==============================] - 1s 240ms/sample - loss: 0.7315
- accuracy: 0.3333
Epoch 2/200
6/6 [==============================] - 0s 583us/sample - loss: 0.7166
- accuracy: 0.4167

...

 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
```

```
 [0  0  0  0]
 [0  0  0  0]]
```

Now that the model is trained, we will run it with sample input and see that it classifies correctly.

Code

```
def runit(model, inp):
    inp = np.array(inp,dtype=np.float32)
    pred = model.predict(inp)
    return np.argmax(pred[0])

print( runit( model, [[[0],[1],[1],[0],[0],[0]]] ))
```

Output

```
1
```

10.5.1 Sun Spots Example - CNN

We now look at an example of CNN regression to predict sunspots. You can find the data files needed for this example at the following location.

- Sunspot Data Files
- Download Daily Sunspots - 1/1/1818 to now.

We use the following code to load the sunspot file:

Code

```
import pandas as pd
import os

# Replacce the following path with your own file. It can be downloaded from:
# http://www.sidc.be/silso/INFO/sndtotcsv.php
path = "./data/"

filename = os.path.join(path,"SN_d_tot_V2.0.csv")
names = ['year', 'month', 'day', 'dec_year', 'sn_value',
         'sn_error', 'obs_num']
df = pd.read_csv(filename,sep=';',header=None,names=names,
                 na_values=['-1'], index_col=False)

print("Starting file:")
print(df[0:10])

print("Ending file:")
print(df[-10:])
```

As you can see, there is quite a bit of missing data near the end of the file. We want to find the starting index where the missing data no longer occurs. This approach is somewhat sloppy; it would be better to find a use for the data between missing values. However, the point of this example is to show how to use CNN with a somewhat simple time-series.

Code

```
start_id = max(df[df['obs_num'] == 0].index.tolist())+1
# Find the last zero and move one beyond
print(start_id)
df = df[start_id:] # Trim the rows that have missing observations
```

Next, we break the data into test and training sets. We will train the neural network in the years before 2000 and test it with years after.

Code

```
df['sn_value'] = df['sn_value'].astype(float)
df_train = df[df['year']<2000]
df_test = df[df['year']>=2000]

spots_train = df_train['sn_value'].tolist()
spots_test = df_test['sn_value'].tolist()

print("Training set has {} observations.".format(len(spots_train)))
print("Test set has {} observations.".format(len(spots_test)))
```

We will make use of the same **to_sequences** function previously used in this course. This function allows us to convert the training and test data into the 2D sequences that will train the neural network.

Code

```
import numpy as np

def to_sequences(seq_size, obs):
    x = []
    y = []

    for i in range(len(obs)-SEQUENCE_SIZE-1):
        #print(i)
        window = obs[i:(i+SEQUENCE_SIZE)]
        after_window = obs[i+SEQUENCE_SIZE]
        window = [[x] for x in window]
        #print("{} - {}".format(window,after_window))
        x.append(window)
        y.append(after_window)

    return np.array(x),np.array(y)
```

```
SEQUENCE_SIZE = 25
x_train,y_train = to_sequences(SEQUENCE_SIZE,spots_train)
x_test,y_test = to_sequences(SEQUENCE_SIZE,spots_test)

print("Shape of training set: {}".format(x_train.shape))
print("Shape of test set: {}".format(x_test.shape))
```

We can display the training data, which is essentially a 2D matrix where each row is a sequence of the same length.

Code

```
x_train
```

We are now ready to build and train the neural network. This code is similar to the previous classification example; however, now we are performing regression.

Code

```
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D,
from tensorflow.keras.layers import Dropout, MaxPooling1D, Flatten
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np

print('Build model...')
model = Sequential()
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
                 input_shape=(SEQUENCE_SIZE,1)))
model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
model.add(Dropout(0.2))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=50,
                        verbose=1, mode='auto', restore_best_weights=True)
print('Train...')

model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

Finally, we evaluate the accuracy of the predictions.

Code

```
from sklearn import metrics

# from sklearn import metrics
pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred.flatten(),y_test))
print("Score (RMSE): {}".format(score))
```

Chapter 11

Natural Language Processing and Speech Recognition

11.1 Part 11.1: Getting Started with Spacy in Python

When we apply neural networks to Natural Language Processing (NLP), we must decide if you want to operate at the word or character level. Up to this point, we've worked primarily at the character level, which was the case for the Treasure Island text pirate story generator that we previously saw. Likewise, we used word-level NLP for the image caption generator. In this module, the focus will be primarily on word-level NLP. Notably, we will examine some of the NLP tools that we can be used to process words before we send them to a neural network. There are two prevalent NLP libraries for Python:

- NLTK
- Spacy

In this course, we will focus on Spacy. I prefer spacy because of the object abstraction of sentences that it provides. However, both are widely used libraries.

11.1.1 Installing Spacy

You can install Spacy with a simple PIP install. Spacy was included in the list of packages to install for this course. You will need to ensure that you've installed a language with Spacy. If you do not, you will get the following error:

```
OSError: [E050] Can't find model 'en_core_web_sm'. It doesn't seem
to be a shortcut link, a Python package or a valid path to a
data directory.
```

To install English, use the following command:

```
python -m spacy download en
```

11.1.2 Tokenization

Tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation. Consider how the program might break up the following sentences into words.

- This is a test.
- Ok, but what about this?
- Is U.S.A. the same as USA.?
- What is the best data-set to use?
- I think I will do this-no wait; I will do that.

Code

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp(u"Apple is looking at buying a U.K. startup for $1 billion")
for token in doc:
    print(token.text)
```

Output

```
Apple
is
looking
at
buying
a
U.K.
startup
for
$
1
billion
```

You can also obtain the part of speech for each word. Common parts of speech include nouns, verbs, pronouns, and adjectives.

Code

```
for word in doc:
    print(word.text,   word.pos_)
```

Output

```
Apple PROPN
is AUX
looking VERB
```

```
at ADP
buying VERB
a DET
U.K. PROPN
startup NOUN
for ADP
$ SYM
1 NUM
billion NUM
```

Spacy includes functions to check if parts of a sentence appear to be numbers, acronyms, or other entities.

Code

```
for word in doc:
    print(f"{word} is like number? {word.like_num}")
```

Output

```
Apple is like number? False
is is like number? False
looking is like number? False
at is like number? False
buying is like number? False
a is like number? False
U.K. is like number? False
startup is like number? False
for is like number? False
$ is like number? False
1 is like number? True
billion is like number? True
```

11.1.3 Sentence Diagramming

For years grade school children have had to endure "sentence diagramming." Such diagrams can help students to understand sentence structure. Spacy provides a means to produce similar sentence diagrams. For example, the sentence "My name is Jeff" is diagrammed as follows by Spacy as Figure 11.1.

I provide the code needed to diagram this sentence below. This code generates a scrollable, interactive display. Because of this interactivity, you will need to stop this cell in Jupyter to continue. Also, the code does not show its output until you run this cell. You will not see it on GitHub or the printed book.

Code

```
import spacy
from spacy import displacy

nlp = spacy.load("en_core_web_sm")
doc = nlp(u"My name is Jeff.")
```

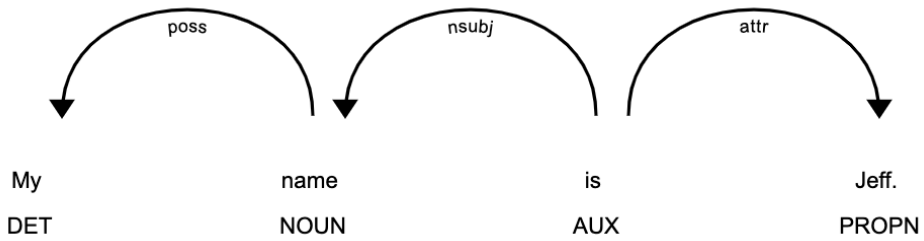


Figure 11.1: Spacy Sentence Diagram

```
displacy.serve(doc, style="dep")
```

Output

```
Using the 'dep' visualizer
Serving on http://0.0.0.0:5000 ...
Shutting down server on port 5000.
```

Note, you will have to manually stop the above cell

Code

```
print(doc)
```

Output

```
My name is Jeff.
```

The following code shows how to reduce words to their stems. Here the sentence words are reduced to their most basic form. For example, "striped" to "stripe."

Code

```
import spacy

# Initialize spacy 'en' model, keeping only tagger
# component needed for lemmatization
nlp = spacy.load('en_core_web_sm', disable=['parser', 'ner'])

sentence = "The striped bats are hanging on their feet for best"

# Parse the sentence using the loaded 'en' model object `nlp`
doc = nlp(sentence)

# Extract the lemma for each token and join
```

```
" ".join([token.lemma_ for token in doc])
```

Output

```
'the stripe bat be hang on -PRON- foot for good'
```

11.1.4 Stop Words

Stop words are words which are filtered out before or after processing of natural language text. Though "stop words" usually refers to the most common words in a language, there is no single universal list of stop words used by all natural language processing tools. Spacy contains a list of stop words that you may wish to exclude or treat with less importance in your programs.

Code

```
from spacy.lang.en.stop_words import STOP_WORDS

print(STOP_WORDS)
```

Output

```
{'toward', 'around', 'yours', 'moreover', 'for', 'and', 'many',
'have', 'noone', 'thereby', 'must', 'among', 'beside', 'make',
'might', "'s", 'please', 'three', 'indeed', 'seem', 'ever', 'he',
'ten', 'too', 'already', 'who', 'had', 'wherever', 'back', 'hence',
'when', 'done', 'now', 'm', 'therefore', 'using', 'elsewhere', 'will',
'them', 'being', 'anything', 'twelve', 'part', 'ca', 'nothing', 'get',
'nobody', 'we', 'somehow', 'alone', 'beforehand', 'ourselves', 'show',
'off', 'him', 'amongst', 'although', 'doing', 'a', 'am', 'anyone',
'as', 'something', 'else', 'those', 'if', 'whoever', 'has', 'does',
'this', 'only', 'becomes', 'twenty', 'by', 'became', 'except', 'well',
'any', 'across', 'fifty', 'neither', 'in', 'front', 'formerly',
'perhaps', 'through', 'whence', 're', 'again', 'to', 're', 'below',
'whose', 'hers', 'first', 'most', 'did', 'serious', 'may', 'they',
'herein', 'everyone', 'it', 'anyhow', 'whither', 'was', 'his',
'whenever', 'itself', 'call', 'becoming', 'fifteen', 'thence', 'her',

...

'nowhere', 'under', 'do', 'seems', 'yet', 'further', 'name', 'my',
'against', 'quite', 'myself', 'behind', 'an', 'whereby', 'll', 'two',
've', 'onto', 'much', 'really', 'nt', 'no', 'sometimes', 'down', 'or',
'should', 'third', 'thereafter', 'you', 'eight', 'without', 'which',
'very', 'so', 'always', 'before', 'otherwise', 'either'}
```

11.2 Part 11.2: Word2Vec and Text Classification

Word2vec is a group of related models that data scientists use to produce word embeddings, which are numeric representations for words. For example, a word embedding lexicon may provide a 100-number vector for each word in the English dictionary. Word2vec is one such embedding.

Word2vec is implemented by shallow, two-layer neural networks that trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus assigned a corresponding vector in high dimension space. Similar words will have similar vectors in this high dimension space.[29]

11.2.1 Suggested Software for Word2Vec

The following URLs provide useful software and data for working with Word2vec.

- GoogleNews Vectors, GitHub Mirror
- Python Gensim

The Python package Gensim is used in this chapter to work with word2vec vectors. It is also necessary to load the embedding lookup table. The following code can download this table.

Code

```
from tensorflow.keras.utils import get_file

try:
    path = get_file('GoogleNews-vectors-negative300.bin.gz',
        origin='https://s3.amazonaws.com/dl4j-distribution/' +\
                    'GoogleNews-vectors-negative300.bin.gz')
except:
    print('Error downloading')
    raise

print(path)
```

Output

```
C:\Users\jheaton\.keras\datasets\GoogleNews-vectors-negative300.bin.gz
```

The following code loads the vector lookup tables and prepares Gensim for use.

Code

```
import gensim

# Not that the path below refers to a location on my hard drive.
# You should download GoogleNews Vectors (see suggested software above)
model = gensim.models.KeyedVectors.load_word2vec_format(path, binary=True)
```

Word2vec makes each word a vector. We are using the 300-number vector, which can be seen for the word "hello".

Code

```
w = model['hello']
```

Code

```
print(len(w))
```

Output

```
300
```

Code

```
print(w)
```

Output

```
[-0.05419922  0.01708984 -0.00527954  0.33203125 -0.25
-0.01397705
 -0.15039062 -0.265625    0.01647949  0.3828125  -0.03295898
-0.09716797
 -0.16308594 -0.04443359  0.00946045  0.18457031  0.03637695
0.16601562
  0.36328125 -0.25585938  0.375       0.171875    0.21386719
-0.19921875
  0.13085938 -0.07275391 -0.02819824  0.11621094  0.15332031
0.09082031
  0.06787109 -0.0300293  -0.16894531 -0.20800781 -0.03710938
-0.22753906
  0.26367188  0.012146    0.18359375  0.31054688 -0.10791016
-0.19140625
  0.21582031  0.13183594 -0.03515625  0.18554688 -0.30859375

...

0.02612305
 -0.11474609  0.265625   -0.02453613  0.11083984 -0.02514648
-0.12060547
  0.05297852  0.07128906  0.00063705 -0.36523438 -0.13769531
-0.12890625]
```

The code below shows the distance between two words.

Code

```
import numpy as np

w1 = model['king']
w2 = model['queen']

dist = np.linalg.norm(w1-w2)

print(dist)
```

Output

```
2.4796925
```

This shows the classic word2vec equation of **queen = (king - man) + female**

Code

```
model.most_similar(positive=['woman', 'king'], negative=['man'])
```

Output

```
[('queen', 0.7118192911148071),
 ('monarch', 0.6189674139022827),
 ('princess', 0.5902431607246399),
 ('crown_prince', 0.5499460697174072),
 ('prince', 0.5377321243286133),
 ('kings', 0.5236844420433044),
 ('Queen_Consort', 0.5235945582389832),
 ('queens', 0.5181134343147278),
 ('sultan', 0.5098593235015869),
 ('monarchy', 0.5087411999702454)]
```

The following code shows which item does not belong with the others.

Code

```
model.doesnt_match("house garage store dog".split())
```

Output

```
'dog'
```

The following code shows the similarity between two words.

Code

```
model.similarity('iphone', 'android')
```

Output

```
0.5633577
```

The following code shows which words are most similar to the given one.

Code

```
model.most_similar('dog')
```

Output

```
[('dogs', 0.8680490255355835),
 ('puppy', 0.8106428384780884),
 ('pit_bull', 0.780396044254303),
 ('pooch', 0.7627376317977905),
 ('cat', 0.7609457969665527),
 ('golden_retriever', 0.7500901818275452),
 ('German_shepherd', 0.7465174198150635),
 ('Rottweiler', 0.7437615394592285),
 ('beagle', 0.7418621778488159),
 ('pup', 0.7406911253929138)]
```

11.3 Part 11.3: What are Embedding Layers in Keras

Embedding Layers are a handy feature of Keras that allows the program to automatically insert additional information into the data flow of your neural network. In the previous section, you saw that Word2Vec could expand words to a 300 dimension vector. An embedding layer would allow you to insert these 300-dimension vectors in the place of word-indexes automatically.

Programmers often use embedding layers with Natural Language Processing (NLP); however, they can be used in any instance where you wish to insert a lengthier vector in an index value place. In some ways, you can think of an embedding layer as dimension expansion. However, the hope is that these additional dimensions provide more information to the model and provide a better score.

11.3.1 Simple Embedding Layer Example

- **input_dim** = How large is the vocabulary? How many categories are you encoding? This parameter is the number of items in your "lookup table."
- **output_dim** = How many numbers in the vector that you wish to return.
- **input_length** = How many items are in the input feature vector that you need to transform?

Now we create a neural network with a vocabulary size of 10, which will reduce those values between 0-9

to 4 number vectors. Each feature vector coming in will have two such features. This neural network does nothing more than pass the embedding on to the output. But it does let us see what the embedding is doing.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding
import numpy as np

model = Sequential()
embedding_layer = Embedding(input_dim=10, output_dim=4, input_length=2)
model.add(embedding_layer)
model.compile('adam', 'mse')
```

Let's take a look at the structure of this neural network so that we can see what is happening inside it.

Code

```
model.summary()
```

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 2, 4)              40
=================================================================
Total params: 40
Trainable params: 40
Non-trainable params: 0
_________________________________________________________________
```

For this neural network, which is just an embedding layer, the input is a vector of size 2. These two inputs are integer numbers from 0 to 9 (corresponding to the requested input_dim quantity of 10 values). Looking at the summary above, we see that the embedding layer has 40 parameters. This value comes from the embedded lookup table that contains four amounts (output_dim) for each of the 10 (input_dim) possible integer values for the two inputs. The output is 2 (input_length) length 4 (output_dim) vectors, resulting in a total output size of 8, which corresponds to the Output Shape given in the summary above.

Now, let us query the neural network with two rows. The input is two integer values, as was specified when we created the neural network.

Code

```
input_data = np.array([
    [1,2]
])

pred = model.predict(input_data)
```

```
print(input_data.shape)
print(pred)
```

Output

```
(1, 2)
[[[ 0.00050902 -0.0099237  -0.02883428 -0.00821529]
  [ 0.02421514  0.03112716  0.02453538 -0.01354214]]]
```

Here we see two length-4 vectors that Keras looked up for each of the input integers. Recall that Python arrays are zero-based. Keras replaced the value of 1 with the second row of the 10 x 4 lookup matrix. Similarly, Keras replaced the value of 2 by the third row of the lookup matrix. The following code displays the lookup matrix in its entirety. The embedding layer performs no mathematical operations other than inserting the correct row from the lookup table.

Code

```
embedding_layer.get_weights()
```

Output

```
[array([[ 3.9767805e-02,  1.3598096e-02, -1.1770856e-02,
-3.6321927e-02],
        [ 5.0902367e-04, -9.9236965e-03, -2.8834283e-02,
-8.2152858e-03],
        [ 2.4215136e-02,  3.1127159e-02,  2.4535384e-02,
-1.3542138e-02],
        [ 4.6041872e-02,  4.6050500e-02,  2.2079099e-02,
-4.1100323e-02],
        [-1.9084716e-02,  1.5681457e-02,  1.9137934e-04,
1.2393482e-03],
        [ 5.9496611e-05, -4.3054130e-02,  3.0203927e-02,
-3.3005968e-02],
        [ 1.4646780e-02, -2.4961460e-02, -8.0889687e-03,
3.7561730e-04],
        [ 3.7584487e-02, -3.7326049e-02,  3.9304320e-02,

...

        [ 1.3853524e-02, -1.7934263e-02, -4.2281806e-02,
-3.8661052e-02],
        [-4.2616617e-02,  1.4965128e-02, -3.5379924e-02,
-3.9788373e-03]],
       dtype=float32)]
```

The values above are random parameters that Keras generated as starting points. Generally, we will either transfer an embedding or train these random values into something useful. The next section demonstrates

how to embed a hand-coded embedding.

11.3.2 Transferring An Embedding

Now, we see how to hard-code an embedding lookup that performs a simple one-hot encoding. One-hot encoding would transform the input integer values of 0, 1, and 2 to the vectors $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ respectively. The following code replaced the random lookup values in the embedding layer with this one-hot coding inspired lookup table.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding
import numpy as np

embedding_lookup = np.array([
    [1,0,0],
    [0,1,0],
    [0,0,1]
])

model = Sequential()
embedding_layer = Embedding(input_dim=3, output_dim=3, input_length=2)
model.add(embedding_layer)
model.compile('adam', 'mse')

embedding_layer.set_weights([embedding_lookup])
```

We have the following parameters to the Embedding layer:

- input_dim=3 - There are three different integer categorical values allowed.
- output_dim=3 - Per one-hot encoding, three columns represent a categorical value with three possible values.
- input_length=2 - The input vector has two of these categorical values.

Now we query the neural network with two categorical values to see the lookup performed.

Code

```
input_data = np.array([
    [0,1]
])

pred = model.predict(input_data)

print(input_data.shape)
print(pred)
```

Output

```
(1, 2)
```

```
[[[1. 0. 0.]
  [0. 1. 0.]]]
```

The given output shows that we provided the program with two rows from the one-hot encoding table. This encoding is a correct one-hot encoding for the values 0 and 1, where there are up to 3 unique values possible.

The next section demonstrates how to train this embedding lookup table.

11.3.3 Training an Embedding

First, we make use of the following imports.

Code

```
from numpy import array
from tensorflow.keras.preprocessing.text import one_hot
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Embedding, Dense
```

We create a neural network that classifies restaurant reviews according to positive or negative. This neural network can accept strings as input, such as given here. This code also includes positive or negative labels for each review.

Code

```
# Define 10 resturant reviews.
reviews = [
    'Never coming back!',
    'Horrible service',
    'Rude waitress',
    'Cold food.',
    'Horrible food!',
    'Awesome',
    'Awesome service!',
    'Rocks!',
    'poor work',
    'Couldn\'t have done better']

# Define labels (1=negative, 0=positive)
labels = array([1,1,1,1,1,0,0,0,0,0])
```

Notice that the second to the last label is incorrect. Errors such as this are not too out of the ordinary, as most training data could have some noise.

We define a vocabulary size of 50 words. Though we do not have 50 words, it is okay to use a value larger than needed. If there are more than 50 words, the least frequently used words in the training set are automatically dropped by the embedding layer during training. For input, we one-hot encode the strings. Note that we use the TensorFlow one-hot encoding method here, rather than Scikit-Learn. Scikit-learn would expand these strings to the 0's and 1's as we would typically see for dummy variables. TensorFlow translates all of the words to index values and replaces each word with that index.

Code

```
VOCAB_SIZE = 50
encoded_reviews = [one_hot(d, VOCAB_SIZE) for d in reviews]
print(f"Encoded reviews: {encoded_reviews}")
```

Output

```
Encoded reviews: [[38, 15, 15], [16, 20], [24, 15], [18, 24], [16,
24], [35], [35, 20], [7], [37, 32], [21, 3, 27, 16]]
```

The program one-hot encodes these reviews to word indexes; however, their lengths are different. We pad these reviews to 4 words and truncate any words beyond the fourth word.

Code

```
MAX_LENGTH = 4

padded_reviews = pad_sequences(encoded_reviews, maxlen=MAX_LENGTH, \
                               padding='post')
print(padded_reviews)
```

Output

```
[[38 15 15  0]
 [16 20  0  0]
 [24 15  0  0]
 [18 24  0  0]
 [16 24  0  0]
 [35  0  0  0]
 [35 20  0  0]
 [ 7  0  0  0]
 [37 32  0  0]
 [21  3 27 16]]
```

Each review is padded by appending zeros at the end, as specified by the padding=post setting. Next, we create a neural network to learn to classify these reviews.

Code

```
model = Sequential()
embedding_layer = Embedding(VOCAB_SIZE, 8, input_length=MAX_LENGTH)
model.add(embedding_layer)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])

print(model.summary())
```

Output

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 4, 8)              400
_________________________________________________________________
flatten (Flatten)            (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 433
Trainable params: 433
Non-trainable params: 0
_________________________________________________________________
None
```

This network accepts four integer inputs that specify the indexes of a padded movie review. The first embedding layer converts these four indexes into four vectors of length 8. These vectors come from the lookup table that contains 50 (VOCAB_SIZE) rows of vectors of length 8. This encoding is evident by the 400 (8 times 50) parameters in the embedding layer. The size of the output from the embedding layer is 32 (4 words expressed as 8-number embedded vectors). A single output neuron is connected to the embedding layer by 33 weights (32 from the embedding layer and a single bias neuron). Because this is a single-class classification network, we use the sigmoid activation function and binary_crossentropy.

The program now trains the neural network. Both the embedding lookup and dense 33 weights are updated to produce a better score.

Code

```
# fit the model
model.fit(padded_reviews, labels, epochs=100, verbose=0)
```

Output

```
<tensorflow.python.keras.callbacks.History at 0x63f0cd210>
```

We can see the learned embeddings. Think of each word's vector as a location in 8 dimension space where words associated with positive reviews are close to other words with positive reviews. Similarly, training places negative reviews close to each other. In addition to the training setting these embeddings, the 33 weights between the embedding layer and output neuron similarly learn to transform these embeddings into an actual prediction. You can see these embeddings here.

Code

```
print(embedding_layer.get_weights()[0].shape)
print(embedding_layer.get_weights())
```

Output

```
(50, 8)
[array([[-1.03855960e-01, -5.59105724e-02, -6.04345910e-02,
         1.50698468e-01,  8.98860618e-02,  8.96284208e-02,
         1.65250570e-01,  4.28558849e-02],
        [ 2.30258144e-02,  4.66232412e-02, -3.89982834e-02,
         2.99915113e-02, -2.06652526e-02,  3.44550945e-02,
        -1.44930705e-02, -4.07445915e-02],
        [-4.72828634e-02,  1.85899399e-02,  8.94228369e-03,
         9.16576385e-03, -1.39516592e-03,  3.86941768e-02,
        -2.00331211e-04, -2.19582673e-02],
        [-1.38059288e-01, -6.69398904e-02, -1.37284666e-01,
         8.37527588e-02,  8.94882604e-02,  9.02104154e-02,
         6.08446635e-02,  1.74611673e-01],
        [-3.90188769e-03,  2.26297863e-02, -3.32948118e-02,
        -7.31148571e-03,  3.02187465e-02, -4.14650813e-02,

...

         4.76033799e-02, -1.80492997e-02, -9.64826345e-03,
        -8.73423740e-03,  4.65954877e-02],
        [-3.16178575e-02, -4.28569689e-02,  4.80644591e-02,
        -9.59502533e-03,  3.28160785e-02,  2.81270780e-02,
         7.52024725e-03, -4.61859480e-02]], dtype=float32)]
```

We can now evaluate this neural network's accuracy, including both the embeddings and the learned dense layer.

Code

```
loss, accuracy = model.evaluate(padded_reviews, labels, verbose=0)
print(f'Accuracy: {accuracy}')
```

Output

```
Accuracy: 1.0
```

The accuracy is a perfect 1.0, indicating there is likely overfitting. For a more complex data set, it would be good to use early stopping to not overfit.

Code

```
print(f'Log-loss: {loss}')
```

Output

```
Log-loss: 0.4717821180820465
```

However, the loss is not perfect, meaning that even though the predicted probabilities indicated a correct prediction in every case, the program did not achieve absolute confidence in each correct answer. The lack of confidence was likely due to the small amount of noise (previously discussed) in the data set. Additionally, the fact that some words appeared in both positive and negative reviews contributed to this lack of absolute certainty.

11.4 Part 11.4: Natural Language Processing with Spacy and Keras

In this part, we will see how to use Spacy and Keras together.

11.4.1 Word-Level Text Generation

There are several different approaches to teaching a neural network to output free-form text. The most basic question is if you wish the neural network to learn at the word or character level. In many ways, learning at the character level is the more interesting of the two. The LSTM is learning to construct its own words without even being shown what a word is. We will begin with character-level text generation. In the next module, we will see how we can use nearly the same technique to operate at the word level. We will implement the automatic captioning in the next module is at the word level.

We begin by importing the needed Python packages and defining the sequence length, named **maxlen**. Time-series neural networks always accept their input as a fixed-length array. We might not need all of the sequence elements for shorter sentences. It is common to fill extra unneeded elements with zeros. The program divides the text into sequences of this length. Then the neural network will be trained to predict what comes after this sequence.

Code

```
from tensorflow.keras.callbacks import LambdaCallback
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import random
import sys
import io
import requests
import re
```

Code

```
import requests

r = requests.get("https://data.heatonresearch.com/data/t81-558/text/"\
                 "treasure_island.txt")
raw_text = r.text.lower()
```

```
print(raw_text[0:1000])
```

Output

```
the project gutenberg ebook of treasure island, by robert louis
stevenson
this ebook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  you may copy it, give it away or
re-use it under the terms of the project gutenberg license included
with this ebook or online at www.gutenberg.net
title: treasure island
author: robert louis stevenson
illustrator: milo winter
release date: january 12, 2009 [ebook #27780]
language: english
*** start of this project gutenberg ebook treasure island ***
produced by juliet sutherland, stephen blundell and the
online distributed proofreading team at http://www.pgdp.net
 the illustrated children's library

...

            milo winter
           [illustration]
           gramercy books
              new york
 foreword copyright  1986 by random house v
```

Code

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp(raw_text)
vocab = set()
tokenized_text = []

for token in doc:
    word = ''.join([i if ord(i) < 128 else ' ' for i in token.text])
    word = word.strip()
    if not token.is_digit \
        and not token.like_url \
        and not token.like_email:
        vocab.add(word)
        tokenized_text.append(word)
```

```
print(f"Vocab size: {len(vocab)}")
```

Output

```
Vocab size: 6418
```

The above section might have given you this error:

```
OSError: [E050] Can't find model 'en_core_web_sm'. It doesn't seem to be
a shortcut link, a Python package, or a valid path to a data directory.
```

If so, you can install Spacy with a simple PIP install. Spacy was included in the list of packages to install for this course. You will need to ensure that you've installed a language with Spacy. If you do not, you will get the following error:

To install English, use the following command:

```
python –m spacy download en
```

We can now display the vocab words.

Code

```
print(list(vocab)[:20])
```

Output

```
['', 'dilapidation', 'supercargo', 'nautical', 'mexican', 'girdle',
'repetition', 'busiest', 'coughing', 'raged', 'spitting', 'how',
'rang', 'corpse--"o\'brien', 'usage', 'sentinel', 'stab',
'consequence', 'listing', 'ease']
```

We need an easy way to convert words into indexes and vise versa. The following code builds two such indexes.

Code

```
word2idx = dict((n, v) for v, n in enumerate(vocab))
idx2word = dict((n, v) for n, v in enumerate(vocab))
```

We can now tokenize the text; this process replaces each word with the correct token.

Code

```
tokenized_text = [word2idx[word] for word in tokenized_text]
```

If we display the tokenized text, we see an array of index values for each word.

Code

```
tokenized_text
```

Output

```
[6102,
 1109,
 4916,
 1052,
 1139,
 3461,
 4965,
 5162,
 896,
 3956,
 4444,
 569,
 0,
 5805,
 1052,

...

 3484,
 1748,
 5162,
 0,
 ...]
```

Next, we break the tokenized text into sequences that are of consistent length. It is necessary to specify this length; here we use a sequence length of 6.

Code

```
# cut the text in semi-redundant sequences of maxlen words
maxlen = 6
step = 3
sentences = []
next_words = []
for i in range(0, len(tokenized_text) - maxlen, step):
    sentences.append(tokenized_text[i: i + maxlen])
    next_words.append(tokenized_text[i + maxlen])
print('nb sequences:', len(sentences))
```

Output

```
nb sequences: 32016
```

We can display the first five sequences to get an idea of the appearance of the data.

Code

```
sentences[0:5]
```

Output

```
[[6102, 1109, 4916, 1052, 1139, 3461],
 [1052, 1139, 3461, 4965, 5162, 896],
 [4965, 5162, 896, 3956, 4444, 569],
 [3956, 4444, 569, 0, 5805, 1052],
 [0, 5805, 1052, 2926, 3200, 6102]]
```

Finally, we create the x and y vectors. The x is a Numpy encoding of the tokenization that we just performed. We use the first six elements of each tokenization to predict the seventh element. We convert the next element to dummy variables, and it becomes the y. For each of the sequences, we teach the neural network to predict the sixth element (or word) based on the previous five elements.

Code

```
import numpy as np

print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(vocab)), dtype=np.bool)
y = np.zeros((len(sentences), len(vocab)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, word in enumerate(sentence):
        x[i, t, word] = 1
    y[i, next_words[i]] = 1
```

Output

```
Vectorization...
```

We display the shapes of the x and y.

Code

```
x.shape
```

Output

```
(32016, 6, 6418)
```

Code

```
y.shape
```

Output

```
(32016, 6418)
```

Because we encoded the y value o dummy variables we can see, there are 6,418 elements for each row of y. This large number of elements is because there are 6,418 words in the vocabulary. Large vocabularies can significantly increase the amount of memory needed.

Code

```
y[0:5]
```

Output

```
array([[False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [ True, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

Now we can train an LSTM-based neural network to generate text.

Code

```
# build the model: a single LSTM
print('Build model...')
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen, len(vocab))))
model.add(Dense(len(vocab), activation='softmax'))

optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Output

```
Build model...
```

Code

```
model.summary()
```

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 128)               3352064
_________________________________________________________________
dense (Dense)                (None, 6418)              827922
=================================================================
Total params: 4,179,986
Trainable params: 4,179,986
Non-trainable params: 0
_________________________________________________________________
```

This function collects sample generations from the neural network. The temperature variable specifies how conservative, or less random, the predictions will be. Higher temperatures encourage more "creativity" from the neural network; however, they also promote more nonsensical output.

Code

```
def sample(preds, temperature=1.0):
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Keras calls the following function at the end of each training Epoch. The code generates sample text generations that visually demonstrate the neural network better at text generation. As the neural network trains, the generations should look more realistic.

Code

```
def on_epoch_end(epoch, _):
    # Function invoked at end of each epoch. Prints generated text.
    print("******************************************************")
    print('----- Generating text after Epoch: %d' % epoch)

    start_index = random.randint(0, len(tokenized_text) - maxlen)
    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('----- temperature:', temperature)

        #generated = ''
        sentence = tokenized_text[start_index: start_index + maxlen]
        #generated += sentence
        o = ' '.join([idx2word[idx] for idx in sentence])
        print(f'----- Generating with seed: "{o}"')
        #sys.stdout.write(generated)
```

```
        for i in range(100):
            x_pred = np.zeros((1, maxlen, len(vocab)))
            for t, word in enumerate(sentence):
                x_pred[0, t, word] = 1.


            preds = model.predict(x_pred, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_word = idx2word[next_index]

            #generated += next_char
            sentence = sentence[1:]
            sentence.append(next_index)

            sys.stdout.write(next_word)
            sys.stdout.write(' ')
            sys.stdout.flush()
        print()
```

We will now fit the model. As the model fits, we display sample text that the model is generating. We display text at several "temperatures." For this example, temperature refers to the amount of randomness allowed in words chosen by the neural network.

Code

```
print_callback = LambdaCallback(on_epoch_end=on_epoch_end)

model.fit(x, y,
          batch_size=128,
          epochs=60,
          callbacks=[print_callback])
```

Output

```
Train on 32016 samples
Epoch 1/60
31744/32016 [============================>.] - ETA: 0s - loss: 5.7923*
**********************************************************************
*****
------ Generating text after Epoch: 0
------ temperature: 0.2
------ Generating with seed: " waving a white cloth ;"
and the  " " " said the captain , " said the captain , " said the
captain , " said the captain , " said the captain , " said the captain
, " said the captain , " said the captain , " said the captain , "
said the  " " said the captain , " said the captain , " said the
captain , " said the captain , " said the  " " said " " said the
```

```
captain , " said the captain , " said the  " "
------ temperature: 0.5

...

when . besides you rising a fortune gently will body start fresh was
the license take best talking sent a sweet
32016/32016 [==============================] - 48s 1ms/sample - loss:
1.5427
<tensorflow.python.keras.callbacks.History at 0x14b80489c08>
```

By the end of the run, we can see that the neural network generates reasonable text at mid-range temperatures. The following is a sample of the neural network created using the starting words "that man with the one leg." We use this seed/start to provide the first six words that the neural network uses for the initial prediction.

----- Generating with seed: "that man with the one leg"
and fancy dangerous 'll was above two so means five her . left you these knew joyce on john . " stand , 'em help own " use i had o'brien . " speaking , perhaps pew redruth east you water door broad the pistol from his bad below his sir , and stood up i brass for save wood , fifteen you added upon have run hawkins ; and i that . seemed look have the two for between at that brought with into told , unprotected mate you and am away chest i notice

11.5 Part 11.5: Learning English from Scratch with Keras and TensorFlow

In this section we will see how a neural network can learn the English language from scratch. We will make use of a type of neural network called end-to-end memory.[38]We will train this type of neural network on a special dataset that was created by researchers at Facebook to test a neural network's ability to answer questions.[38]

Other useful links for End-To-End Memory Networks

- bAbI Datasets - The Facebook dataset used to train this network.
- Keras End-To-End Memory Networks - Example from Keras author on end-to-end networks.
- Online JavaScript Demo of End-to-End Memory Networks

11.5.1 Imports and Utility Functions

The following imports are needed to create the end-to-end memory network. Neither Keras nor TensorFlow directly supports End-to-End Memory Networks (yet), so it is necessary to develop them using existing tools. Several functions are needed to be defined here to read the bAbI dataset that we are using to train.

Code

```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Input, Activation, Dense, Permute
from tensorflow.keras.layers import Dropout, add, dot, concatenate
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import get_file
```

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import load_model
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from functools import reduce
import pickle
import tarfile
import numpy as np
import re
import os
import time

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return f"{h}:{m:>02}:{s:>05.2f}"


def tokenize(sent):
    '''Return the tokens of a sentence including punctuation.
    >>> tokenize('Bob dropped the apple. Where is the apple?')
    ['Bob', 'dropped', 'the', 'apple', '.', 'Where', 'is', 'the',\
        'apple', '?']
    '''
    return [x.strip() for x in re.split('(\W+)', sent) if x.strip()]


def parse_stories(lines, only_supporting=False):
    '''Parse stories provided in the bAbi tasks format
    If only_supporting is true, only the sentences
    that support the answer are kept.
    '''
    data = []
    story = []
    for line in lines:
        line = line.decode('utf-8').strip()
        nid, line = line.split(' ', 1)
        nid = int(nid)
        if nid == 1:
            story = []
        if '\t' in line:
            q, a, supporting = line.split('\t')
            q = tokenize(q)
            substory = None
            if only_supporting:
                # Only select the related substory
```

```
                supporting = map(int, supporting.split())
                substory = [story[i - 1] for i in supporting]
            else:
                # Provide all the substories
                substory = [x for x in story if x]
            data.append((substory, q, a))
            story.append('')
        else:
            sent = tokenize(line)
            story.append(sent)
    return data


def get_stories(f, only_supporting=False, max_length=None):
    '''Given a file name, read the file,
    retrieve the stories,
    and then convert the sentences into a single story.
    If max_length is supplied,
    any stories longer than max_length tokens will be discarded.
    '''
    data = parse_stories(f.readlines(), only_supporting=only_supporting)
    flatten = lambda data: reduce(lambda x, y: x + y, data)
    data = [(flatten(story), q, answer) for story, q, answer in data \
            if not max_length or len(flatten(story)) < max_length]
    return data


def vectorize_stories(data):
    inputs, queries, answers = [], [], []
    for story, query, answer in data:
        inputs.append([word_idx[w] for w in story])
        queries.append([word_idx[w] for w in query])
        answers.append(word_idx[answer])
    return (pad_sequences(inputs, maxlen=story_maxlen),
            pad_sequences(queries, maxlen=query_maxlen),
            np.array(answers))
```

11.5.2 Getting the Data

The data is first downloaded from the Internet if needed. This dataset contains stories and questions about those stories. The computer is not learning these specific stories. However, it is learning how to read a story and answer a question about that story. Consider the first story, "Mary moved to the bathroom. John went to the hallway." the computer is not learning that Mary is in the bathroom or John is in the hallway, this changes per story. Instead, the machine is learning to parse the story and extract information about individual people and their locations.

The computer is learning to read, at least in a limited sense.

Code

```
try:
    path = get_file('babi-tasks-v1-2.tar.gz',
        origin='https://s3.amazonaws.com/text-datasets/"\
␣␣␣␣␣␣␣␣␣␣␣␣"babi_tasks_1-20_v1-2.tar.gz')
except:
    print("""
Error downloading dataset, please download it manually:\n'
$ wget http://www.thespermwhale.com/jaseweston/babi/
    tasks_1-20_v1-2.tar.gz\n'
$ mv tasks_1-20_v1-2.tar.gz ~/.keras/datasets/
    babi-tasks-v1-2.tar.gz""")
    raise
tar = tarfile.open(path)

challenges = {
    # QA1 with 10,000 samples
    'single_supporting_fact_10k':
        'tasks_1-20_v1-2/en-10k/qa1_single-supporting-fact_{}.txt',
    # QA2 with 10,000 samples
    'two_supporting_facts_10k':
        'tasks_1-20_v1-2/en-10k/qa2_two-supporting-facts_{}.txt',
}
challenge_type = 'single_supporting_fact_10k'
challenge = challenges[challenge_type]

print('Extracting␣stories␣for␣the␣challenge:', challenge_type)
train_stories = get_stories(tar.extractfile(challenge.format('train')))
test_stories = get_stories(tar.extractfile(challenge.format('test')))
```

Output

```
Extracting stories for the challenge: single_supporting_fact_10k
```

Code

```
# See what the data looks like

for i in range(5):
    print("Story:␣{}".format('␣'.join(train_stories[i][0])))
    print("Query:␣{}".format('␣'.join(train_stories[i][1])))
    print("Answer:␣{}".format(train_stories[i][2]))
    print("---")
```

Output

```
Story: Mary moved to the bathroom . John went to the hallway .
Query: Where is Mary ?
Answer: bathroom
---
Story: Mary moved to the bathroom . John went to the hallway . Daniel
went back to the hallway . Sandra moved to the garden .
Query: Where is Daniel ?
Answer: hallway
---
Story: Mary moved to the bathroom . John went to the hallway . Daniel
went back to the hallway . Sandra moved to the garden . John moved to
the office . Sandra journeyed to the bathroom .
Query: Where is Daniel ?
Answer: hallway
---

...

hallway . Daniel travelled to the office . John went back to the
garden . John moved to the bedroom .
Query: Where is Sandra ?
Answer: bathroom
---
```

11.5.3 Building the Vocabulary

This type of neural network can only deal with a set vocabulary. The words are indexed, and each becomes a number. Words not in the training vocabulary will not be recognized.

Code

```
vocab = set()
for story, q, answer in train_stories + test_stories:
    vocab |= set(story + q + [answer])
vocab = sorted(vocab)

# Reserve 0 for masking via pad_sequences
vocab_size = len(vocab) + 1
story_maxlen = max(map(len, (x for x, _, _ in train_stories + test_stories)))
query_maxlen = max(map(len, (x for _, x, _ in train_stories + test_stories)))

print('-')
print('Vocab size:', vocab_size, 'unique words')
print('Story max length:', story_maxlen, 'words')
print('Query max length:', query_maxlen, 'words')
print('Number of training stories:', len(train_stories))
print('Number of test stories:', len(test_stories))
print('-')
```

```
print('Here\'s what a "story" tuple looks like (input, query, answer):')
print(train_stories[0])
print('-')


for s in list(enumerate(vocab)):
    print(s)
```

Output

```
-
Vocab size: 22 unique words
Story max length: 68 words
Query max length: 4 words
Number of training stories: 10000
Number of test stories: 1000
-
Here's what a "story" tuple looks like (input, query, answer):
(['Mary', 'moved', 'to', 'the', 'bathroom', '.', 'John', 'went', 'to',
'the', 'hallway', '.'], ['Where', 'is', 'Mary', '?'], 'bathroom')
-
(0, '.')
(1, '?')
(2, 'Daniel')
(3, 'John')

...

(16, 'office')
(17, 'the')
(18, 'to')
(19, 'travelled')
(20, 'went')
```

11.5.4 Building the Training and Test Data

We present the training data to the neural network as a vectorized representation of the sentences. We replace each word with that word's corresponding vocabulary index. Additionally, there are two parts to the input (x) data: story and query. The answer (x) is always a single vocab word number. We set up this neural network for classification. Any of the vocab words could potentially be the answer. Stories can be at most 68 words and questions at most 4. The program automatically determines both of these limits from the training data.

Code

```
print('Vectorizing the word sequences...')
word_idx = dict((c, i + 1) for i, c in enumerate(vocab))
```

```
inputs_train, queries_train, answers_train \
    = vectorize_stories(train_stories)
inputs_test, queries_test, answers_test \
    = vectorize_stories(test_stories)

print('-')
print('inputs: integer tensor of shape (samples, max_length)')
print('inputs_train shape:', inputs_train.shape)
print('inputs_test shape:', inputs_test.shape)
print('-')
print('queries: integer tensor of shape (samples, max_length)')
print('queries_train shape:', queries_train.shape)
print('queries_test shape:', queries_test.shape)
print('-')
print('answers: binary (1 or 0) tensor of shape (samples, vocab_size)')
print('answers_train shape:', answers_train.shape)
print('answers_test shape:', answers_test.shape)
print('-')
```

Output

```
Vectorizing the word sequences...
-
inputs: integer tensor of shape (samples, max_length)
inputs_train shape: (10000, 68)
inputs_test shape: (1000, 68)
-
queries: integer tensor of shape (samples, max_length)
queries_train shape: (10000, 4)
queries_test shape: (1000, 4)
-
answers: binary (1 or 0) tensor of shape (samples, vocab_size)
answers_train shape: (10000,)
answers_test shape: (1000,)
-
```

Code

```
# See individual training element.

print("Story (x): {}".format(inputs_train[0]))
print("Question (x): {}".format(queries_train[0]))
print("Answer: {}".format(answers_train[0]))
```

Output

```
Story (x): [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0
  0  0  0  0  0  0  0  0  5 16 19 18  9  1  4 21 19 18 12  1]
Question (x): [ 7 13  5  2]
Answer: 9
```

11.5.5 Compile the Neural Network

We define the structure of the neural network according to the end-to-end structure defined in the paper discussed previously.

Code

```
print('Compiling...')

# placeholders
input_sequence = Input((story_maxlen,))
question = Input((query_maxlen,))

# encoders
# embed the input sequence into a sequence of vectors
input_encoder_m = Sequential()
input_encoder_m.add(Embedding(input_dim=vocab_size,
                              output_dim=64))
input_encoder_m.add(Dropout(0.3))
# output: (samples, story_maxlen, embedding_dim)

# embed the input into a sequence of vectors of size query_maxlen
input_encoder_c = Sequential()
input_encoder_c.add(Embedding(input_dim=vocab_size,
                              output_dim=query_maxlen))
input_encoder_c.add(Dropout(0.3))
# output: (samples, story_maxlen, query_maxlen)

# embed the question into a sequence of vectors
question_encoder = Sequential()
question_encoder.add(Embedding(input_dim=vocab_size,
                               output_dim=64,
                               input_length=query_maxlen))
question_encoder.add(Dropout(0.3))
# output: (samples, query_maxlen, embedding_dim)

# encode input sequence and questions (which are indices)
# to sequences of dense vectors
input_encoded_m = input_encoder_m(input_sequence)
input_encoded_c = input_encoder_c(input_sequence)
```

```
question_encoded = question_encoder(question)

# compute a 'match' between the first input vector sequence
# and the question vector sequence
# shape: `(samples, story_maxlen, query_maxlen)`
match = dot([input_encoded_m, question_encoded], axes=(2, 2))
match = Activation('softmax')(match)

# add the match matrix with the second input vector sequence
response = add([match, input_encoded_c])
# (samples, story_maxlen, query_maxlen)
response = Permute((2, 1))(response)
# (samples, query_maxlen, story_maxlen)

# concatenate the match matrix with the question vector sequence
answer = concatenate([response, question_encoded])

# the original paper uses a matrix multiplication for this reduction step.
# we choose to use a RNN instead.
answer = LSTM(32)(answer)  # (samples, 32)

# one regularization layer -- more would probably be needed.
answer = Dropout(0.3)(answer)
answer = Dense(vocab_size)(answer)  # (samples, vocab_size)
# we output a probability distribution over the vocabulary
answer = Activation('softmax')(answer)

# build the final model
model = Model([input_sequence, question], answer)
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
print("Done.")
```

Output

```
Compiling...
Done.
```

11.5.6 Train the Neural Network

It will take some time (probably up to 1/2 hour) to train this network on a CPU. Once complete, the program saves the network. If you've previously saved the neural network, you can skip this step and load it in the next step.

Code

```
start_time = time.time()
# train
model.fit([inputs_train, queries_train], answers_train,
          batch_size=32,
          epochs=120,
          validation_data=([inputs_test, queries_test], answers_test))

# save
save_path = "./data/"
# save entire network to HDF5 (save everything, suggested)
model.save(os.path.join(save_path,"chatbot.h5"))
# save the vocab too, indexes must be the same
pickle.dump( vocab, open( os.path.join(save_path,"vocab.pkl"), "wb" ) )

elapsed_time = time.time() - start_time
print("Elapsed time: {}".format(hms_string(elapsed_time)))
```

Output

```
Train on 10000 samples, validate on 1000 samples
Epoch 1/120
10000/10000 [==============================] - 8s 833us/sample - loss:
1.9471 - accuracy: 0.1651 - val_loss: 1.7972 - val_accuracy: 0.1810
Epoch 2/120
10000/10000 [==============================] - 3s 264us/sample - loss:
1.7601 - accuracy: 0.2340 - val_loss: 1.6338 - val_accuracy: 0.3360
Epoch 3/120
10000/10000 [==============================] - 3s 266us/sample - loss:
1.5809 - accuracy: 0.3713 - val_loss: 1.5326 - val_accuracy: 0.3750
Epoch 4/120
10000/10000 [==============================] - 3s 269us/sample - loss:
1.5284 - accuracy: 0.3880 - val_loss: 1.4769 - val_accuracy: 0.4120
Epoch 5/120
10000/10000 [==============================] - 3s 264us/sample - loss:

...

0.0705 - accuracy: 0.9760 - val_loss: 0.1762 - val_accuracy: 0.9490
Epoch 120/120
10000/10000 [==============================] - 3s 283us/sample - loss:
0.0708 - accuracy: 0.9753 - val_loss: 0.1477 - val_accuracy: 0.9560
Elapsed time: 0:05:49.06
```

Code

```
# Load the model, if it exists, load vocab too
save_path = "./data/"
model = load_model(os.path.join(save_path,"chatbot.h5"))
vocab = pickle.load( open( os.path.join(save_path,"vocab.pkl"), "rb" ) )
```

11.5.7 Evaluate Accuracy

We evaluate the accuracy, using the same technique as previous classification networks.

Code

```
pred = model.predict([inputs_test, queries_test])
# See what the predictions look like, they are just probabilities
# of each class.
print(pred)
```

Output

```
[[3.2316551e-18 4.5703689e-18 4.2100677e-18 ... 2.7742484e-18
  3.3352055e-18 3.6209529e-18]
 [8.6646321e-15 8.6298309e-15 7.7134844e-15 ... 8.2590549e-15
  7.4069846e-15 9.1656156e-15]
 [1.6307067e-15 2.1523891e-15 2.2016419e-15 ... 1.6251486e-15
  2.1003965e-15 2.4207346e-15]
 ...
 [2.0678353e-16 2.2777190e-16 2.1688923e-16 ... 2.1682140e-16
  2.0552370e-16 1.9355541e-16]
 [5.6333461e-17 5.9818837e-17 4.9700267e-17 ... 5.3023650e-17
  6.0595586e-17 5.4082653e-17]
 [1.6186161e-09 1.6892178e-09 1.6479546e-09 ... 1.6192523e-09
  1.3914115e-09 1.7729884e-09]]
```

Code

```
# Use argmax to turn those into actual predictions.  The class (word)
# with the highest
# probability is the answer.

pred = np.argmax(pred,axis=1)
print(pred)
```

Output

```
[12  9 15 12 15 12 11 12 17 17 15 11 17 10 10 15 11 15 17 12 12 17 17
12
```

```
 10  10  10  15  15  15  12  17  15  15   9  17   9  11  15  10   9  11  10  12  11  15  12
9
 17  10  10  17  11  11  12  10  15  12  11  12  12  17   9  11  10  15  15   9  17  17  11
12
 11  10   9  15  15  15  12  17  11   9  10  15  17  11  11  12  15  15   9   9  12   9   9
15
 15  15  10  10  10   9  17  11  11  12  10   9  17  10   9   9  15  12  15  11  17  15   9
17
 10   9   9  12  10  17  11  11  11   9  11  12   9  11  17   9  17  15   9   9  17  12  12
17
 10  15  15  15  15  15  11  11   9  17  10   9  11  10  17  12  10  10  10  11   9  11  11
11
 15  10  17  17  11   9   9  17  15  10  17  17  12  12  12  12  10   9  17  11  15  17  12

...

 12  11  10  10  10  17  17  15  12  17   9   9  15  10  12  15  15  15  17  17  15  11  15
10
 12   9   9  10  11  11   9   9  12  17   9  15   9  12  12  10  17  17  10  12  15   9  10
17
 11  11  11  17   9   9  11  15  11  11  11  15  12  17   9  11]
```

Code

```
score = metrics.accuracy_score(answers_test, pred)
print("Final accuracy: {}".format(score))
```

Output

```
Final accuracy: 0.956
```

11.5.8 Adhoc Query

You might want to create your own stories and questions.

Code

```
print("Remember, I only know these words: {}".format(vocab))
print()
story = "Daniel went to the hallway. Mary went to the bathroom."\
    "Daniel went to the bedroom."
query = "Where is Sandra?"

adhoc_stories = (tokenize(story), tokenize(query), '?')

adhoc_train, adhoc_query, adhoc_answer = vectorize_stories([adhoc_stories])
```

```
pred = model.predict([adhoc_train, adhoc_query])
print(pred[0])
pred = np.argmax(pred,axis=1)
print("Answer: {}({})".format(vocab[pred[0]-1],pred))
```

Output

```
Remember, I only know these words: ['.', '?', 'Daniel', 'John',
'Mary', 'Sandra', 'Where', 'back', 'bathroom', 'bedroom', 'garden',
'hallway', 'is', 'journeyed', 'kitchen', 'moved', 'office', 'the',
'to', 'travelled', 'went']
[5.3319661e-11 5.9570487e-11 5.1706601e-11 5.1733437e-11 5.4510025e-11
 5.7622622e-11 4.8129528e-11 5.3163075e-11 5.6659514e-11 8.2092601e-01
 2.8999595e-02 1.4757804e-02 3.3609481e-03 5.6475557e-11 5.3110880e-11
 1.3155086e-01 5.3828338e-11 4.0473844e-04 5.5407737e-11 5.2006194e-11
 5.0534833e-11 5.6617922e-11]
Answer: bathroom([9])
```

Chapter 12

Reinforcement Learning

12.1 Part 12.1: Introduction to the OpenAI Gym

OpenAI Gym aims to provide an easy-to-setup general-intelligence benchmark with a wide variety of different environments. The goal is to standardize how environments are defined in AI research publications so that published research becomes more easily reproducible. The project claims to provide the user with a simple interface. As of June 2017, developers can only use Gym with Python.

OpenAI gym is pip-installed onto your local machine. There are a few significant limitations to be aware of:

- OpenAI Gym Atari only **directly** supports Linux and Macintosh
- OpenAI Gym Atari can be used with Windows; however, it requires a particular installation procedure
- OpenAI Gym can not directly render animated games in Google CoLab.

Because OpenAI Gym requires a graphics display, the only way to display Gym in Google CoLab is an embedded video. The presentation of OpenAI Gym game animations in Google CoLab is discussed later in this module.

12.1.1 OpenAI Gym Leaderboard

The OpenAI Gym does have a leaderboard, similar to Kaggle; however, the OpenAI Gym's leaderboard is much more informal compared to Kaggle. The user's local machine performs all scoring. As a result, the OpenAI gym's leaderboard is strictly an "honor's system." The leaderboard is maintained the following GitHub repository:

- OpenAI Gym Leaderboard

If you submit a score, you are required to provide a writeup with sufficient instructions to reproduce your result. A video of your results is suggested, but not required.

12.1.2 Looking at Gym Environments

The centerpiece of Gym is the environment, which defines the "game" in which your reinforcement algorithm will compete. An environment does not need to be a game; however, it describes the following game-like features:

- **action space**: What actions can we take on the environment, at each step/episode, to alter the environment.
- **observation space**: What is the current state of the portion of the environment that we can observe. Usually, we can see the entire environment.

Before we begin to look at Gym, it is essential to understand some of the terminology used by this library.

- **Agent** - The machine learning program or model that controls the actions.

Step - One round of issuing actions that affect the observation space.

- **Episode** - A collection of steps that terminates when the agent fails to meet the environment's objective, or the episode reaches the maximum number of allowed steps.
- **Render** - Gym can render one frame for display after each episode.
- **Reward** - A positive reinforcement that can occur at the end of each episode, after the agent acts.
- **Nondeterministic** - For some environments, randomness is a factor in deciding what effects actions have on reward and changes to the observation space.

It is important to note that many of the gym environments specify that they are not nondeterministic even though they make use of random numbers to process actions. It is generally agreed upon (based on the gym GitHub issue tracker) that nondeterministic property means that a deterministic environment will still behave randomly even when given consistent seed value. The seed method of an environment can be used by the program to seed the random number generator for the environment.

The Gym library allows us to query some of these attributes from environments. I created the following function to query gym environments.

Code

```
import gym

def query_environment(name):
  env = gym.make(name)
  spec = gym.spec(name)
  print(f"Action Space: {env.action_space}")
  print(f"Observation Space: {env.observation_space}")
  print(f"Max Episode Steps: {spec.max_episode_steps}")
  print(f"Nondeterministic: {spec.nondeterministic}")
  print(f"Reward Range: {env.reward_range}")
  print(f"Reward Threshold: {spec.reward_threshold}")
```

We will begin by looking at the MountainCar-v0 environment, which challenges an underpowered car to escape the valley between two mountains. The following code describes the Mountian Car environment.

Code

```
query_environment("MountainCar-v0")
```

Output

```
Action Space: Discrete(3)
Observation Space: Box(2,)
```

```
Max Episode Steps: 200
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: -110.0
```

There are three distinct actions that can be taken: accelrate forward, decelerate, or accelerate backwards. The observation space contains two continuous (floating point) values, as evident by the box object. The observation space is simply the position and velocity of the car. The car has 200 steps to escape for each epasode. You would have to look at the code to know, but the mountian car recieves no incramental reward. The only reward for the car is given when it escapes the valley.

Code

```
query_environment("CartPole-v1")
```

Output

```
Action Space: Discrete(2)
Observation Space: Box(4,)
Max Episode Steps: 500
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 475.0
```

The CartPole-v1 environment challenges the agent to move a cart while keeping a pole balanced. The environment has an observation space of 4 continuous numbers:

- Cart Position
- Cart Velocity
- Pole Angle
- Pole Velocity At Tip

To achieve this goal, the agent can take the following actions:

- Push cart to the left
- Push cart to the right

There is also a continuous variant of the mountain car. This version does not simply have the motor on or off. For the continuous car the action space is a single floating point number that specifies how much forward or backward force is being applied.

Code

```
query_environment("MountainCarContinuous-v0")
```

Output

```
Action Space: Box(1,)
Observation Space: Box(2,)
```

```
Max Episode Steps: 999
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 90.0
```

Note: ignore the warning above, it is a relativly inconsequential bug in OpenAI Gym.

Atari games, like breakout can use an observation space that is either equal to the size of the Atari screen (210x160) or even use the RAM memory of the Atari (128 bytes) to determine the state of the game. Yes thats bytes, not kilobytes!

Code

```
query_environment("Breakout-v0")
```

Output

```
Action Space: Discrete(4)
Observation Space: Box(210, 160, 3)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

Code

```
query_environment("Breakout-ram-v0")
```

Output

```
Action Space: Discrete(4)
Observation Space: Box(128,)
Max Episode Steps: 10000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

12.1.3 Render OpenAI Gym Environments from CoLab

It is possible to visualize the game your agent is playing, even on CoLab. This section provides information on how to generate a video in CoLab that shows you an episode of the game your agent is playing. This video process is based on suggestions found here.

Begin by installing **pyvirtualdisplay** and **python-opengl**.

Code

```
!pip install gym pyvirtualdisplay > /dev/null 2>&1
```

```
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
```

Next, we install needed requirements to display an Atari game.

Code

```
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
```

Output

```
Requirement already up-to-date: setuptools in
/usr/local/lib/python3.6/dist-packages (46.1.3)
```

Next we define functions used to show the video by adding it to the CoLab notebook.

Code

```
import gym
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
Utility functions to enable video recording of gym environment
and displaying it.
To enable video, just do "env = wrap_env(env)"
"""

def show_video():
  mp4list = glob.glob('video/*.mp4')
  if len(mp4list) > 0:
    mp4 = mp4list[0]
    video = io.open(mp4, 'r+b').read()
    encoded = base64.b64encode(video)
    ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                loop controls style="height: 400px;">
                <source src="data:video/mp4; base64,{0}" type="video/mp4" />
```

```
                </video>'''.format(encoded.decode('ascii'))))
  else:
    print("Could not find video")


def wrap_env(env):
  env = Monitor(env, './video', force=True)
  return env
```

Now we are ready to play the game. We use a simple random agent.

Code

```
#env = wrap_env(gym.make("MountainCar-v0"))
env = wrap_env(gym.make("Atlantis-v0"))

observation = env.reset()

while True:

    env.render()

    #your agent goes here
    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)


    if done:
      break;

env.close()
show_video()
```

Output

12.2 Part 12.2: Introduction to Q-Learning

Q-Learning is a foundational technique upon which deep reinforcement learning is based. Before we explore deep reinforcement learning, it is essential to understand Q-Learning. Several components make up any Q-Learning system.

- **Agent** - The agent is an entity that exists in an environment that takes actions to affect the state of the environment, to receive rewards.
- **Environment** - The environment is the universe that the agent exists in. The environment is always in a specific state that is changed by the actions of the agent.
- **Actions** - Steps that can be performed by the agent to alter the environment

- **Step** - A step occurs each time that the agent performs an action and potentially changes the environment state.
- **Episode** - A chain of steps that ultimately culminates in the environment entering a terminal state.
- **Epoch** - A training iteration of the agent that contains some number of episodes.
- **Terminal State** - A state in which further actions do not make sense. In many environments, a terminal state occurs when the agent has one, lost, or the environment exceeding the maximum number of steps.

Q-Learning works by building a table that suggests an action for every possible state. This approach runs into several problems. First, the environment is usually composed of several continuous numbers, resulting in an infinite number of states. Q-Learning handles continuous states by binning these numeric values into ranges.

Additionally, Q-Learning primarily deals with discrete actions, such as pressing a joystick up or down. Out of the box, Q-Learning does not deal with continuous inputs, such as a car's accelerator that can be in a range of positions from released to fully engaged. Researchers have come up with clever tricks to allow Q-Learning to accommodate continuous actions.

In the next chapter, we will learn more about deep reinforcement learning. Deep neural networks can help to solve the problems of continuous environments and action spaces. For now, we will apply regular Q-Learning to the Mountain Car problem from OpenAI Gym.

12.2.1 Introducing the Mountain Car

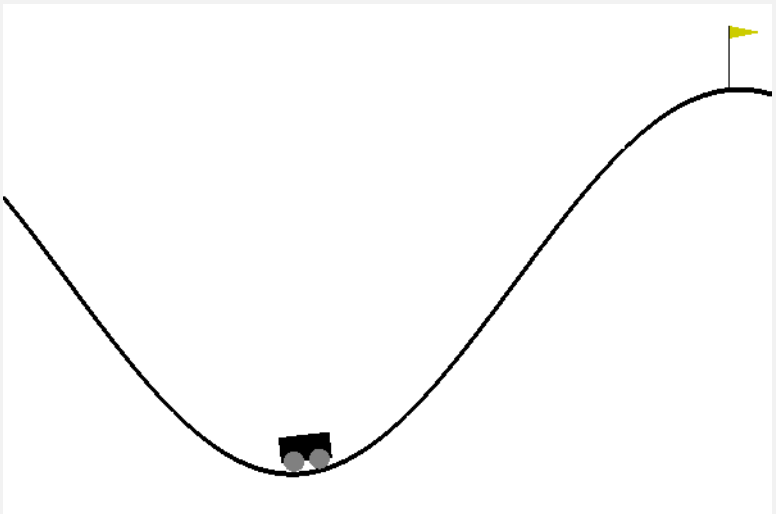
This section will demonstrate how Q-Learning can create a solution to the mountain car gym environment. The Mountain car is an environment where a car must climb a mountain. Because gravity is stronger than the car's engine, even with full throttle, it cannot merely accelerate up the steep slope. The vehicle is situated in a valley and must learn to utilize potential energy by driving up the opposite hill before the car can make it to the goal at the top of the rightmost hill.

First, it might be helpful to visualize the mountain car environment. The following code shows this environment. This code makes use of TF-Agents to perform this render. Usually, we use TF-Agents for the type of deep reinforcement learning that we will see in the next module. However, for now, TF-Agents is just used to render the mountain care environment.

Code

```
import tf_agents
from tf_agents.environments import suite_gym
import PIL.Image
import pyvirtualdisplay

display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()

env_name = 'MountainCar-v0'
env = suite_gym.load(env_name)
env.reset()
PIL.Image.fromarray(env.render())
```

Output



The mountain car environment provides the following discrete actions:

- 0 - Apply left force
- 1 - Apply no force
- 2 - Apply right force

The mountain car environment is made up of the following continuous values:

- state[0] - Position
- state[1] - Velocity

The following code shows an agent that applies full throttle to climb the hill. The cart is not strong enough. It will need to use potential energy from the mountain behind it.

Code

```
import gym
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
Utility functions to enable video recording of gym environment
and displaying it.
To enable video, just do "env = wrap_env(env)"
"""
```

```
def show_video():
  mp4list = glob.glob('video/*.mp4')
  if len(mp4list) > 0:
    mp4 = mp4list[0]
    video = io.open(mp4, 'r+b').read()
    encoded = base64.b64encode(video)
    ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                loop controls style="height: 400px;">
                <source src="data:video/mp4; base64,{0}" type="video/mp4" />
             </video>'''.format(encoded.decode('ascii'))))
  else:
    print("Could not find video")


def wrap_env(env):
  env = Monitor(env, './video', force=True)
  return env
```

Code

```
import gym

if COLAB:
  env = wrap_env(gym.make("MountainCar-v0"))
else:
  env = gym.make("MountainCar-v0")

env.reset()
done = False

i = 0
while not done:
    i += 1
    state, reward, done, _ = env.step(2)
    env.render()
    print(f"Step {i}: State={state}, Reward={reward}")

env.close()
```

Output

```
Step 1: State=[-0.55754837  0.00126361], Reward=-1.0
Step 2: State=[-0.55503058  0.00251779], Reward=-1.0
Step 3: State=[-0.5512774   0.00375318], Reward=-1.0
Step 4: State=[-0.54631687  0.00496053], Reward=-1.0
Step 5: State=[-0.54018608  0.00613078], Reward=-1.0
Step 6: State=[-0.53293095  0.00725514], Reward=-1.0
```

```
Step 7: State=[-0.52460583  0.00832512], Reward=-1.0
Step 8: State=[-0.51527315  0.00933267], Reward=-1.0
Step 9: State=[-0.50500292  0.01027024], Reward=-1.0
Step 10: State=[-0.49387208  0.01113084], Reward=-1.0
Step 11: State=[-0.48196389  0.01190819], Reward=-1.0
Step 12: State=[-0.46936715  0.01259674], Reward=-1.0
Step 13: State=[-0.45617536  0.01319179], Reward=-1.0
Step 14: State=[-0.44248581  0.01368956], Reward=-1.0
Step 15: State=[-0.42839861  0.01408719], Reward=-1.0

...

Step 196: State=[-0.20218851 -0.00198451], Reward=-1.0
Step 197: State=[-0.20522704 -0.00303853], Reward=-1.0
Step 198: State=[-0.20930653 -0.00407949], Reward=-1.0
Step 199: State=[-0.21440914 -0.00510261], Reward=-1.0
Step 200: State=[-0.22051217 -0.00610302], Reward=-1.0
```

Code

```
show_video()
```

Output

12.2.2 Programmed Car

Now we will look at a car that I hand-programmed. This car is straightforward; however, it solves the problem. The programmed car always applies force to one direction or another. It does not break. Whatever direction the vehicle is currently rolling, the agent uses power in that direction. Therefore, the car begins to climb a hill, is overpowered, and turns backward. However, once it starts to roll backward force is immediately applied in this new direction.

The following code implements this preprogrammed car.

Code

```
import gym

if COLAB:
  env = wrap_env(gym.make("MountainCar-v0"))
else:
  env = gym.make("MountainCar-v0")

state = env.reset()
done = False

i = 0
while not done:
```

```
    i += 1

    if state[1]>0:
        action = 2
    else:
        action = 0

    state, reward, done, _ = env.step(action)
    env.render()
    print(f"Step {i}: State={state}, Reward={reward}")

env.close()
```

Output

```
Step 1: State=[-0.57730941 -0.00060338], Reward=-1.0
Step 2: State=[-0.5785117  -0.00120229], Reward=-1.0
Step 3: State=[-0.580304   -0.0017923], Reward=-1.0
Step 4: State=[-0.58267307 -0.00236906], Reward=-1.0
Step 5: State=[-0.58560139 -0.00292832], Reward=-1.0
Step 6: State=[-0.58906736 -0.00346598], Reward=-1.0
Step 7: State=[-0.59304548 -0.00397811], Reward=-1.0
Step 8: State=[-0.5975065  -0.00446102], Reward=-1.0
Step 9: State=[-0.60241775 -0.00491125], Reward=-1.0
Step 10: State=[-0.60774335 -0.0053256 ], Reward=-1.0
Step 11: State=[-0.61344454 -0.00570119], Reward=-1.0
Step 12: State=[-0.61948002 -0.00603548], Reward=-1.0
Step 13: State=[-0.62580627 -0.00632625], Reward=-1.0
Step 14: State=[-0.63237791 -0.00657165], Reward=-1.0
Step 15: State=[-0.63914812 -0.00677021], Reward=-1.0

...

Step 149: State=[0.30975487 0.04947665], Reward=-1.0
Step 150: State=[0.35873547 0.0489806 ], Reward=-1.0
Step 151: State=[0.40752939 0.04879392], Reward=-1.0
Step 152: State=[0.45647027 0.04894088], Reward=-1.0
Step 153: State=[0.50591109 0.04944082], Reward=-1.0
```

We now visualize the preprogrammed car solving the problem.

Code

```
show_video()
```

Output

12.2.3 Reinforcement Learning

Q-Learning is a system of rewards that the algorithm gives an agent for successfully moving the environment into a state considered successful. These rewards are the Q-values from which this algorithm takes its name. The final output from the Q-Learning algorithm is a table of Q-values that indicate the reward value of every action that the agent can take, given every possible environment state. The agent must bin continuous state values into a fixed finite number of columns.

Learning occurs when the algorithm runs the agent and environment through a series of episodes and updates the Q-values based on the rewards received from actions taken; Figure 12.1 provides a high-level overview of this reinforcement or Q-Learning loop.

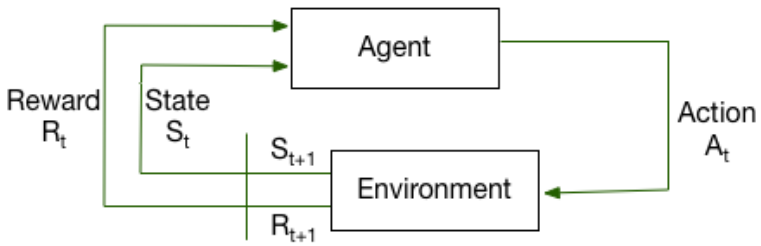


Figure 12.1: Reinforcement/Q Learning

The Q-values can dictate action by selecting the action column with the highest Q-value for the current environment state. The choice between choosing a random action and a Q-value driven action is governed by the epsilon (ϵ) parameter, which is the probability of random action.

Each time through the training loop, the training algorithm updates the Q-values according to the following equation.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

There are several parameters in this equation:
* alpha (α) - The learning rate, how much should the current step cause the Q-values to be updated.
* lambda (λ) - The discount factor is the percentage of future reward that the algorithm should consider in this update.

This equation modifies several values:

* $Q(s_t, a_t)$ - The Q-table. For each combination of states, what reward would the agent likely receive for performing each action?
* s_t - The current state.
* r_t - The last reward received.
* a_t - The action that the agent will perform.

The equation works by calculating a delta (temporal difference) that the equation should apply to the old state. This learning rate (α) scales this delta. A learning rate of 1.0 would fully implement the temporal difference to the Q-values each iteration and would likely be very chaotic.

There are two parts to the temporal difference: the new and old values. The new value is subtracted from the old value to provide a delta; the full amount that we would change the Q-value by if the learning rate did not scale this value. The new value is a summation of the reward received from the last action and the maximum of the Q-values from the resulting state when the client takes this action. It is essential to add the maximum of action Q-values for the new state because it estimates the optimal future values from proceeding with this action.

Q-Learning Car

We will now use Q-Learning to produce a car that learns to drive itself. Look out, Tesla! We begin by defining two essential functions.

Code

```
import gym
import numpy as np

# This function converts the floating point state values into
# discrete values. This is often called binning. We divide
# the range that the state values might occupy and assign
# each region to a bucket.
def calc_discrete_state(state):
    discrete_state = (state - env.observation_space.low)/buckets
    return tuple(discrete_state.astype(np.int))

# Run one game. The q_table to use is provided. We also
# provide a flag to indicate if the game should be
# rendered/animated. Finally, we also provide
# a flag to indicate if the q_table should be updated.
def run_game(q_table, render, should_update):
    done = False
    discrete_state = calc_discrete_state(env.reset())
    success = False

    while not done:
        # Exploit or explore
        if np.random.random() > epsilon:
            # Exploit - use q-table to take current best action
            # (and probably refine)
            action = np.argmax(q_table[discrete_state])
        else:
            # Explore - t
            action = np.random.randint(0, env.action_space.n)

        # Run simulation step
        new_state, reward, done, _ = env.step(action)

        # Convert continuous state to discrete
```

```
        new_state_disc = calc_discrete_state(new_state)

        # Have we reached the goal position (have we won?)?
        if new_state[0] >= env.unwrapped.goal_position:
            success = True

        # Update q-table
        if should_update:
            max_future_q = np.max(q_table[new_state_disc])
            current_q = q_table[discrete_state + (action,)]
            new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE * \
              (reward + DISCOUNT * max_future_q)
            q_table[discrete_state + (action,)] = new_q

        discrete_state = new_state_disc

        if render:
            env.render()

    return success
```

Several hyperparameters are very important for Q-Learning. These parameters will likely need adjustment as you apply Q-Learning to other problems. Because of this, it is crucial to understand the role of each parameter.

- **LEARNING_RATE** The rate at which previous Q-values are updated based on new episodes run during training.
- **DISCOUNT** The amount of significance to give estimates of future rewards when added to the reward for the current action taken. A value of 0.95 would indicate a discount of 5% to the future reward estimates.
- **EPISODES** The number of episodes to train over. Increase this for more complex problems; however, training time also increases.
- **SHOW_EVERY** How many episodes to allow to elapse before showing an update.
- **DISCRETE_GRID_SIZE** How many buckets to use when converting each of the continuous state variables. For example, [10, 10] indicates that the algorithm should use ten buckets for the first and second state variables.
- **START_EPSILON_DECAYING** Epsilon is the probability that the agent will select a random action over what the Q-Table suggests. This value determines the starting probability of randomness.
- **END_EPSILON_DECAYING** How many episodes should elapse before epsilon goes to zero and no random actions are permitted. For example, EPISODES//10 means only the first 1/10th of the episodes might have random actions.

Code

```
LEARNING_RATE = 0.1
DISCOUNT = 0.95
EPISODES = 50000
SHOW_EVERY = 1000
```

```
DISCRETE_GRID_SIZE = [10, 10]
START_EPSILON_DECAYING = 0.5
END_EPSILON_DECAYING = EPISODES//10
```

We can now make the environment. If we are running in Google COLAB then we wrap the environment to be displayed inside the web browser. Next create the discrete buckets for state and build Q-table.

Code

```
if COLAB:
  env = wrap_env(gym.make("MountainCar-v0"))
else:
  env = gym.make("MountainCar-v0")

epsilon = 1
epsilon_change = epsilon/(END_EPSILON_DECAYING - START_EPSILON_DECAYING)
buckets = (env.observation_space.high - env.observation_space.low) \
    /DISCRETE_GRID_SIZE
q_table = np.random.uniform(low=-3, high=0, size=(DISCRETE_GRID_SIZE \
    + [env.action_space.n]))
success = False
```

We can now make the environment. If we are running in Google COLAB then we wrap the environment to be displayed inside the web browser. Next, create the discrete buckets for state and build Q-table.

Code

```
episode = 0
success_count = 0

# Loop through the required number of episodes
while episode<EPISODES:
    episode+=1
    done = False

    # Run the game.  If we are local, display render animation at SHOW_EVERY
    # intervals.
    if episode % SHOW_EVERY == 0:
        print(f"Current␣episode:␣{episode},␣success:␣{success_count}" +\
              "␣({float(success_count)/SHOW_EVERY})")
        success = run_game(q_table, True, False)
        success_count = 0
    else:
        success = run_game(q_table, False, True)

    # Count successes
    if success:
        success_count += 1
```

```
    # Move epsilon towards its ending value, if it still needs to move
    if END_EPSILON_DECAYING >= episode >= START_EPSILON_DECAYING:
        epsilon = max(0, epsilon - epsilon_change)

print(success)
```

Output

```
Current episode: 1000, success: 0 (0.0)
Current episode: 2000, success: 0 (0.0)
Current episode: 3000, success: 0 (0.0)
Current episode: 4000, success: 29 (0.029)
Current episode: 5000, success: 345 (0.345)
Current episode: 6000, success: 834 (0.834)
Current episode: 7000, success: 797 (0.797)
Current episode: 8000, success: 679 (0.679)
Current episode: 9000, success: 600 (0.6)
Current episode: 10000, success: 728 (0.728)
Current episode: 11000, success: 205 (0.205)
Current episode: 12000, success: 612 (0.612)
Current episode: 13000, success: 733 (0.733)
Current episode: 14000, success: 1000 (1.0)
Current episode: 15000, success: 998 (0.998)

...

Current episode: 47000, success: 1000 (1.0)
Current episode: 48000, success: 1000 (1.0)
Current episode: 49000, success: 1000 (1.0)
Current episode: 50000, success: 1000 (1.0)
True
```

As you can see, the number of successful episodes generally increases as training progresses. It is not advisable to stop the first time that we observe 100% success over 1,000 episodes. There is a randomness to most games, so it is not likely that an agent would retain its 100% success rate with a new run. Once you observe that the agent has gotten 100% for several update intervals, it might be safe to stop training.

12.2.4 Running and Observing the Agent

Now that the algorithm has trained the agent, we can observe the agent in action. You can use the following code to see the agent in action.

Code

```
run_game(q_table, True, False)
show_video()
```

Output

12.2.5 Inspecting the Q-Table

We can also display the Q-table. The following code shows the action that the agent would perform for each environment state. As the weights of a neural network, this table is not straightforward to interpret. Some patterns do emerge in that directions do arise, as seen by calculating the means of rows and columns. The actions seem consistent at upper and lower halves of both velocity and position.

Code

```
import pandas as pd

df = pd.DataFrame(q_table.argmax(axis=2))

df.columns = [f'v-{x}' for x in range(DISCRETE_GRID_SIZE[0])]
df.index = [f'p-{x}' for x in range(DISCRETE_GRID_SIZE[1])]
df
```

Output

	v-0	v-1	v-2	v-3	v-4	v-5	v-6	v-7	v-8	v-9

Code

```
df.mean(axis=0)
```

Output

```
v-0    0.9
v-1    0.5
v-2    0.6
v-3    0.7
v-4    0.6
v-5    1.0
v-6    1.7
v-7    1.3
v-8    1.5
v-9    1.2
dtype: float64
```

Code

```
df.mean(axis=1)
```

Output

```
p-0    1.3
p-1    1.4
p-2    1.3
p-3    1.0
p-4    0.6
p-5    0.9
p-6    0.7
p-7    0.5
p-8    1.2
p-9    1.1
dtype: float64
```

12.3 Part 12.3: Keras Q-Learning in the OpenAI Gym

Q-Learning, as we covered in the previous part, is a robust machine learning algorithm. Unfortunately, Q-Learning requires that the Q-table contain an entry for every possible state that the environment can take. If the environment only includes a handful of discrete state elements, then traditional Q-learning might be a good learning algorithm. However, if the state space is large, the Q-table can become prohibitively large.

Creating policies for large state spaces is a task that Deep Q-Learning Networks (DQN) can usually handle. Unlike a table, a neural network does not require the program to represent every combination of state and action. Neural networks can generalize these states and learn commonalities. A DQN maps the state to its input neurons and the action Q-values to the output neurons. The DQN effectively becomes a function that accepts state and suggestions an action by returning the expected reward for each of the possible actions. Figure 12.2 demonstrates the DQN structure and mapping between state and action.

As this diagram illustrates, the environment state contains several elements. For the basic DQN the state can be a mix of continuous and categorical/discrete values. For the DQN, the discrete state elements the program typically encoded as dummy variables. The actions should be discrete when your program implements a DQN. Other algorithms support continuous outputs, which we will discuss later in this chapter.

In this chapter, we will make use of TF-Agents to implement a DQN to solve the cart-pole environment. TF-Agents makes designing, implementing, and testing new RL algorithms easier by providing well tested modular components that can be modified and extended. It enables fast code iteration, with functional test integration and benchmarking.

12.3.1 DQN and the Cart-Pole Problem

Barto (1983) first described the cart-pole problem.[2]A cart is connected to a hinged rigid pole. The cart is free to move only in the vertical plane of the cart/track. The agent can apply an implusive "left" or "right" force F of a fixed magnitude to the cart at discrete time intervals. The cart-pole environment simulates the physics behind keeping the pole in a reasonably upright position on the cart. The environment has four state variables:

- x The position of the cart on the track.
- θ The angle of the pole with the vertical

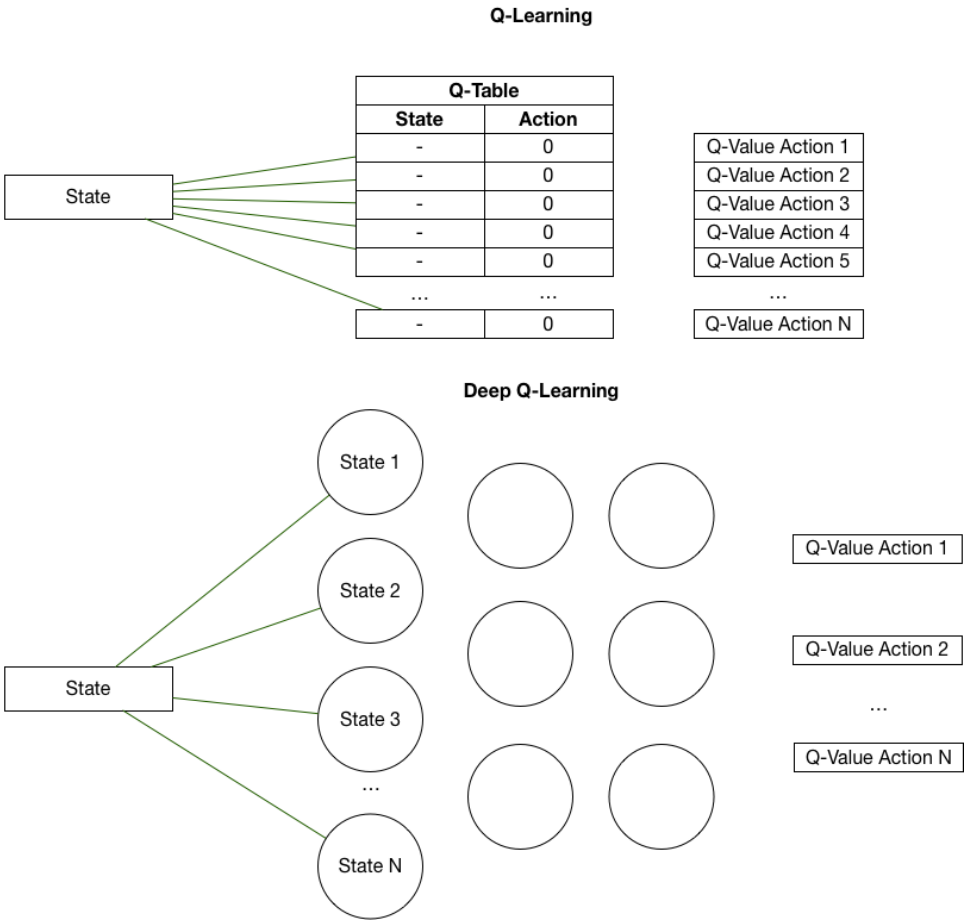


Figure 12.2: Deep Q-Learning (DQL)

- $\dot{x}$ The cart velocity.
- $\dot{\theta}$ The rate of change of the angle.

The action space consists of discrete actions:

- Apply force left
- Apply force right

To apply DQN to this problem, you need to create the following components for TF-Agents.

- Environment
- Agent
- Policies
- Metrics and Evaluation
- Replay Buffer
- Data Collection

- Training

These components are standard in most DQN implementations. Later, we will apply these same components to an Atari game, and after that, a problem of our design. This example is based on the cart-pole tutorial provided for TF-Agents. We begin by importing needed Python libraries.

Code

```
import base64
import imageio
import IPython
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image
import pyvirtualdisplay

import tensorflow as tf

from tf_agents.agents.dqn import dqn_agent
from tf_agents.drivers import dynamic_step_driver
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.eval import metric_utils
from tf_agents.metrics import tf_metrics
from tf_agents.networks import q_network
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.utils import common
```

Code

```
# Set up a virtual display for rendering OpenAI gym environments.
display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

12.3.2 Hyperparameters

Several hyperparameters must be defined. The TF-Agent example provided reasonably well-tuned hyperparameters for cart-pole. Later we will adapt these to an Atari game.

Code

```
# How long should training run?
num_iterations = 20000
# How many initial random steps, before training start, to
# collect initial data.
initial_collect_steps = 1000
# How many steps should we run each iteration to collect
```

```
# data from.
collect_steps_per_iteration = 1
# How much data should we store for training examples.
replay_buffer_max_length = 100000

batch_size = 64
learning_rate = 1e-3
# How often should the program provide an update.
log_interval = 200

# How many episodes should the program use for each evaluation.
num_eval_episodes = 10
# How often should an evaluation occur.
eval_interval = 1000
```

12.3.3 Environment

TF-Agents uses OpenAI gym environments to represent the task or problem to be solved. Standard environments can be created in TF-Agents using **tf_agents.environments** suites. TF-Agents has suites for loading environments from sources such as the OpenAI Gym, Atari, and DM Control. We begin by loading the CartPole environment from the OpenAI Gym suite.

Code

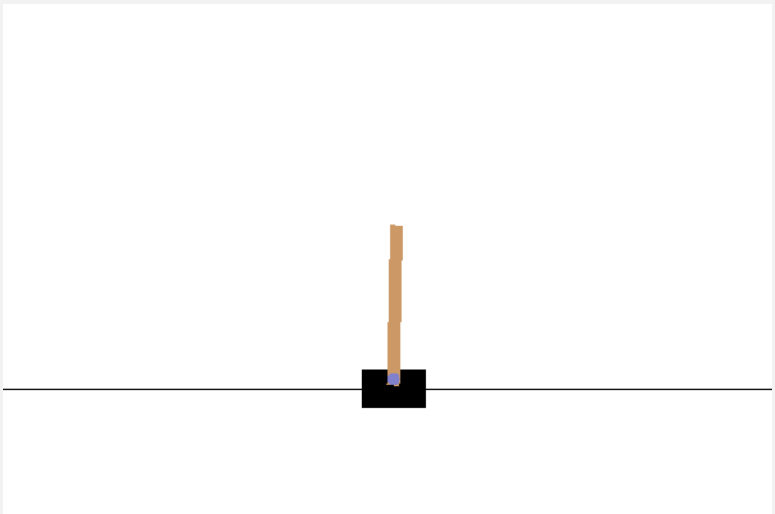
```
env_name = 'CartPole-v0'
env = suite_gym.load(env_name)
```

We will quickly render this environment to see the visual representation.

Code

```
env.reset()
PIL.Image.fromarray(env.render())
```

Output



The **environment.step** method takes an **action** in the environment and returns a **TimeStep** tuple containing the next observation of the environment and the reward for the action.

The **time_step_spec()** method returns the specification for the **TimeStep** tuple. Its **observation** attribute shows the shape of observations, the data types, and the ranges of allowed values. The **reward** attribute shows the same details for the reward.

Code

```
print('Observation Spec:')
print(env.time_step_spec().observation)
```

Output

```
Observation Spec:
BoundedArraySpec(shape=(4,), dtype=dtype('float32'),
name='observation', minimum=[-4.8000002e+00 -3.4028235e+38
-4.1887903e-01 -3.4028235e+38], maximum=[4.8000002e+00 3.4028235e+38
4.1887903e-01 3.4028235e+38])
```

Code

```
print('Reward Spec:')
print(env.time_step_spec().reward)
```

Output

```
Reward Spec:
ArraySpec(shape=(), dtype=dtype('float32'), name='reward')
```

The **action_spec()** method returns the shape, data types, and allowed values of valid actions.

Code

```
print('Action Spec:')
print(env.action_spec())
```

Output

```
Action Spec:
BoundedArraySpec(shape=(), dtype=dtype('int64'), name='action',
minimum=0, maximum=1)
```

In the Cartpole environment:

- `observation` is an array of 4 floats:
  - the position and velocity of the cart
  - the angular position and velocity of the pole
- `reward` is a scalar float value
- `action` is a scalar integer with only two possible values:
  - `0` --- "move left"
  - `1` --- "move right"

Code

```
time_step = env.reset()
print('Time step:')
print(time_step)

action = np.array(1, dtype=np.int32)

next_time_step = env.step(action)
print('Next time step:')
print(next_time_step)
```

Output

```
Time step:
TimeStep(step_type=array(0, dtype=int32), reward=array(0.,
dtype=float32), discount=array(1., dtype=float32),
observation=array([-0.00263771,  0.04141404, -0.02421604,
-0.02355336], dtype=float32))
Next time step:
TimeStep(step_type=array(1, dtype=int32), reward=array(1.,
dtype=float32), discount=array(1., dtype=float32),
observation=array([-0.00180943,  0.23687476, -0.02468711, -0.3237773
], dtype=float32))
```

Usually, the program instantiates two environments: one for training and one for evaluation.

Code

```
train_py_env = suite_gym.load(env_name)
eval_py_env = suite_gym.load(env_name)
```

The Cartpole environment, like most environments, is written in pure Python and is converted to TF-Agents and TensorFlow using the **TFPyEnvironment** wrapper. The original environment's API uses Numpy arrays. The **TFPyEnvironment** turns these to **Tensors** to make it compatible with Tensorflow agents and policies.

Code

```
train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
```

12.3.4 Agent

An Agent represents the algorithm used to solve an RL problem. TF-Agents provides standard implementations of a variety of Agents:

- DQN (used in this example)
- REINFORCE
- DDPG
- TD3
- PPO
- SAC.

You can only use the DQN agent in environments which have a discrete action space. The DQN makes use of a QNetwork, a neural network model that learns to predict Q-Values (expected returns) for all actions, given a state from the environment.

The following code uses **tf_agents.networks.q_network** to create a QNetwork, passing in the **observation_spec**, **action_spec**, and a tuple describing the number and size of the model's hidden layers.

Code

```
fc_layer_params = (100,)

q_net = q_network.QNetwork(
    train_env.observation_spec(),
    train_env.action_spec(),
    fc_layer_params=fc_layer_params)
```

Now we use **tf_agents.agents.dqn.dqn_agent** to instantiate a **DqnAgent**. In addition to the **time_step_spec**, **action_spec** and the QNetwork, the agent constructor also requires an optimizer (in this case, **AdamOptimizer**), a loss function, and an integer step counter.

Code

```
optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate)

train_step_counter = tf.Variable(0)

agent = dqn_agent.DqnAgent(
    train_env.time_step_spec(),
    train_env.action_spec(),
    q_network=q_net,
    optimizer=optimizer,
    td_errors_loss_fn=common.element_wise_squared_loss,
    train_step_counter=train_step_counter)

agent.initialize()
```

12.3.5 Policies

A policy defines the way an agent acts in an environment. Typically, the goal of reinforcement learning is to train the underlying model until the policy produces the desired outcome.

In this example:

- The desired outcome is keeping the pole balanced upright over the cart.
- The policy returns an action (left or right) for each `time_step` observation.

Agents contain two policies:

- **agent.policy** - The algorithm uses this main policy for evaluation and deployment.
- **agent.collect_policy** - The algorithm this secondary policy for data collection.

Code

```
eval_policy = agent.policy
collect_policy = agent.collect_policy
```

Policies can be created independently of agents. For example, use **tf_agents.policies.random_tf_policy** to create a policy which will randomly select an action for each **time_step**. We will use this random policy to create initial collection data to begin training.

Code

```
random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())
```

To get an action from a policy, call the **policy.action(time_step)** method. The **time_step** contains the observation from the environment. This method returns a **PolicyStep**, which is a named tuple with three components:

- **action** - The action to be taken (in this case, 0 or 1).

- **state** - Used for stateful (that is, RNN-based) policies.
- **info** - Auxiliary data, such as log probabilities of actions.

Next we create an environment and setup the random policy.

Code

```
example_environment = tf_py_environment.TFPyEnvironment(
    suite_gym.load('CartPole-v0'))
time_step = example_environment.reset()
random_policy.action(time_step)
```

Output

```
PolicyStep(action=<tf.Tensor: shape=(1,), dtype=int64,
numpy=array([0])>, state=(), info=())
```

12.3.6 Metrics and Evaluation

The most common metric used to evaluate a policy is the average return. The return is the sum of rewards obtained while running a policy in an environment for an episode. Several episodes are run, creating an average return. The following function computes the average return of a policy, given the policy, environment, and a number of episodes. We will use this same evaluation for Atari.

Code

```
def compute_avg_return(environment, policy, num_episodes=10):

  total_return = 0.0
  for _ in range(num_episodes):

    time_step = environment.reset()
    episode_return = 0.0

    while not time_step.is_last():
      action_step = policy.action(time_step)
      time_step = environment.step(action_step.action)
      episode_return += time_step.reward
    total_return += episode_return

  avg_return = total_return / num_episodes
  return avg_return.numpy()[0]


# See also the metrics module for standard implementations
# of different metrics.
# https://github.com/tensorflow/agents/tree/master/tf_agents/metrics
```

Running this computation on the `random_policy` shows a baseline performance in the environment.

Code

```
compute_avg_return(eval_env, random_policy, num_eval_episodes)
```

Output

```
19.8
```

12.3.7 Replay Buffer

The replay buffer keeps track of data collected from the environment. This tutorial uses **TFUniformReplayBuffer**. The constructor requires the specs for the data it will be collecting. This value is available from the agent using the **collect_data_spec** method. The batch size and maximum buffer length are also required.

Code

```
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=train_env.batch_size,
    max_length=replay_buffer_max_length)
```

For most agents, **collect_data_spec** is a named tuple called **Trajectory**, containing the specs for observations, actions, rewards, and other items.

Code

```
agent.collect_data_spec
```

Output

```
Trajectory(step_type=TensorSpec(shape=(), dtype=tf.int32,
name='step_type'), observation=BoundedTensorSpec(shape=(4,),
dtype=tf.float32, name='observation', minimum=array([-4.8000002e+00,
-3.4028235e+38, -4.1887903e-01, -3.4028235e+38],
      dtype=float32), maximum=array([4.8000002e+00, 3.4028235e+38,
4.1887903e-01, 3.4028235e+38],
      dtype=float32)), action=BoundedTensorSpec(shape=(),
dtype=tf.int64, name='action', minimum=array(0), maximum=array(1)),
policy_info=(), next_step_type=TensorSpec(shape=(), dtype=tf.int32,
name='step_type'), reward=TensorSpec(shape=(), dtype=tf.float32,
name='reward'), discount=BoundedTensorSpec(shape=(), dtype=tf.float32,
name='discount', minimum=array(0., dtype=float32), maximum=array(1.,
dtype=float32)))
```

12.3.8 Data Collection

Now execute the random policy in the environment for a few steps, recording the data in the replay buffer.

Code

```
def collect_step(environment, policy, buffer):
  time_step = environment.current_time_step()
  action_step = policy.action(time_step)
  next_time_step = environment.step(action_step.action)
  traj = trajectory.from_transition(time_step, action_step, next_time_step)

  # Add trajectory to the replay buffer
  buffer.add_batch(traj)

def collect_data(env, policy, buffer, steps):
  for _ in range(steps):
    collect_step(env, policy, buffer)

collect_data(train_env, random_policy, replay_buffer, steps=100)

# This loop is so common in RL, that we provide standard implementations.
# For more details see the drivers module.
# https://www.tensorflow.org/agents/api_docs/python/tf_agents/drivers
```

The replay buffer is now a collection of Trajectories. The agent needs access to the replay buffer. TF-Agents provides this access by creating an iterable **tf.data.Dataset** pipeline, which will feed data to the agent.

Each row of the replay buffer only stores a single observation step. But since the DQN Agent needs both the current and next observation to compute the loss, the dataset pipeline will sample two adjacent rows for each item in the batch (**num_steps=2**).

The program also optimizes this dataset by running parallel calls and prefetching data.

Code

```
# Dataset generates trajectories with shape [Bx2x...]
dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size,
    num_steps=2).prefetch(3)


dataset
```

Output

```
<PrefetchDataset shapes: (Trajectory(step_type=(64, 2),
observation=(64, 2, 4), action=(64, 2), policy_info=(),
next_step_type=(64, 2), reward=(64, 2), discount=(64, 2)),
BufferInfo(ids=(64, 2), probabilities=(64,))), types:
```

```
(Trajectory(step_type=tf.int32, observation=tf.float32,
action=tf.int64, policy_info=(), next_step_type=tf.int32,
reward=tf.float32, discount=tf.float32), BufferInfo(ids=tf.int64,
probabilities=tf.float32))>
```

Code

```
iterator = iter(dataset)

print(iterator)
```

Output

```
<tensorflow.python.data.ops.iterator_ops.OwnedIterator object at
0x7f478c078470>
```

12.3.9 Training the agent

Two things must happen during the training loop:

- Collect data from the environment
- Use that data to train the agent's neural network(s)

This example also periodically evaluates the policy and prints the current score.

The following will take ~5 minutes to run.

Code

```
# (Optional) Optimize by wrapping some of the code in a graph
# using TF function.
agent.train = common.function(agent.train)

# Reset the train step
agent.train_step_counter.assign(0)

# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, agent.policy, \
                                num_eval_episodes)
returns = [avg_return]

for _ in range(num_iterations):

  # Collect a few steps using collect_policy and save to the replay buffer.
  for _ in range(collect_steps_per_iteration):
    collect_step(train_env, agent.collect_policy, replay_buffer)

  # Sample a batch of data from the buffer and update the agent's network.
  experience, unused_info = next(iterator)
```

```
  train_loss = agent.train(experience).loss

  step = agent.train_step_counter.numpy()

  if step % log_interval == 0:
    print('step = {0}: loss = {1}'.format(step, train_loss))

  if step % eval_interval == 0:
    avg_return = compute_avg_return(eval_env, agent.policy, \
                                    num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
```

Output

```
step = 200: loss = 14.340982437133789
step = 400: loss = 10.511648178100586
step = 600: loss = 3.2730941772460938
step = 800: loss = 3.8935298919677734
step = 1000: loss = 14.826580047607422
step = 1000: Average Return = 9.100000381469727
step = 1200: loss = 9.10183048248291
step = 1400: loss = 10.874725341796875
step = 1600: loss = 4.35567045211792
step = 1800: loss = 8.448399543762207
step = 2000: loss = 7.585771560668945
step = 2000: Average Return = 20.200000762939453
step = 2200: loss = 7.654891014099121
step = 2400: loss = 11.028305053710938
step = 2600: loss = 12.608250617980957

...

step = 19400: loss = 28.240354537963867
step = 19600: loss = 60.595802307128906
step = 19800: loss = 56.309600830078125
step = 20000: loss = 62.594276428222656
step = 20000: Average Return = 200.0
```

12.3.10 Visualization

12.3.11 Plots

Use **matplotlib.pyplot** to chart how the policy improved during training.

One iteration of **Cartpole-v0** consists of 200 time steps. The environment gives a reward of `+1` for each step the pole stays up, so the maximum return for one episode is 200. The charts show the return increasing towards that maximum each time it is evaluated during training. (It may be a little unstable and

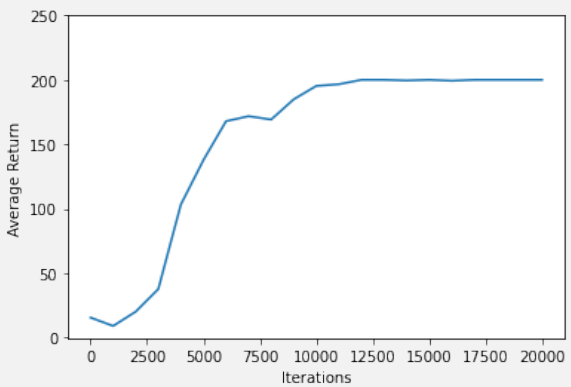
not increase each time monotonically.)

Code

```
iterations = range(0, num_iterations + 1, eval_interval)
plt.plot(iterations, returns)
plt.ylabel('Average Return')
plt.xlabel('Iterations')
plt.ylim(top=250)
```

Output

```
(-0.44499959945678746, 250.0)
```



12.3.12 Videos

Charts are nice. But more exciting is seeing an agent actually performing a task in an environment.

First, create a function to embed videos in the notebook.

Code

```
def embed_mp4(filename):
  """Embeds an mp4 file in the notebook."""
  video = open(filename,'rb').read()
  b64 = base64.b64encode(video)
  tag = '''
  <video width="640" height="480" controls>
    <source src="data:video/mp4; base64,{0}" type="video/mp4">
  Your browser does not support the video tag.
  </video>'''.format(b64.decode())

  return IPython.display.HTML(tag)
```

Now iterate through a few episodes of the Cartpole game with the agent. The underlying Python environment (the one "inside" the TensorFlow environment wrapper) provides a `render()` method, which outputs an image of the environment state. These can be collected into a video.

Code

```
def create_policy_eval_video(policy, filename, num_episodes=5, fps=30):
  filename = filename + ".mp4"
  with imageio.get_writer(filename, fps=fps) as video:
    for _ in range(num_episodes):
      time_step = eval_env.reset()
      video.append_data(eval_py_env.render())
      while not time_step.is_last():
        action_step = policy.action(time_step)
        time_step = eval_env.step(action_step.action)
        video.append_data(eval_py_env.render())
  return embed_mp4(filename)



create_policy_eval_video(agent.policy, "trained-agent")
```

Output

For fun, compare the trained agent (above) to an agent moving randomly. (It does not do as well.)

Code

```
create_policy_eval_video(random_policy, "random-agent")
```

Output

12.4 Part 12.4: Atari Games with Keras Neural Networks

The Atari 2600 is a home video game console from Atari, Inc. Released on September 11, 1977. It is credited with popularizing the use of microprocessor-based hardware and games stored on ROM cartridges instead of dedicated hardware with games physically built into the unit. The 2600 was bundled with two joystick controllers, a conjoined pair of paddle controllers, and a game cartridge: initially Combat, and later Pac-Man.

Atari emulators are popular and allow many of the old Atari video games to be played on modern computers. They are even available as JavaScript.

- Virtual Atari

Atari games have become popular benchmarks for AI systems, particularly reinforcement learning. OpenAI Gym internally uses the Stella Atari Emulator. The Atari 2600 is shown in Figure 12.3.

12.4.1 Actual Atari 2600 Specs

- CPU: 1.19 MHz MOS Technology 6507

Figure 12.3: The Atari 2600

- Audio + Video processor: Television Interface Adapter (TIA)
- Playfield resolution: 40 x 192 pixels (NTSC). Uses a 20-pixel register that is mirrored or copied, left side to right side, to achieve the width of 40 pixels.
- Player sprites: 8 x 192 pixels (NTSC). Player, ball, and missile sprites use pixels that are 1/4 the width of playfield pixels (unless stretched).
- Ball and missile sprites: 1 x 192 pixels (NTSC).
- Maximum resolution: 160 x 192 pixels (NTSC). Max resolution is only somewhat achievable with programming tricks that combine sprite pixels with playfield pixels.
- 128 colors (NTSC). 128 possible on screen. Max of 4 per line: background, playfield, player0 sprite, and player1 sprite. Palette switching between lines is common. Palette switching mid line is possible but not common due to resource limitations.
- 2 channels of 1-bit monaural sound with 4-bit volume control.

12.4.2 OpenAI Lab Atari Pong

OpenAI Gym can be used with Windows; however, it requires a special installation procedure.

This chapter demonstrates playing Atari Pong. Pong is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The goal is for each player to reach eleven points before the opponent; you earn points when one fails to return it to the other. For the Atari 2600 version of Pong, a computer player (controlled by the 2600) is the opposing player.

This section shows how to adapt TF-Agents to an Atari game. Some changes are necessary when compared to the pole-cart game presented earlier in this chapter. You can quickly adapt this example to any Atari game by simply changing the environment name. However, I tuned the code presented here for Pong, and it may not perform as well for other games. Some tuning will likely be necessary to produce a good agent for other games.

We begin by importing the needed Python packages.

Code

```
import base64
import imageio
import IPython
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image
import pyvirtualdisplay

import tensorflow as tf

from tf_agents.agents.dqn import dqn_agent
from tf_agents.drivers import dynamic_step_driver
from tf_agents.environments import suite_gym, suite_atari
from tf_agents.environments import tf_py_environment, batched_py_environment
from tf_agents.eval import metric_utils
from tf_agents.metrics import tf_metrics
from tf_agents.networks import q_network
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.utils import common

from tf_agents.specs import tensor_spec
from tf_agents.trajectories import time_step as ts
```

Code

```
# Set up a virtual display for rendering OpenAI gym environments.
display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

12.4.3 Hyperparameters

The hyperparameter names are the same as the previous DQN example; however, I tuned the numeric values for the more complex Atari game.

Code

```
num_iterations = 250000

initial_collect_steps = 200
collect_steps_per_iteration = 10
replay_buffer_max_length = 100000

batch_size =  32
learning_rate = 2.5e-3
```

```
log_interval =   5000

num_eval_episodes = 5
eval_interval = 25000
```

The algorithm needs more iterations for an Atari game. I also found that increasing the number of collection steps helped the algorithm to train.

12.4.4 Atari Environment's

You must handle Atari environments differently than games like cart-poll. Atari games typically use their 2D displays as the environment state. AI Gym represents Atari games as either a 3D (height by width by color) state spaced based on their screens, or a vector representing the state of the game's computer RAM. To preprocess Atari games for greater computational efficiency, we generally skip several frames, decrease the resolution, and discard color information. The following code shows how we can set up an Atari environment.

Code

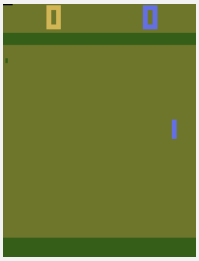
```
#env_name = 'Breakout-v4'
env_name = 'Pong-v0'
#env_name = 'BreakoutDeterministic-v4'
#env = suite_gym.load(env_name)

# AtariPreprocessing runs 4 frames at a time, max-pooling over the last 2
# frames. We need to account for this when computing things like update
# intervals.
ATARI_FRAME_SKIP = 4

max_episode_frames=108000  # ALE frames

env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames / ATARI_FRAME_SKIP,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)
#env = batched_py_environment.BatchedPyEnvironment([env])
```

We can now reset the environment and display one step. The following image shows how the Pong game environment appears to a user.

Code

```
env.reset()
PIL.Image.fromarray(env.render())
```

Output



We are now ready to load and wrap the two environments for TF-Agents. The algorithm uses the first environment for evaluation, and the second to train.

Code

```
train_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames / ATARI_FRAME_SKIP,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

eval_py_env = suite_atari.load(
    env_name,
    max_episode_steps=max_episode_frames / ATARI_FRAME_SKIP,
    gym_env_wrappers=suite_atari.DEFAULT_ATARI_GYM_WRAPPERS_WITH_STACKING)

train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
```

12.4.5 Agent

I used the following class, from TF-Agents examples, to wrap the regular Q-network class. The AtariQNetwork class ensures that the pixel values from the Atari screen are divided by 255. This division assists the neural network by normalizing the pixel values to between 0 and 1.

Code

```
class AtariQNetwork(q_network.QNetwork):
  """QNetwork subclass that divides observations by 255."""

  def call(self,
           observation,
           step_type=None,
           network_state=(),
           training=False):
    state = tf.cast(observation, tf.float32)
    # We divide the grayscale pixel values by 255 here rather than storing
    # normalized values beause uint8s are 4x cheaper to store than float32s.
    state = state / 255
    return super(AtariQNetwork, self).call(
        state, step_type=step_type, network_state=network_state,
```

```
        training=training)
```

Next, we introduce two hyperparameters that are specific to the neural network we are about to define.

Code

```
fc_layer_params = (512,)
conv_layer_params=((32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1))

q_net = AtariQNetwork(
            train_env.observation_spec(),
            train_env.action_spec(),
            conv_layer_params=conv_layer_params,
            fc_layer_params=fc_layer_params)
```

Convolutional neural networks usually are made up of several alternating pairs of convolution and max-pooling layers, ultimately culminating in one or more dense layers. These layers are the same types as previously seen in this course. The QNetwork accepts two parameters that define the convolutional neural network structure.

The more simple of the two parameters is **fc_layer_params**. This parameter specifies the size of each of the dense layers. A tuple specifies the size of each of the layers in a list.

The second parameter, named **conv_layer_params**, is a list of convolution layers parameters, where each item is a length-three tuple indicating (filters, kernel_size, stride). This implementation of QNetwork supports only convolution layers. If you desire a more complex convolutional neural network, you must define your variant of the QNetwork.

The QNetwork defined here is not the agent, instead, the QNetwork is used by the DQN agent to implement the actual neural network. This allows flexibility as you can set your own class if needed.

Next, we define the optimizer. For this example, I used RMSPropOptimizer. However, AdamOptimizer is another popular choice. We also create the DQN agent and reference the Q-network we just created.

Code

```
optimizer = tf.compat.v1.train.RMSPropOptimizer(
    learning_rate=learning_rate,
    decay=0.95,
    momentum=0.0,
    epsilon=0.00001,
    centered=True)

train_step_counter = tf.Variable(0)

observation_spec = tensor_spec.from_spec(train_env.observation_spec())
time_step_spec = ts.time_step_spec(observation_spec)

action_spec = tensor_spec.from_spec(train_env.action_spec())
target_update_period=32000  # ALE frames
update_period=16  # ALE frames
_update_period = update_period / ATARI_FRAME_SKIP
_global_step = tf.compat.v1.train.get_or_create_global_step()
```

```
agent = dqn_agent.DqnAgent(
    time_step_spec,
    action_spec,
    q_network=q_net,
    optimizer=optimizer,
    epsilon_greedy=0.01,
    n_step_update=1.0,
    target_update_tau=1.0,
    target_update_period=(
        target_update_period / ATARI_FRAME_SKIP / _update_period),
    td_errors_loss_fn=common.element_wise_huber_loss,
    gamma=0.99,
    reward_scale_factor=1.0,
    gradient_clipping=None,
    debug_summaries=False,
    summarize_grads_and_vars=False,
    train_step_counter=_global_step)



agent.initialize()
```

12.4.6 Metrics and Evaluation

There are many different ways to measure the effectiveness of a model trained with reinforcement learning. The loss function of the internal Q-network is not a good measure of the entire DQN algorithm's overall fitness. The network loss function measures how close the Q-network was fit to the collected data and did not indicate how effective the DQN is in maximizing rewards. The method used for this example tracks the average reward received over several episodes.

Code

```
def compute_avg_return(environment, policy, num_episodes=10):

  total_return = 0.0
  for _ in range(num_episodes):

    time_step = environment.reset()
    episode_return = 0.0

    while not time_step.is_last():
      action_step = policy.action(time_step)
      time_step = environment.step(action_step.action)
      episode_return += time_step.reward
    total_return += episode_return

  avg_return = total_return / num_episodes
```

```
  return avg_return.numpy()[0]


# See also the metrics module for standard implementations of
# different metrics.
# https://github.com/tensorflow/agents/tree/master/tf_agents/metrics
```

12.4.7 Replay Buffer

DQN works by training a neural network to predict the Q-values for every possible environment-state. A neural network needs training data, so the algorithm accumulates this training data as it runs episodes. The replay buffer is where this data is stored. Only the most recent episodes are stored, older episode data rolls off the queue as the queue accumulates new data.

Code

```
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=train_env.batch_size,
    max_length=replay_buffer_max_length)

# Dataset generates trajectories with shape [Bx2x...]
dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size,
    num_steps=2).prefetch(3)
```

12.4.8 Random Collection

The algorithm must prime the pump. Training cannot begin on an empty replay buffer. The following code performs a predefined number of steps to generate initial training data.

Code

```
random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())

def collect_step(environment, policy, buffer):
  time_step = environment.current_time_step()
  action_step = policy.action(time_step)
  next_time_step = environment.step(action_step.action)
  traj = trajectory.from_transition(time_step, action_step, next_time_step)

  # Add trajectory to the replay buffer
  buffer.add_batch(traj)

def collect_data(env, policy, buffer, steps):
  for _ in range(steps):
```

```
    collect_step(env, policy, buffer)

collect_data(train_env, random_policy, replay_buffer, \
             steps=initial_collect_steps)
```

12.4.9 Training the agent

We are now ready to train the DQN. This process can take many hours, depending on how many episodes you wish to run through. As training occurs, this code will update on both the loss and average return. As training becomes more successful, the average return should increase. The losses reported reflecting the average loss for individual training batches.

Code

```
iterator = iter(dataset)

# (Optional) Optimize by wrapping some of the code in a graph
# using TF function.
agent.train = common.function(agent.train)

# Reset the train step
agent.train_step_counter.assign(0)

# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, agent.policy, \
                                num_eval_episodes)
returns = [avg_return]

for _ in range(num_iterations):

  # Collect a few steps using collect_policy and save to the replay buffer.
  for _ in range(collect_steps_per_iteration):
    collect_step(train_env, agent.collect_policy, replay_buffer)

  # Sample a batch of data from the buffer and update the agent's network.
  experience, unused_info = next(iterator)
  train_loss = agent.train(experience).loss

  step = agent.train_step_counter.numpy()

  if step % log_interval == 0:
    print('step = {0}: loss = {1}'.format(step, train_loss))

  if step % eval_interval == 0:
    avg_return = compute_avg_return(eval_env, agent.policy, \
                                    num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
```

Output

```
step = 5000: loss = 0.005372311919927597
step = 10000: loss = 0.029342571273446083
step = 15000: loss = 0.023372460156679153
step = 20000: loss = 0.012967261485755444
step = 25000: loss = 0.03114483878016472
step = 25000: Average Return = -20.0
step = 30000: loss = 0.015883663669228554
step = 35000: loss = 0.022952664643526077
step = 40000: loss = 0.024018988013267517
step = 45000: loss = 0.015258202329277992
step = 50000: loss = 0.01642722450196743
step = 50000: Average Return = -18.399999618530273
step = 55000: loss = 0.024171829223632812
step = 60000: loss = 0.010190263390541077
step = 65000: loss = 0.005736709106713533

...

step = 235000: loss = 0.025019707158207893
step = 240000: loss = 0.02555653266608715
step = 245000: loss = 0.012253865599632263
step = 250000: loss = 0.004736536182463169
step = 250000: Average Return = 2.4000000953674316
```

12.4.10 Visualization

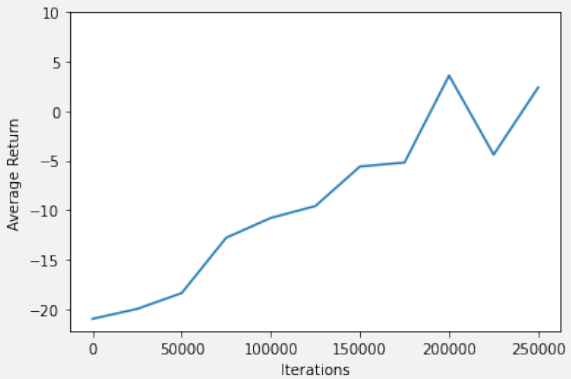
The notebook can plot the average return over training iterations. The average return should increase as the program performs more training iterations.

Code

```
iterations = range(0, num_iterations + 1, eval_interval)
plt.plot(iterations, returns)
plt.ylabel('Average Return')
plt.xlabel('Iterations')
plt.ylim(top=10)
```

Output

```
(-22.22999999523163, 10.0)
```



12.4.11 Videos

We now have a trained model and observed its training progress on a graph. Perhaps the most compelling way to view an Atari game's results is a video that allows us to see the agent play the game. The following functions are defined so that we can watch the agent play the game in the notebook.

Code

```
def embed_mp4(filename):
  """Embeds an mp4 file in the notebook."""
  video = open(filename,'rb').read()
  b64 = base64.b64encode(video)
  tag = '''
  <video width="640" height="480" controls>
    <source src="data:video/mp4; base64,{0}" type="video/mp4">
  Your browser does not support the video tag.
  </video>'''.format(b64.decode())

  return IPython.display.HTML(tag)

def create_policy_eval_video(policy, filename, num_episodes=5, fps=30):
  filename = filename + ".mp4"
  with imageio.get_writer(filename, fps=fps) as video:
    for _ in range(num_episodes):
      time_step = eval_env.reset()
      video.append_data(eval_py_env.render())
      while not time_step.is_last():
        action_step = policy.action(time_step)
        time_step = eval_env.step(action_step.action)
        video.append_data(eval_py_env.render())
  return embed_mp4(filename)
```

First, we will observe the trained agent play the game.

Code

```
create_policy_eval_video(agent.policy, "trained-agent")
```

Output

For comparison, we observe a random agent play. While the trained agent is far from perfect, it does outperform the random agent by a considerable amount.

Code

```
create_policy_eval_video(random_policy, "random-agent")
```

Output

12.5 Part 12.5: Application of Reinforcement Learning

Creating an environment is the first step to applying TF-Agent-based reinforcement learning to a problem with your design. In this part, we will see how to create your environment and apply it to an agent that allows actions to be floating-point values, rather than the discrete actions employed by the Deep Q-Networks (DQN) that we used earlier in this chapter. This new type of agent is called a Deep Deterministic Policy Gradients (DDPG) network. From an application standpoint, the primary difference between DDPG and DQN is that DQN only supports discrete actions, whereas DDPG supports continuous actions; however, there are other essential differences that we will cover later in this chapter.

The environment that I will demonstrate in this chapter simulates paying off a mortgage and saving for retirement. This simulation allows the agent to allocate their income between several types of account, buying luxury items, and paying off their mortgage. The goal is to maximize net worth. Because we wish to provide the agent with the ability to distribute their income among several accounts, we provide continuous (floating point) actions that determine this distribution of the agent's salary.

We begin, similarly to previous TF-Agent examples in this chapter, by importing needed packages.

Code

```
import base64
import imageio
import IPython
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image
import pyvirtualdisplay
import math
import numpy as np

import tensorflow as tf

from tf_agents.agents.ddpg import actor_network
```

```
from tf_agents.agents.ddpg import critic_network
from tf_agents.agents.ddpg import ddpg_agent

from tf_agents.agents.dqn import dqn_agent
from tf_agents.drivers import dynamic_step_driver
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.eval import metric_utils
from tf_agents.metrics import tf_metrics
from tf_agents.networks import q_network
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import tf_uniform_replay_buffer
from tf_agents.trajectories import trajectory
from tf_agents.trajectories import policy_step
from tf_agents.utils import common

import gym
from gym import spaces
from gym.utils import seeding
from gym.envs.registration import register
import PIL.ImageDraw
import PIL.Image
from PIL import ImageFont
```

Note, if you get the following error, restart and rerun the Google CoLab environment. Sometimes a restart is needed after installing TF-Agents.

```
AttributeError: module 'google.protobuf.descriptor' has no
    attribute '_internal_create_key'
```

We create a virtual display so that we can view the simulation in a Jupyter notebook.

Code

```
# Set up a virtual display for rendering OpenAI gym environments.
vdisplay = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

12.5.1 Create an Environment of your Own

An environment is a simulator that your agent runs in. An environment must have a current state. Some of this state is visible to the agent. However, the environment also hides some aspects of the state from the agent. Likewise, the agent takes actions that will affect the state of the environment. There may also be internal actions that occur outside of the control of the agent. For example, in the finance simulator demonstrated in this section, the agent does not control the investment returns or rate of inflation. Instead, the agent must react to these external actions and state components.

The environment class that you create must contain these elements:

- Be a child class of **gym.Env**

- Implement a **seed** function that sets a seed that governs the simulation's random aspects. For this environment, the seed oversees the random fluctuations in inflation and rates of return.
- Implement a **reset** function that resets the state for a new episode.
- Implement a **render** function that renders one frame of the simulation. The rendering is only for display and does not affect reinforcement learning.
- Implement a **step** function that performs one step of your simulation.

The class presented below implements a financial planning simulation. The agent must save for retirement and should attempt to amass the greatest possible net worth. The simulation includes the following key elements:

- Random starting salary between 40K (USD) and 60K (USD).
- Home loan for a house with a random purchase price that is between 1.5 and 4 times the starting salary.
- Home loan is a standard amortized 30-year loan with a fixed monthly payment.
- Paying higher than the home's monthly payment pays the loan down quicker. Paying below the monthly payment results in late fees and eventually foreclosure.
- Ability to allocate income between luxury purchases, home payments (above or below payment amount), as well as a taxable and tax-advantaged savings account.

The state is composed of the following floating point values:

- **age** - The agent's current age in months (steps)
- **salary** - The agent's starting salary, increases relative to inflation.
- **home_value** - The value of the agent's home, increases relative to inflation.
- **home_loan** - How much the agent still owes on their home.
- **req_home_pmt** - The minimum required home payment.
- **acct_tax_adv** - The balance of the tax advantaged retirement account.
- **acct_tax** - The balance of the taxable retuirement account.

The action space is composed of the following floating-point values (between 0 and 1):

- **home_loan** - The amount to apply to a home loan.
- **savings_tax_adv** - The amount to deposit in a tax-advantaged savings account.
- **savings taxable** - The amount to deposit in a taxable savings account.
- **luxury** - The amount to spend on luxury items/services.

The actions are weights that the program converts to a percentage of the total. For example, the home loan percentage is the home loan action value divided by all actions (including a home loan). The following code implements the environment and provides implementation details in the comments.

Code

```
class SimpleGameOfLifeEnv(gym.Env):
    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second': 1
    }

    STATE_ELEMENTS = 7
    STATES = ['age', 'salary', 'home_value', 'home_loan', 'req_home_pmt',
```

```
            'acct_tax_adv', 'acct_tax', "expenses", "actual_home_pmt",
            "tax_deposit",
            "tax_adv_deposit", "net_worth"]
    STATE_AGE = 0
    STATE_SALARY = 1
    STATE_HOME_VALUE = 2
    STATE_HOME_LOAN = 3
    STATE_HOME_REQ_PAYMENT = 4
    STATE_SAVE_TAX_ADV = 5
    STATE_SAVE_TAXABLE = 6

    MEG = 1.0e6

    ACTION_ELEMENTS = 4
    ACTION_HOME_LOAN = 0
    ACTION_SAVE_TAX_ADV = 1
    ACTION_SAVE_TAXABLE = 2
    ACTION_LUXURY = 3

    INFLATION = (0.015)/12.0
    INTEREST = (0.05)/12.0
    TAX_RATE = (.142)/12.0
    EXPENSES = 0.6
    INVEST_RETURN = 0.065/12.0
    SALARY_LOW = 40000.0
    SALARY_HIGH = 60000.0
    START_AGE = 18
    RETIRE_AGE = 80

    def __init__(self, goal_velocity=0):
      self.verbose = False
      self.viewer = None

      self.action_space = spaces.Box(
          low=0.0,
          high=1.0,
          shape=(SimpleGameOfLifeEnv.ACTION_ELEMENTS,),
          dtype=np.float32
      )
      self.observation_space = spaces.Box(
          low=0,
          high=2,
          shape=(SimpleGameOfLifeEnv.STATE_ELEMENTS,),
          dtype=np.float32
      )

      self.seed()
      self.reset()
```

```
        self.state_log = []

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def _calc_net_worth(self):
        home_value = self.state[SimpleGameOfLifeEnv.STATE_HOME_VALUE]
        principal = self.state[SimpleGameOfLifeEnv.STATE_HOME_LOAN]
        worth = home_value - principal
        worth += self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAX_ADV]
        worth += self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAXABLE]
        return worth

    def _eval_action(self, action, payment):
        # Calculate actions
        act_home_payment = action[SimpleGameOfLifeEnv.ACTION_HOME_LOAN]
        act_tax_adv_pay = action[SimpleGameOfLifeEnv.ACTION_SAVE_TAX_ADV]
        act_taxable = action[SimpleGameOfLifeEnv.ACTION_SAVE_TAXABLE]
        act_luxury = action[SimpleGameOfLifeEnv.ACTION_LUXURY]
        if payment <=0:
            act_home_payment = 0
        total_act = act_home_payment + act_tax_adv_pay + act_taxable + \
                act_luxury + self.expenses

        if total_act <1e-2:
            pct_home_payment = 0
            pct_tax_adv_pay = 0
            pct_taxable = 0
            pct_luxury = 0
        else:
            pct_home_payment = act_home_payment / total_act
            pct_tax_adv_pay = act_tax_adv_pay / total_act
            pct_taxable = act_taxable / total_act
            pct_luxury = act_luxury / total_act

        return pct_home_payment, pct_tax_adv_pay, pct_taxable, pct_luxury

    def step(self, action):
        self.last_action = action
        age = self.state[SimpleGameOfLifeEnv.STATE_AGE]
        salary = self.state[SimpleGameOfLifeEnv.STATE_SALARY]
        home_value = self.state[SimpleGameOfLifeEnv.STATE_HOME_VALUE]
        principal = self.state[SimpleGameOfLifeEnv.STATE_HOME_LOAN]
        payment = self.state[SimpleGameOfLifeEnv.STATE_HOME_REQ_PAYMENT]
        net1 = self._calc_net_worth()
        remaining_salary = salary
```

```
        # Calculate actions
        pct_home_payment, pct_tax_adv_pay, pct_taxable, pct_luxury = \
          self._eval_action(action,payment)

        # Expenses
        current_expenses = salary * self.expenses
        remaining_salary -= current_expenses
        if self.verbose:
          print(f"Expenses: {current_expenses}")
          print(f"Remaining Salary: {remaining_salary}")

        # Tax advantaged deposit action
        my_tax_adv_deposit = min(salary * pct_tax_adv_pay, remaining_salary)
        my_tax_adv_deposit = min(my_tax_adv_deposit, \
          self.year_tax_adv_deposit_left) # Govt CAP
        self.year_tax_adv_deposit_left -= my_tax_adv_deposit
        remaining_salary -= my_tax_adv_deposit
        tax_adv_deposit = my_tax_adv_deposit * 1.05 # Company match
        self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAX_ADV] += \
          int(tax_adv_deposit)

        if self.verbose:
          print(f"IRA Deposit: {tax_adv_deposit}")
          print(f"Remaining Salary: {remaining_salary}")

        # Tax
        remaining_salary -= remaining_salary * SimpleGameOfLifeEnv.TAX_RATE
        if self.verbose:
          print(f"Tax Salary: {remaining_salary}")

        # Home payment
        actual_payment = min(salary * pct_home_payment, remaining_salary)

        if principal>0:
          ipart = principal * SimpleGameOfLifeEnv.INTEREST
          ppart = actual_payment - ipart
          principal = int(principal-ppart)
          if principal<=0:
            principal = 0
            self.state[SimpleGameOfLifeEnv.STATE_HOME_REQ_PAYMENT] = 0
          elif actual_payment < payment:
            self.late_count += 1
            if self.late_count>15:
              sell = (home_value-principal)/2
              sell -= 20000
              sell = max(sell,0)
              self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAXABLE] += sell
```

```
            principal = 0
            home_value = 0
            self.expenses += .3
            self.state[SimpleGameOfLifeEnv.STATE_HOME_REQ_PAYMENT] = 0
            if self.verbose:
              print(f"Foreclosure!!")
          else:
            late_fee = payment * 0.1
            principal += late_fee
            if self.verbose:
              print(f"Late Fee: {late_fee}")


        self.state[SimpleGameOfLifeEnv.STATE_HOME_LOAN] = principal
        remaining_salary -= actual_payment

      if self.verbose:
        print(f"Home Payment: {actual_payment}")
        print(f"Remaining Salary: {remaining_salary}")

      # Taxable savings
      actual_savings = remaining_salary * pct_taxable
      self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAXABLE] += actual_savings
      remaining_salary -= actual_savings

      if self.verbose:
        print(f"Tax Save: {actual_savings}")
        print(f"Remaining Salary (goes to Luxury): {remaining_salary}")

      # Investment income
      return_taxable = self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAXABLE] * \
          self.invest_return
      return_tax_adv = self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAX_ADV] * \
          self.invest_return

      return_taxable *= 1-SimpleGameOfLifeEnv.TAX_RATE
      self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAXABLE] += return_taxable
      self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAX_ADV] += return_tax_adv

      # Yearly events
      if age>0 and age % 12 == 0:
        self.perform_yearly()

      # Monthly events
      self.state[SimpleGameOfLifeEnv.STATE_AGE] += 1

      # Time to retire (by age?)
      done = self.state[SimpleGameOfLifeEnv.STATE_AGE] > \
```

```
        (SimpleGameOfLifeEnv.RETIRE_AGE*12)

      # Calculate reward
      net2 = self._calc_net_worth()
      reward = net2 - net1

      # Track progress
      if self.verbose:
        print(f"Networth: {nw}")
        print(f"*** End Step {self.step_num}: State={self.state}, \
          Reward={reward}")
      self.state_log.append(self.state + [current_expenses, actual_payment,
      actual_savings, my_tax_adv_deposit, net2])
      self.step_num += 1

      # Normalize state and finish up
      norm_state = [x/SimpleGameOfLifeEnv.MEG for x in self.state]
      return norm_state, reward/SimpleGameOfLifeEnv.MEG, done, {}

    def perform_yearly(self):
      salary = self.state[SimpleGameOfLifeEnv.STATE_SALARY]
      home_value = self.state[SimpleGameOfLifeEnv.STATE_HOME_VALUE]

      self.inflation = SimpleGameOfLifeEnv.INTEREST + \
          self.np_random.normal(loc=0,scale=1e-2)
      self.invest_return = SimpleGameOfLifeEnv.INVEST_RETURN + \
          self.np_random.normal(loc=0,scale=1e-2)

      self.year_tax_adv_deposit_left = 19000
      self.state[SimpleGameOfLifeEnv.STATE_SALARY] = \
        int(salary * (1+self.inflation))

      self.state[SimpleGameOfLifeEnv.STATE_HOME_VALUE] \
        = int(home_value * (1+self.inflation))

    def reset(self):
      self.expenses = SimpleGameOfLifeEnv.EXPENSES
      self.late_count = 0
      self.step_num = 0
      self.last_action = [0] * SimpleGameOfLifeEnv.ACTION_ELEMENTS
      self.state = [0] * SimpleGameOfLifeEnv.STATE_ELEMENTS
      self.state_log = []
      salary = float(self.np_random.randint(low=\
                SimpleGameOfLifeEnv.SALARY_LOW,\
                high=SimpleGameOfLifeEnv.SALARY_HIGH))
      house_mult = self.np_random.uniform(low=1.5,high=4)
      value = round(salary*house_mult)
      p = (value*0.9)
```

```
        i = SimpleGameOfLifeEnv.INTEREST
        n = 30 * 12
        m = float(int(p *  ( i * (1 + i)**n ) / ( (1 + i)**n - 1)))
        self.state[SimpleGameOfLifeEnv.STATE_AGE] = \
          SimpleGameOfLifeEnv.START_AGE * 12
        self.state[SimpleGameOfLifeEnv.STATE_SALARY] = salary / 12.0
        self.state[SimpleGameOfLifeEnv.STATE_HOME_VALUE] = value
        self.state[SimpleGameOfLifeEnv.STATE_HOME_LOAN] = p
        self.state[SimpleGameOfLifeEnv.STATE_HOME_REQ_PAYMENT] = m
        self.year_tax_adv_deposit_left = 19000
        self.perform_yearly()
        return np.array(self.state)

    def render(self, mode='human'):
        screen_width = 600
        screen_height = 400

        img = PIL.Image.new('RGB', (600, 400))
        d = PIL.ImageDraw.Draw(img)
        font = ImageFont.load_default()
        y = 0
        _, height = d.textsize("W", font)

        age = self.state[SimpleGameOfLifeEnv.STATE_AGE]
        salary = self.state[SimpleGameOfLifeEnv.STATE_SALARY]*12
        home_value = self.state[SimpleGameOfLifeEnv.STATE_HOME_VALUE]
        home_loan = self.state[SimpleGameOfLifeEnv.STATE_HOME_LOAN]
        home_payment = self.state[SimpleGameOfLifeEnv.STATE_HOME_REQ_PAYMENT]
        balance_tax_adv = self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAX_ADV]
        balance_taxable = self.state[SimpleGameOfLifeEnv.STATE_SAVE_TAXABLE]
        net_worth = self._calc_net_worth()

        d.text((0, y), f"Age: {age/12}", fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Salary: {salary:,}", fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Home Value: {home_value:,}", \
                fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Home Loan: {home_loan:,}", \
               fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Home Payment: {home_payment:,}", \
               fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Balance Tax Adv: {balance_tax_adv:,}", \
               fill=(0, 255, 0))
        y+=height
```

```
        d.text((0, y), f"Balance Taxable: {balance_taxable:,}", \
               fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Net Worth: {net_worth:,}", fill=(0, 255, 0))
        y+=height*2

        payment = self.state[SimpleGameOfLifeEnv.STATE_HOME_REQ_PAYMENT]
        pct_home_payment, pct_tax_adv_pay, pct_taxable, pct_luxury = \
          self._eval_action(self.last_action,payment)
        d.text((0, y), f"Percent Home Payment: {pct_home_payment}", \
               fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Percent Tax Adv: {pct_tax_adv_pay}",
               fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Percent Taxable: {pct_taxable}", fill=(0, 255, 0))
        y+=height
        d.text((0, y), f"Percent Luxury: {pct_luxury}", fill=(0, 255, 0))

        return np.array(img)

    def close(self):
        pass
```

You must register the environment class with TF-Agents before your program can use it.

Code

```
register(
    id='simple-game-of-life-v0',
    entry_point=f'{__name__}:SimpleGameOfLifeEnv',
)
```

12.5.2 Testing the Environment

This financial planning environment is complex. It took me some degree of testing to perfect it. Even at the current state of this simulator, it is far from a complete financial simulator. The primary objective of this simulator is to demonstrate creating your own environment for a non-video game project.

I used the following code to help test this simulator. I ran the simulator with fixed actions and then loaded the state into a Pandas data frame for easy viewing.

Code

```
env_name = 'simple-game-of-life-v0'
env = gym.make(env_name)

env.reset()
done = False
```

```
i = 0
env.verbose = False
while not done:
    i += 1
    #if i>36: break
    # ACTION_HOME_LOAN = 0, ACTION_SAVE_TAX_ADV = 1, ACTION_SAVE_TAXABLE = 2
    # ACTION_LUXURY = 3
    state, reward, done, _ = env.step([1,1,0,0])
    env.render()

env.close()
print(env._calc_net_worth())
```

Output

```
8012911.816196918
```

Code

```
import pandas as pd

df = pd.DataFrame(env.state_log, columns=SimpleGameOfLifeEnv.STATES)
df = df.round(0)
df['age'] = df['age']/12
df['age'] = df['age'].round(2)
for col in df.columns:
  df[col] = df[col].apply(lambda x : "{:,}".format(x))

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 12)
display(df)
```

Output

	age	salary	home_value	...	tax_deposit	tax_adv_deposit	net_worth
0	18.08	4,189	90,751	...	0.0	1,604.0	10,305.0
1	18.17	4,189	90,751	...	0.0	1,611.0	11,677.0
2	18.25	4,189	90,751	...	0.0	1,611.0	13,050.0
3	18.33	4,189	90,751	...	0.0	1,611.0	14,422.0
4	18.42	4,189	90,751	...	0.0	1,611.0	15,795.0
...	...	...	...	...	...	...	...
740	79.75	5,591	121,865	...	0.0	559.0	7,378,407.0
741	79.83	5,591	121,865	...	0.0	559.0	7,531,492.0
742	79.92	5,591	121,865	...	0.0	559.0	7,687,793.0
743	80.0	5,591	121,865	...	0.0	559.0	7,847,379.0
744	80.08	5,710	124,459	...	0.0	559.0	8,012,912.0

1810888.5833333335

12.5.3 Hyperparameters

I tuned the following hyperparamaters to get a reasonable result from training the agent. Further optimization would be beneficial.

Code

```
# How long should training run?
num_iterations = 50000
# How many initial random steps, before training start, to
# collect initial data.
initial_collect_steps = 1000
# How many steps should we run each iteration to collect
# data from.
collect_steps_per_iteration = 50
# How much data should we store for training examples.
replay_buffer_max_length = 100000

batch_size = 64
#learning_rate = 1e-4
# How often should the program provide an update.
log_interval = 2500

# How many episodes should the program use for each evaluation.
num_eval_episodes = 100
# How often should an evaluation occur.
eval_interval = 5000
```

12.5.4 Instanciate the Environment

We are now ready to make use of our environment. Because we registered the environment with TF-Agents the program can load the environment by its name "simple-game-of-life-v". You can also try the continuous version of the mountain car environment by switching the name given below. The continuous mountain car works similar to the discrete mountain car seen previously. However, the continuous mountain car allows the degree of forward or backward force to be specified.

Code

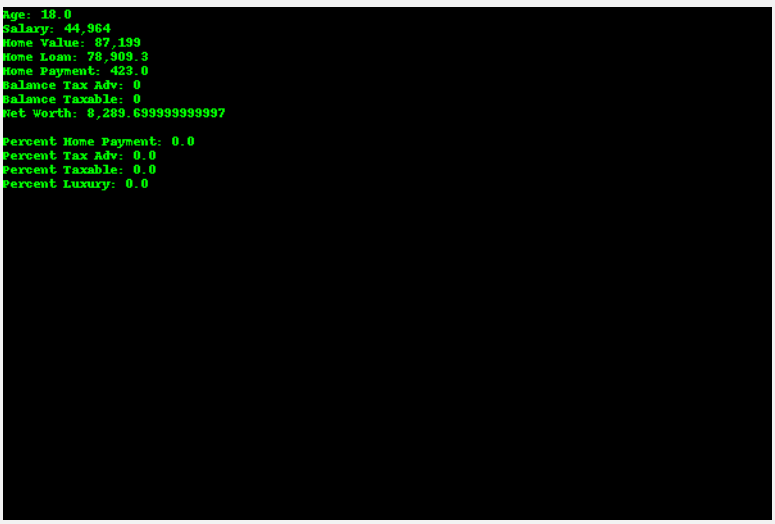
```
env_name = 'simple-game-of-life-v0'
#env_name = 'MountainCarContinuous-v0'
env = suite_gym.load(env_name)
```

We can now have a quick look at the first state rendered. Here we can see the random salary and home values are chosen for an agent. The learned policy must be able to take into consideration different starting salaries and home values and find an appropriate strategy.

Code

```
env.reset()
PIL.Image.fromarray(env.render())
```

Output



Just as before, the program instantiates two environments: one for training and one for evaluation.

Code

```
train_py_env = suite_gym.load(env_name)
eval_py_env = suite_gym.load(env_name)

train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
```

You might be wondering why a DQN does not support continuous actions. The reason for this limitation is that the DQN algorithm maps each action is an output neuron. Each of these neurons predicts the likely future reward fo taking each action. Generally, the DQN agent will perform the action that has the highest reward. The algorithm knows the future rewards for each particular action. However, because a continuous number represented in a computer has an effectively infinite number of possible values, it is not possible to calculate a future reward estimate for all of them.

To provide a continuous action space we will use the Deep Deterministic Policy Gradients (DDPG) algorithm.[26]This technique uses two neural networks. The first neural network, called an actor, acts as the agent and predicts the expected reward for a given value of the action. The second neural network, called a critic, is trained to predict the accuracy of the actor-network. Training two neural networks in parallel that operator adversarially to each other is a popular technique. Earlier in this course, we saw that Generative Adversarial Networks (GAN) made use of a similar method. Figure 12.4 shows the structure of the DDPG network that we will use.

The environment provides the same input ($x(t)$) for each time step to both the actor and critic networks. The temporal difference error ($r(t)$) reports the difference between the estimated reward at any given state

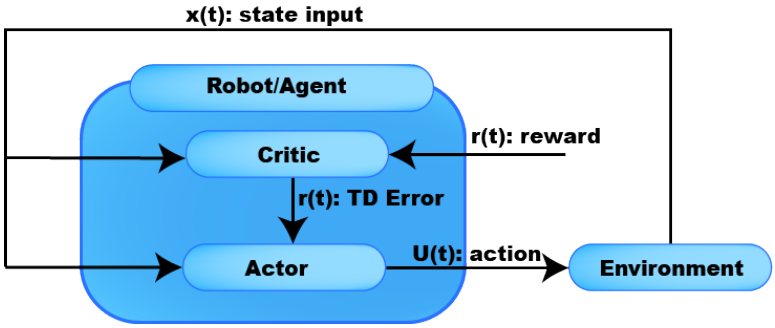


Figure 12.4: Actor Critic Model

or time step and the actual reward.

The following code creates the actor and critic neural networks.

Code

```
actor_fc_layers=(400, 300)
critic_obs_fc_layers=(400,)
critic_action_fc_layers=None
critic_joint_fc_layers=(300,)
ou_stddev=0.2
ou_damping=0.15
target_update_tau=0.05
target_update_period=5
dqda_clipping=None
td_errors_loss_fn=tf.compat.v1.losses.huber_loss
gamma=0.995
reward_scale_factor=1.0
gradient_clipping=None

actor_learning_rate=1e-4
critic_learning_rate=1e-3
debug_summaries=False
summarize_grads_and_vars=False

global_step = tf.compat.v1.train.get_or_create_global_step()

actor_net = actor_network.ActorNetwork(
        train_env.time_step_spec().observation,
        train_env.action_spec(),
        fc_layer_params=actor_fc_layers,
    )

critic_net_input_specs = (train_env.time_step_spec().observation,
                          train_env.action_spec())

critic_net = critic_network.CriticNetwork(
```

```
    critic_net_input_specs,
    observation_fc_layer_params=critic_obs_fc_layers,
    action_fc_layer_params=critic_action_fc_layers,
    joint_fc_layer_params=critic_joint_fc_layers,
)

tf_agent = ddpg_agent.DdpgAgent(
    train_env.time_step_spec(),
    train_env.action_spec(),
    actor_network=actor_net,
    critic_network=critic_net,
    actor_optimizer=tf.compat.v1.train.AdamOptimizer(
        learning_rate=actor_learning_rate),
    critic_optimizer=tf.compat.v1.train.AdamOptimizer(
        learning_rate=critic_learning_rate),
    ou_stddev=ou_stddev,
    ou_damping=ou_damping,
    target_update_tau=target_update_tau,
    target_update_period=target_update_period,
    dqda_clipping=dqda_clipping,
    td_errors_loss_fn=td_errors_loss_fn,
    gamma=gamma,
    reward_scale_factor=reward_scale_factor,
    gradient_clipping=gradient_clipping,
    debug_summaries=debug_summaries,
    summarize_grads_and_vars=summarize_grads_and_vars,
    train_step_counter=global_step)
tf_agent.initialize()
```

12.5.5 Metrics and Evaluation

Just as in previous examples, we will compute the average return over several episodes to evaluate performance.

Code

```
def compute_avg_return(environment, policy, num_episodes=10):

  total_return = 0.0
  for _ in range(num_episodes):

    time_step = environment.reset()
    episode_return = 0.0

    while not time_step.is_last():
      action_step = policy.action(time_step)
      time_step = environment.step(action_step.action)
      episode_return += time_step.reward
```

```
    total_return += episode_return

  avg_return = total_return / num_episodes
  return avg_return.numpy()[0]


# See also the metrics module for standard implementations of
# different metrics.
# https://github.com/tensorflow/agents/tree/master/tf_agents/metrics
```

12.5.6 Data Collection

Now execute the random policy in the environment for a few steps, recording the data in the replay buffer.

Code

```
def collect_step(environment, policy, buffer):
  time_step = environment.current_time_step()
  action_step = policy.action(time_step)
  next_time_step = environment.step(action_step.action)
  traj = trajectory.from_transition(time_step, action_step, next_time_step)

  # Add trajectory to the replay buffer
  buffer.add_batch(traj)

def collect_data(env, policy, buffer, steps):
  for _ in range(steps):
    collect_step(env, policy, buffer)

random_policy = random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
                                                train_env.action_spec())

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=tf_agent.collect_data_spec,
    batch_size=train_env.batch_size,
    max_length=replay_buffer_max_length)

collect_data(train_env, random_policy, replay_buffer, steps=100)

# Dataset generates trajectories with shape [Bx2x...]
dataset = replay_buffer.as_dataset(
    num_parallel_calls=3,
    sample_batch_size=batch_size,
    num_steps=2).prefetch(3)
```

12.5.7 Training the agent

We are now ready to train the agent. This process can take many hours, depending on how many episodes you wish to run through. As training occurs, this code will update on both the loss and average return. As training becomes more successful, the average return should increase. The losses reported reflecting the average loss for individual training batches.

Code

```
iterator = iter(dataset)

# (Optional) Optimize by wrapping some of the code in a graph using
# TF function.
tf_agent.train = common.function(tf_agent.train)

# Reset the train step
tf_agent.train_step_counter.assign(0)

# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, tf_agent.policy, \
                                num_eval_episodes)
returns = [avg_return]

for _ in range(num_iterations):

  # Collect a few steps using collect_policy and save to the replay buffer.
  for _ in range(collect_steps_per_iteration):
    collect_step(train_env, tf_agent.collect_policy, replay_buffer)

  # Sample a batch of data from the buffer and update the agent's network.
  experience, unused_info = next(iterator)
  train_loss = tf_agent.train(experience).loss

  step = tf_agent.train_step_counter.numpy()

  if step % log_interval == 0:
    print('step = {0}: loss = {1}'.format(step, train_loss))

  if step % eval_interval == 0:
    avg_return = compute_avg_return(eval_env, tf_agent.policy, \
                                    num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
```

Output

```
step = 2500: loss = 0.00048063506255857646
step = 5000: loss = 0.0009628287516534328
step = 5000: Average Return = 5.293339252471924
```

```
step = 7500: loss = 0.0018985542701557279
step = 10000: loss = 0.0003614715242292732
step = 10000: Average Return = 8.426756858825684
step = 12500: loss = 0.001215201336890459
step = 15000: loss = 0.000617831654381007
step = 15000: Average Return = 10.426517486572266
step = 17500: loss = 0.0016348579665645957
step = 20000: loss = 0.0035204419400542974
step = 20000: Average Return = 8.884005546569824
step = 22500: loss = 0.5189836025238037
step = 25000: loss = 0.004120268858969212
step = 25000: Average Return = 10.956489562988281

...

step = 45000: loss = 0.19088667631149292
step = 45000: Average Return = 7.3852128982543945
step = 47500: loss = 0.0020619637798517942
step = 50000: loss = 0.030354363843798637
step = 50000: Average Return = 8.955848693847656
```

12.5.8 Visualization

The notebook can plot the average return over training iterations. The average return should increase as the program performs more training iterations.

12.5.9 Plots

Use **matplotlib.pyplot** to chart how the policy improved during training.

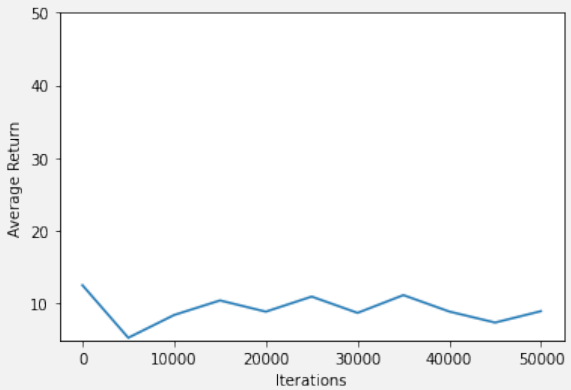
One iteration of **Cartpole-v0** consists of 200 time steps. The environment gives a reward of `+1` for each step the pole stays up, so the maximum return for one episode is 200. The charts show the return increasing towards that maximum each time it is evaluated during training. (It may be a little unstable and not increase each time monotonically.)

Code

```
iterations = range(0, num_iterations + 1, eval_interval)
plt.plot(iterations, returns)
plt.ylabel('Average Return')
plt.xlabel('Iterations')
plt.ylim(top=50)
```

Output

```
(4.931626772880554, 50.0)
```



12.5.10 Videos

We use the following functions to produce video in Jupyter notebook.

Code

```
def embed_mp4(filename):
  """Embeds an mp4 file in the notebook."""
  video = open(filename,'rb').read()
  b64 = base64.b64encode(video)
  tag = '''
  <video width="640" height="480" controls>
    <source src="data:video/mp4; base64,{0}" type="video/mp4">
  Your browser does not support the video tag.
  </video>'''.format(b64.decode())

  return IPython.display.HTML(tag)

def create_policy_eval_video(policy, filename, num_episodes=5, fps=30):
  filename = filename + ".mp4"
  with imageio.get_writer(filename, fps=fps) as video:
    for _ in range(num_episodes):
      time_step = eval_env.reset()
      video.append_data(eval_py_env.render())
      while not time_step.is_last():
        action_step = policy.action(time_step)
        time_step = eval_env.step(action_step.action)
        video.append_data(eval_py_env.render())
  return embed_mp4(filename)

create_policy_eval_video(tf_agent.policy, "trained-agent")
```

Output

Chapter 13

Advanced/Other Topics

13.1 Part 13.1: Flask and Deep Learning Web Services

Suppose you would like to create websites based on neural networks. In that case, the neural network must be exposed in a way that can be efficiently executed by Python and other programming languages. The usual means for such integration is a web service. One of the most popular libraries for doing this in Python is Flask. This library allows you to quickly deploy your Python applications, including TensorFlow, as web services.

Neural network deployment is a complex process, usually carried out by a company's Information Technology (IT) group. When large numbers of clients must access your model, scalability becomes essential. The cloud usually handles this. The designers of Flask did not design for high-volume systems. When deployed to production, you will usually wrap models in Gunicorn or TensorFlow Serving. We will discuss high volume cloud deployment in the next section. Everything presented in this part ith Flask is directly compatible with the higher volume Gunicorn system. It is common to use a development system, such as Flask, when developing your initial system.

13.1.1 Flask Hello World

It is uncommon to run Flask from a Jupyter notebook. Flask is the server, and Jupyter usually fills the role of the client. However, we can run a simple web service from Jupyter. We will quickly move beyond this and deploy using a Python script (.py). Because we must use .py files, it won't be easy to use Google CoLab, as you will be running from the command line. For now, let's execute a Flask web container in Jupyter.

Code

```
from werkzeug.wrappers import Request, Response
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == '__main__':
    from werkzeug.serving import run_simple
```

```
    run_simple('localhost', 9000, app)
```

This program starts a web service on port 9000 of your computer. This cell will remain running (appearing locked up). However, it is merely waiting for browsers to connect. If you point your browser at the following URL, you will interact with the Flask web service.

- http://localhost:9000/

You should see Hello World displayed.

13.1.2 MPG Flask

Usually, you will interact with a web service through JSON. A program will send a JSON message to your Flask application, and your Flask application will return a JSON. Later, in module 13.3, we will see how to attach this web service to a web application that you can interact with through a browser. We will create a Flask wrapper for a neural network that predicts the miles per gallon. The sample JSON will look like this.

```
{
  "cylinders": 8,
  "displacement": 300,
  "horsepower": 78,
  "weight": 3500,
  "acceleration": 20,
  "year": 76,
  "origin": 1
}
```

We will see two different means of POSTing this JSON data to our web server. First, we will use a utility called POSTman. Secondly, we will use Python code to construct the JSON message and interact with Flask.

First, it is necessary to train a neural network with the MPG dataset. This technique is very similar to what we've done many times before. However, we will save the neural network so that we can load it later. We do not want to have Flask train the neural network. We wish to have the neural network already trained and deploy the already prepared .H5 file to save the neural network. The following code trains an MPG neural network.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
```

```
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5, \
        verbose=1, mode='auto',\
        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),\
          callbacks=[monitor],verbose=2,epochs=1000)
```

Output

```
Train on 298 samples, validate on 100 samples
Epoch 1/1000
298/298 - 0s - loss: 2009.9526 - val_loss: 1452.2106
Epoch 2/1000
298/298 - 0s - loss: 562.4718 - val_loss: 532.2283
Epoch 3/1000
298/298 - 0s - loss: 331.3064 - val_loss: 229.2169
Epoch 4/1000
298/298 - 0s - loss: 197.1110 - val_loss: 149.5195
Epoch 5/1000
298/298 - 0s - loss: 167.2275 - val_loss: 137.9593
Epoch 6/1000
298/298 - 0s - loss: 141.5222 - val_loss: 123.4302
Epoch 7/1000
298/298 - 0s - loss: 134.0808 - val_loss: 119.0394
```

```
...

Epoch 52/1000
Restoring model weights from the end of the best epoch.
298/298 - 0s - loss: 37.9472 - val_loss: 32.6139
Epoch 00052: early stopping
<tensorflow.python.keras.callbacks.History at 0x12df96208>
```

Next, we evaluate the score. This evaluation is more of a sanity check to ensure the code above worked as expected.

Code

```
pred = model.predict(x_test)
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"After load score (RMSE): {score}")
```

Output

```
After load score (RMSE): 5.465193688130732
```

Next, we save the neural network to a .H5 file.

Code

```
model.save(os.path.join("./dnn/","mpg_model.h5"))
```

We want the Flask web service to check that the input JSON is valid. To do this, we need to know what values we expect and what their logical ranges are. The following code outputs the expected fields, their ranges, and packages all of this information into a JSON object that you should copy to the Flask web application. This code allows us to validate the incoming JSON requests.

Code

```
cols = [x for x in df.columns if x not in ('mpg','name')]

print("{")
for i,name in enumerate(cols):
    print(f'"{name}":{{"min":{df[name].min()},\
          "max":{df[name].max()}}}{","if i<(len(cols)-1)else""}')
print("}")
```

Output

```
{
"cylinders":{"min":3,"max":8},
"displacement":{"min":68.0,"max":455.0},
```

```
"horsepower":{"min":46.0,"max":230.0},
"weight":{"min":1613,"max":5140},
"acceleration":{"min":8.0,"max":24.8},
"year":{"min":70,"max":82},
"origin":{"min":1,"max":3}
}
```

Finally, we set up the Python code to call the model for a single car and get a prediction. You should also copy this code to the Flask web application.

Code

```
import os
from tensorflow.keras.models import load_model
import numpy as np

model = load_model(os.path.join("./dnn/","mpg_model.h5"))
x = np.zeros( (1,7) )

x[0,0] = 8 # 'cylinders',
x[0,1] = 400 # 'displacement',
x[0,2] = 80 # 'horsepower',
x[0,3] = 2000 # 'weight',
x[0,4] = 19 # 'acceleration',
x[0,5] = 72 # 'year',
x[0,6] = 1 # 'origin'


pred = model.predict(x)
float(pred[0])
```

Output

```
6.212100505828857
```

The completed web application can be found here:

- mpg_server_1.py

You can run this server from the command line with the following command:

```
python mpg_server_1.py
```

If you are using a virtual environment (described in Module 1.1), make sure to use the **`activate tensorflow`** command for Windows or **`source activate tensorflow`** for Mac before executing the above command.

13.1.3 Flask MPG Client

Now that we have a web service running, we would like to access it. This server is a bit more complicated than the "Hello World" web server we first saw in this part. The request to display was an HTTP GET. We must now do an HTTP POST. To accomplish access to a web service, you must use a client. We will see how to use PostMan and directly through a Python program in Jupyter.

We will begin with PostMan. If you have not already done so, install PostMan.

To successfully use PostMan to query your web service, you must enter the following settings:

- POST Request to http://localhost:5000/api/mpg
- RAW JSON and paste in JSON from above
- Click Send and you should get a correct result

Figure 13.1 shows a successful result.

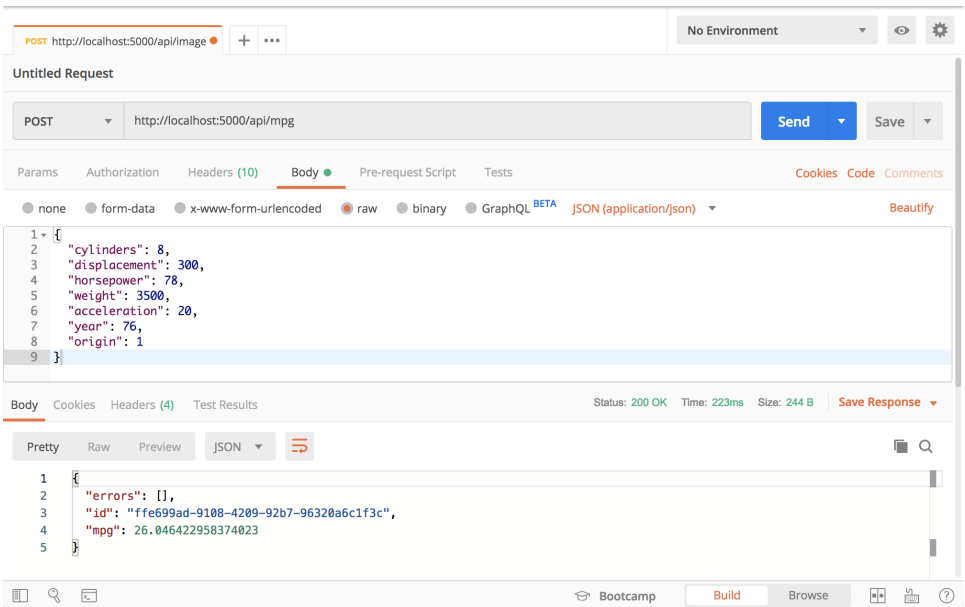


Figure 13.1: PostMan JSON

This same process can be done programmatically in Python.

Code

```
import requests

json = {
  "cylinders": 8,
  "displacement": 300,
  "horsepower": 78,
  "weight": 3500,
  "acceleration": 20,
  "year": 76,
  "origin": 1
```

```
}

r = requests.post("http://localhost:5000/api/mpg",json=json)
if r.status_code == 200:
    print("Success: {}".format(r.text))
else: print("Failure: {}".format(r.text))
```

Output

```
Success: {
  "errors": [],
  "id": "643d027e-554f-4401-ba5f-78592ae7e070",
  "mpg": 23.885438919067383
}
```

13.1.4 Images and Web Services

We can also accept images from web services. We will create a web service that accepts images and classifies them using MobileNet. To use your neural network, you will follow the same process; load your network as we did for the MPG example. You can find the completed web service can here:

image_server_1.py

You can run this server from the command line with:

```
python mpg_server_1.py
```

If you are using a virtual environment (described in Module 1.1), make sure to use the **`activate tensorflow`** command for Windows or **`source activate tensorflow`** for Mac before executing the above command.

To successfully use PostMan to query your web service, you must enter the following settings:

- POST Request to http://localhost:5000/api/image
- Use "Form Data" and create one entry named "image" that is a file. Choose an image file to classify.
- Click Send and you should get a correct result

Figure 13.2 shows a successful result.

This same process can be done programmatically in Python.

Code

```
import requests
response = requests.post('http://localhost:5000/api/image', files=\
        dict(image=('hickory.jpeg',open('./photos/hickory.jpeg','rb'))))
if response.status_code == 200:
    print("Success: {}".format(response.text))
else: print("Failure: {}".format(response.text))
```

Output

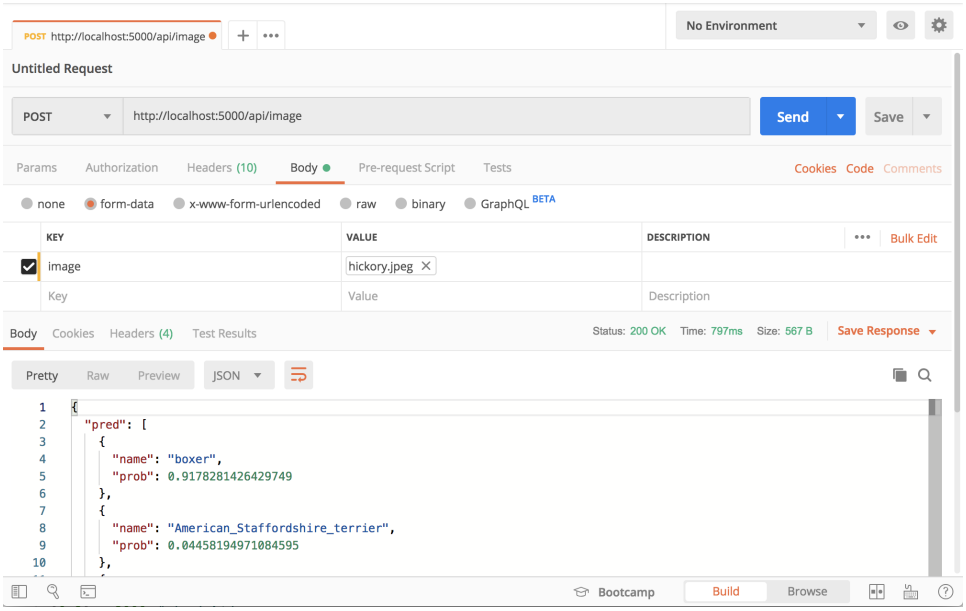


Figure 13.2: PostMan Images


```
Success: {
  "pred": [
    {
      "name": "boxer",
      "prob": 0.9178281426429749
    },
    {
      "name": "American_Staffordshire_terrier",
      "prob": 0.04458194971084595
    },
    {
      "name": "French_bulldog",
      "prob": 0.018736232072114944
    },
    {

...

      "name": "pug",
      "prob": 0.0009862519800662994
    }
  ]
}
```

13.2 Part 13.2: Deploying a Model to AWS

13.2.1 Train Model (optionally, outside of AWS)

There are three parts to this example. You must run each of these parts differently. For the first part to work, you must run this code from AWS SageMaker. To do this, you will need to upload this IPYNB (for Module 13.2) to AWS Sage Maker and open it from Jupyter. I demonstrate the complete process in the above YouTube video.

We begin by training an MPG dataset.

Code

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
        'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Split into validation and training sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')

monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
            verbose=1, mode='auto',
```

```
        restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

Next, we evaluate the RMSE. The goal is more to show how to create a cloud API than to achieve a low RMSE.

Code

```
pred = model.predict(x_test)
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
print(f"RMSE␣Score:␣{score}")
```

Next, we save the weights and structure of the neural network, as was demonstrated earlier in this course. You will use these two files to generate a **ProtoBuf** file that AWS will for the actual deployment. We store it into two separate files because we ONLY want the structure and weights. A single MD5 file, such as model.save(...), also contains training parameters and other features that may cause version issues when uploading to AWS. Remember, AWS may have a different version for TensorFlow than you do locally. Usually, AWS will have an older version.

Code

```
save_path = "./dnn/"

model.save_weights(os.path.join(save_path,"mpg_model-weights.h5"))

# save neural network structure to JSON (no weights)
model_json = model.to_json()
with open(os.path.join(save_path,"mpg_model.json"), "w") as json_file:
    json_file.write(model_json)
```

We will upload the two files generated to the **./dnn/** folder to AWS. If you were running the entire process from AWS, then they will not need to be uploaded.

We also print out the values to one car; we will copy these later when we test the API.

Code

```
x[0].tolist()
```

13.2.2 Next Step: Convert Model (must use AWS SageMaker Notebook)

To complete this portion, you will need to be running from s Jupyter notebook on AWS SageMaker. The following is based on an example from AWS documentation but customized to this class example.

13.2.3 Step 1. Set up

In the AWS Management Console, go to the Amazon SageMaker console. Choose Notebook Instances and create a new notebook instance. Upload this notebook and set the kernel to conda_tensorflow_p36.

The get_execution_role function retrieves the AWS Identity and Access Management (IAM) role you created at the time of creating your notebook instance.

Code

```
import boto3, re
from sagemaker import get_execution_role

role = get_execution_role()
```

13.2.4 Step 2. Load the Keras model using the JSON and weights file

The following cell loads the necessary imports from AWS. Note that we are using "import keras" compared to the "import keras.tensorflow" advised for the rest of the course. This practice is recommended by AWS currently.

Code

```
import keras
from keras.models import model_from_json
```

Create a directory called keras_model, navigate to keras_model from the Jupyter notebook home, and upload the model.json and model-weights.h5 files (using the "Upload" menu on the Jupyter notebook home).

Code

```
!mkdir keras_model
```

Navigate to keras_model from the Jupyter notebook home, and upload your model.json and model-weights.h5 files (using the "Upload" menu on the Jupyter notebook home). Use the files that you generated in step 2.

Code

```
!ls keras_model
```

Make sure you've uploaded your model to the directory by this point. If you saw no files at the above step, upload your files and rerun.

Code

```
import tensorflow as tf

json_file = open('/home/ec2-user/SageMaker/keras_model/'+\
                 'mpg_model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json,custom_objects=\
        {"GlorotUniform": tf.keras.initializers.glorot_uniform})
```

Code

```
loaded_model.load_weights(\
    '/home/ec2-user/SageMaker/keras_model/mpg_model-weights.h5')
print("Loaded model from disk")
```

13.2.5 Step 3. Export the Keras model to the TensorFlow ProtoBuf format (must use AWS SageMaker Notebook)

As you are probably noticing, there are many ways to save a Keras neural network. So far, we've seen:

- YAML File - Structure only
- JSON File - Structure only
- H5 Complete Model
- H5 Weights only

There is a fifth, which is the **ProtoBuf** format. **ProtoBuf** is typically only used for deployment. We will now convert the model we just loaded into this format.

Code

```
from tensorflow.python.saved_model import builder
from tensorflow.python.saved_model.signature_def_utils
import predict_signature_def
from tensorflow.python.saved_model import tag_constants

# Note: This directory structure will need to be followed - see notes
# for the next section
model_version = '1'
export_dir = 'export/Servo/' + model_version
```

It is very important that this export directory be empty. Be careful, the following command deletes the entire expor directory. (this should be fine)

Code

```
import shutil
shutil.rmtree(export_dir)
```

Code

```
# Build the Protocol Buffer SavedModel at 'export_dir'
build = builder.SavedModelBuilder(export_dir)
```

Code

```
# Create prediction signature to be used by TensorFlow Serving Predict API
signature = predict_signature_def(
    inputs={"inputs": loaded_model.input}, \
```

```
    outputs={"score": loaded_model.output})
```

Code

```
from keras import backend as K

with K.get_session() as sess:
    # Save the meta graph and variables
    build.add_meta_graph_and_variables(
        sess=sess, tags=[tag_constants.SERVING], \
        signature_def_map={"serving_default": signature})
    build.save()
```

You will notice the use of **signature_def_map**. This map bridges any incompatibilities between the version of TensorFlow you were running locally and the AWS version. You might need to add additional entries here.

13.2.6 Step 4. Convert TensorFlow model to a SageMaker readable format (must use AWS SageMaker Notebook)

Move the TensorFlow exported model into a directory export\Servo. SageMaker will recognize this as a loadable TensorFlow model. Your directory and file structure should look like:

Code

```
!ls export
```

Code

```
!ls export/Servo
```

Code

```
!ls export/Servo/1/variables
```

13.2.7 Tar the entire directory and upload to S3

Next you will TAR the directory. TAR is a UNIX file format that combines multiple files, similar to ZIP.

Code

```
import tarfile
with tarfile.open('model.tar.gz', mode='w:gz') as archive:
    archive.add('export', recursive=True)
```

Upload TAR file to S3.

Code

```
import sagemaker

sagemaker_session = sagemaker.Session()
inputs = sagemaker_session.upload_data(path='model.tar.gz', \
                 key_prefix='model')
```

13.2.8 Step 5. Deploy the trained model (must use AWS SageMaker Notebook)

The entry_point file "train.py" can be an empty Python file. AWS currently requires this step; however, AWS will likely remove this requirement at a later date.

Code

```
!touch train.py
```

Code

```
from sagemaker.tensorflow.model import TensorFlowModel
sagemaker_model = TensorFlowModel(model_data = 's3://' + \
    sagemaker_session.default_bucket() + '/model/model.tar.gz',
    role = role,
    framework_version = '1.12',
    entry_point = 'train.py')
```

Note, the following command can take 5-8 minutes to complete.

Code

```
%%time
predictor = sagemaker_model.deploy(initial_instance_count=1,
                                   instance_type='ml.m4.xlarge')
```

Code

```
predictor.endpoint
```

Note: You will need to update the endpoint in the command below with the endpoint name from the output of the previous cell (e.g. sagemaker-tensorflow-2019-07-24-01-47-19-895)

Code

```
endpoint_name = 'sagemaker-tensorflow-2019-08-05-03-29-25-591'
```

Code

```
import sagemaker
```

```
from sagemaker.tensorflow.model import TensorFlowModel
predictor=sagemaker.tensorflow.model.TensorFlowPredictor(\
        endpoint_name, sagemaker_session)
```

13.2.9 Part 3: Test Model Deployment (optionally, outside of AWS)

Code

```
import json
import boto3
import numpy as np
import io

# see above, must be set to current value
endpoint_name = 'sagemaker-tensorflow-2019-08-05-03-29-25-591'

# Pick one of the following two cells to run based on how you will access...
```

Important, do not run both of the cells below!! Read comments

Code

```
# If you access the API from outside of AWS SageMaker
# notebooks you must authenticate and specify region...
# (do not run both this cell and the next)

client = boto3.client('runtime.sagemaker',
    region_name='us-east-1', # make sure to set correct region
    aws_access_key_id='AKIAYKSSG3L5P2H5EU77',
                        # These you get from AWS, for your account
    aws_secret_access_key='1GYDRaE1o/nFfW2nF6jAJpWrd2R5Eut/d6fS6ruL')
```

Code

```
# If you access from inside AWS in a notebook
# {do not run both this cell and the previous)
client = boto3.client('runtime.sagemaker', region_name='us-east-1')
# make sure to set correct region
```

13.2.10 Call the end point

Code

```
# Create a car based on one of the cars captured at beginning of this part.
data = [[8,307,130,3504,12,70,1]]
```

```
response = client.invoke_endpoint(EndpointName=endpoint_name, \
                                  Body=json.dumps(data))
response_body = response['Body']
print(response_body.read())
```

13.2.11 Additional Reading

For more information on the topics covered in this part, refer to the following web documents:

- Serving TensorFlow Models - Using Google's own deployment server.
- Deploy trained Keras or TensorFlow models using Amazon SageMaker - Using AWS to deploy TensorFlow ProtoBuffer models.
- Google ProtoBuf - The file format used to store neural networks for deployment.

13.3 Part 13.3: Using a Keras Deep Neural Network with a Web Application

In this part, we will extend the image API developed in Part 13.1 to work with a web application. This technique allows you to use a simple website to upload/predict images, such as Figure 13.3.

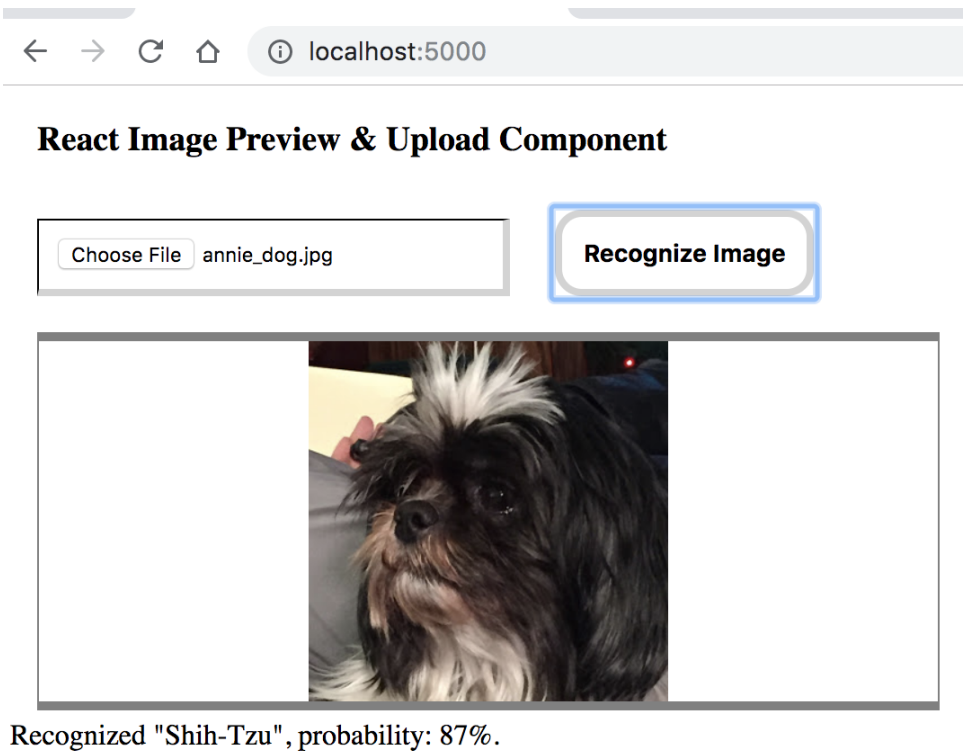


Figure 13.3: AI Web Application

To do this, we will use the same API developed in Module 13.1. However, we will now add a ReactJS website around it. This application is a single page web application that allows you to upload images for classification by the neural network. If you would like to read more about ReactJS and image uploading, you can refer to the blog post that provided some inspiration for this example. I added neural network functionality to a simple ReactJS image upload and preview example.

I built this example from the following components:

- GitHub Location for Web App
- image_web_server_1.py - The code both to start Flask and serve the HTML/JavaScript/CSS needed to provide the web interface.
- Directory WWW - Contains web assets.
  - index.html - The main page for the web application.
  - style.css - The stylesheet for the web application.
  - script.js - The JavaScript code for the web application.

13.4 Part 13.4: When to Retrain Your Neural Network

Dataset drift is a problem frequently seen in real-world applications of machine learning. Academic problems that courses typically present in school assignments usually do not experience this problem. For a class assignment, your instructor provides a single data set representing all of the data you will ever see for a task. In the real world, you obtain initial data to train your model; then, you will acquire new data over time that you use your model to predict.

Consider this example. You create a startup company that develops a mobile application that helps people find jobs. To train your machine learning model, you collect attributes about people and their careers. Once you have your data, you can prepare your neural network to suggest the best jobs for individuals.

Once your application is released, you will hopefully obtain new data. This data will come from job seekers using your app. These people are your customers. You have x values (their attributes), but you do not have y-values (their jobs). Your customers have come to you to find out what their be jobs will be. You will provide the customer's attributes to the neural network, and then it will predict their jobs. Usually, companies develop neural networks on initial data than use the neural network to perform predictions on new data obtained over time from their customers.

However, as time passes, companies must look if their model is still relevant. Your job prediction model will become less relevant as industry introduces new job types and the demographics of your customers change. This change in your underlying data is called dataset drift. In this section, we will see ways that you can measure dataset drift.

You can present your model with new data and see how its accuracy changes over time. However, to calculate efficiency, you must know the expected outputs from the model (y-values). For new data that you are obtaining in real-time, you may not know the correct outcomes. Therefore, we will look at algorithms that examine the x-inputs and determine how much they have changed in distribution from the original x-inputs that we trained on. These changes are called dataset drift.

Let's begin by creating generated data that illustrates drift. We present the following code to create a chart that shows such drift.

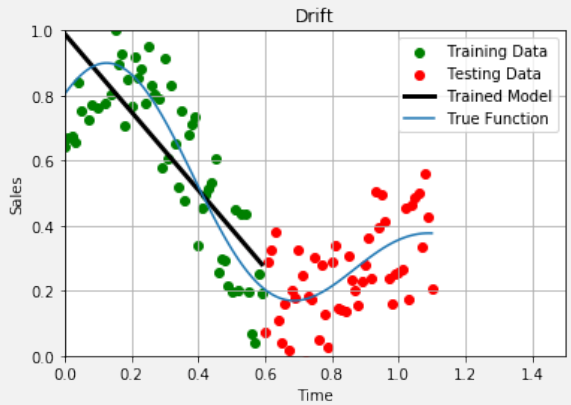
Code

```
import numpy as np

import matplotlib.pyplot as plot
from sklearn.linear_model import LinearRegression
```

```
def true_function(x):
    x2 = (x*8) - 1
    return ((np.sin(x2)/x2)*0.6)+0.3

#
x_train  = np.arange(0, 0.6, 0.01)
x_test  = np.arange(0.6, 1.1, 0.01)
x_true = np.concatenate( (x_train, x_test) )

#
y_true_train = true_function(x_train)
y_true_test = true_function(x_test)
y_true   = np.concatenate( (y_true_train, y_true_test) )

#
y_train = y_true_train + (np.random.rand(*x_train.shape)-0.5)*0.4
y_test  = y_true_test + (np.random.rand(*x_test.shape)-0.5)*0.4

#
lr_x_train = x_train.reshape((x_train.shape[0],1))
reg = LinearRegression().fit(lr_x_train, y_train)
reg_pred = reg.predict(lr_x_train)
print(reg.coef_[0])
print(reg.intercept_)

#
plot.xlim([0,1.5])
plot.ylim([0,1])
l1 = plot.scatter(x_train, y_train, c="g", label="Training Data")
l2 = plot.scatter(x_test, y_test, c="r", label="Testing Data")
l3, = plot.plot(lr_x_train, reg_pred, color='black', linewidth=3,
                label="Trained Model")
l4, = plot.plot(x_true, y_true, label = "True Function")
plot.legend(handles=[l1, l2, l3, l4])

#
plot.title('Drift')
plot.xlabel('Time')
plot.ylabel('Sales')
plot.grid(True, which='both')
plot.show()
```

Output

```
-1.1979470956001936
0.9888340153211445
```



The true-function represents what the data does over time. Unfortunately, you only have the training portion of the data. Your model will do quite well on the data that you trained it trained with; however, it will be very inaccurate on the new test data presented to it. The prediction line for the model fits the training data well but does not fit the est data well.

13.4.1 Preprocessing the Sberbank Russian Housing Market Data

The examples provided in this section use a Kaggle dataset named The Sberbank Russian Housing Market, which can be found at the following link.

- Sberbank Russian Housing Market

Kaggle datasets are already broken into training and test. We must load both of these files.

Code

```
import os
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder

PATH = "/Users/jheaton/Downloads/sberbank-russian-housing-market"


train_df = pd.read_csv(os.path.join(PATH,"train.csv"))
test_df = pd.read_csv(os.path.join(PATH,"test.csv"))
```

I provide a simple preprocess function that converts all numerics to z-scores and all categoricals to dummies.

Code

```
def preprocess(df):
    for i in df.columns:
        if df[i].dtype == 'object':
            df[i] = df[i].fillna(df[i].mode().iloc[0])
        elif (df[i].dtype == 'int' or df[i].dtype == 'float'):
```

```
            df[i] = df[i].fillna(np.nanmedian(df[i]))

    enc = LabelEncoder()
    for i in df.columns:
        if (df[i].dtype == 'object'):
            df[i] = enc.fit_transform(df[i].astype('str'))
            df[i] = df[i].astype('object')
```

Next, we run both the training and test datasets througn the preprocessing function.

Code

```
preprocess(train_df)
preprocess(test_df)
```

Finally, we remove thr target variable. We are only looking for drift on the x (input data).

Code

```
train_df.drop('price_doc',axis=1,inplace=True)
```

13.4.2 KS-Statistic

We will use the KS-Statistic to determine the difference in distribution between columns in the training and test sets. Just as a baseline, consider if we compare the same field to itself. I this case, we are comparing the **kitch_sq** in the training set. Because there is no difference in distribution between a field in itself, the p-value is 1.0, and the KS-Statistic statistic is 0. The P-Value is the probability that there is no difference between the two distributions. Typically some lower threshold is used for how low a P-Value is needed to reject the null hypothesis and assume there is a difference. The value of 0.05 is a standard threshold for p-values. Because the p-value is NOT below 0.05, we can expect the two distributions are the same. If the p-value were below the threshold, then the **statistic** value becomes interesting. This value tells you how different the two distributions are. A value of 0.0, in this case, means no differences.

Code

```
from scipy import stats

stats.ks_2samp(train_df['kitch_sq'], train_df['kitch_sq'])
```

Output

```
Ks_2sampResult(statistic=-0.0, pvalue=1.0)
```

Now let's do something more interesting. We will compare the same field **kitch_sq** between the test and training sets. In this case, the p-value is below 0.05, so the **statistic** value now contains the amount of difference detected.

Code

```
stats.ks_2samp(train_df['kitch_sq'], test_df['kitch_sq'])
```

Output

```
Ks_2sampResult(statistic=0.25829078867676714, pvalue=0.0)
```

Next, we pull the KS-Stat for every field. We also establish a boundary for the maximum p-value to display and how much of a difference is needed before we display the column.

Code

```
for col in train_df.columns:
    ks = stats.ks_2samp(train_df[col], test_df[col])
    if ks.pvalue < 0.05 and ks.statistic >0.1:
        print(f'{col}: {ks}')
```

Output

```
id: Ks_2sampResult(statistic=1.0, pvalue=0.0)
timestamp: Ks_2sampResult(statistic=0.8982081426022823, pvalue=0.0)
life_sq: Ks_2sampResult(statistic=0.2255084471628891,
pvalue=7.29401465948424e-271)
max_floor: Ks_2sampResult(statistic=0.17313454154786817,
pvalue=7.82000315371674e-160)
build_year: Ks_2sampResult(statistic=0.3176883950430345, pvalue=0.0)
num_room: Ks_2sampResult(statistic=0.1226755470309048,
pvalue=1.8622542043144584e-80)
kitch_sq: Ks_2sampResult(statistic=0.25829078867676714, pvalue=0.0)
state: Ks_2sampResult(statistic=0.13641341252952505,
pvalue=2.1968159319271184e-99)
preschool_quota: Ks_2sampResult(statistic=0.2364160801236304,
pvalue=1.1710777340471466e-297)
school_quota: Ks_2sampResult(statistic=0.25657342859882415,

...

cafe_sum_2000_max_price_avg:
Ks_2sampResult(statistic=0.10732529051140638,
pvalue=1.1100804327460878e-61)
cafe_avg_price_2000: Ks_2sampResult(statistic=0.1081218037860151,
pvalue=1.3575759911857293e-62)
```

13.4.3 Detecting Drift between Training and Testing Datasets by Training

Sample the training and test into smaller sets to train. We want 10K elements from each; however, the test set only has 7,662, so we only sample that amount from each side.

Code

```
SAMPLE_SIZE = min(len(train_df),len(test_df))
SAMPLE_SIZE = min(SAMPLE_SIZE,10000)
print(SAMPLE_SIZE)
```

Output

```
7662
```

We take the random samples from the training and test sets and add a flag called **source_training** to tell the two apart.

Code

```
training_sample = train_df.sample(SAMPLE_SIZE, random_state=49)
testing_sample = test_df.sample(SAMPLE_SIZE, random_state=48)

# Is the data from the training set?
training_sample['source_training'] = 1
testing_sample['source_training'] = 0
```

Next, we combine the data that we sampled from the training and test data sets and shuffle them.

Code

```
# Build combined training set
combined = testing_sample.append(training_sample)
combined.reset_index(inplace=True, drop=True)

# Now randomize
combined = combined.reindex(np.random.permutation(combined.index))
combined.reset_index(inplace=True, drop=True)
```

We will now generate x and y to train. We are attempting to predict the **source_training** value as y, which indicates if the data came from the training or test set. If the model is very successful at using the data to predict if it came from training or testing, then there is likely drift. Ideally, the train and test data should be indistinguishable.

Code

```
# Get ready to train
y = combined['source_training'].values
combined.drop('source_training',axis=1,inplace=True)
x = combined.values
```

Code

```
y
```

Output

```
array([1, 1, 1, ..., 1, 0, 0])
```

We will consider anything above a 0.75 AUC as having a good chance of drift.

Code

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

model = RandomForestClassifier(n_estimators = 60, max_depth = 7,
    min_samples_leaf = 5)
lst = []

for i in combined.columns:
    score = cross_val_score(model,pd.DataFrame(combined[i]),y,cv=2,
                            scoring='roc_auc')
    if (np.mean(score) > 0.75):
        lst.append(i)
        print(i,np.mean(score))
```

Output

```
id 1.0
timestamp 0.9601862111975688
full_sq 0.7966785611424911
life_sq 0.8724218330166038
build_year 0.8004825176688191
kitch_sq 0.9070093804672634
cafe_sum_500_min_price_avg 0.8435920036035689
cafe_avg_price_500 0.8453533835344671
```

13.5 Part 13.5: Using a Keras Deep Neural Network with a Web Application

In this section, we will see how to deploy a neural network to an iOS mobile device. Android is also another option that I plan to support at some point. However, for now, I am focusing on iOS. Apple added its CoreML library that makes it considerably easier to deploy a deep neural network than it used to be. The example in this part will focus on creating a simple computer vision mobile application for image recognition. All computation will occur on the actual device. Compared with "cloud compute," which occurs in the cloud,

edge computing occurs on physical devices the user has physical access to.

Apple makes several pre-trained neural networks available for CoreML. It is also possible to convert Keras models into the format needed by CoreML. For this example, we will convert a Keras pre-trained model to CoreML. This technique gives a good demonstration of this conversion that can use for other Keras models that you've created.

Please note the following two requirements set forth by Apple.

- You will need a Mac running XCode to create an iOS application.
- You must have a free Apple Developer account to deploy your app to your iOS device. Sign up here.
- To add your application to the Apple App Store and deploy to other people's hardware, you mist enroll in the 100 USD/year developer program.

13.5.1 Converting Keras to CoreML

The following code exports the MobileNet network to an H5 file.

```
conda create -y --name coreml python=3.6
source activate coreml
conda install -y jupyter
conda install -y scipy
pip install --exists-action i --upgrade sklearn
pip install --exists-action i --upgrade pandas
pip install --exists-action i --upgrade pandas-datareader
pip install --exists-action i --upgrade matplotlib
pip install --exists-action i --upgrade pillow
pip install --exists-action i --upgrade tqdm
pip install --exists-action i --upgrade requests
pip install --exists-action i --upgrade h5py
pip install --exists-action i --upgrade pyyaml
pip install --exists-action i --upgrade tensorflow==1.14
pip install --exists-action i --upgrade keras==2.2.4
pip install --exists-action i --upgrade coremltools
conda update -y --all
python -m ipykernel install --user --name coreml
       --display-name "Python 3.6 (coreml)"
```

Code

```
# Export MobileNet to an H5 file
import os
from keras.applications import MobileNet

save_path = "./dnn/"
model = MobileNet(weights='imagenet',include_top=True)
model.save(os.path.join(save_path,"mobilenet.h5"))
```

Unfortunately, as of August 2019, CoreML does not support TensorFlow 2.0.

Code

```
import tensorflow as tf
import keras
import coremltools

print(f"TensorFlow version: {tf.__version__}")
print(f"Keras version: {keras.__version__}")
```

Output

```
TensorFlow version: 1.14.0
Keras version: 2.2.4
```

Code

```
import requests
r = requests.get('https://data.heatonresearch.com/data'+\
                 '/t81-558/imagenet_class_index.json')

js = r.json()

lookup = ['' for x in range(1000)]
for idx in js:
    lookup[int(idx)] = js[idx][1]
```

Code

```
coreml_model = coremltools.converters.keras.convert(model,
    input_names="image",
    image_input_names="image",
    image_scale=1/255.0,
    class_labels=lookup,
    is_bgr=True)
```

Output

```
0 : input_1, <keras.engine.input_layer.InputLayer object at
0xa2ef30b70>
1 : conv1_pad, <keras.layers.convolutional.ZeroPadding2D object at
0xa2ef30fd0>
2 : conv1, <keras.layers.convolutional.Conv2D object at 0xa2ef30d68>
3 : conv1_bn, <keras.layers.normalization.BatchNormalization object at
0xa2ed79240>
4 : conv1_relu, <keras.layers.advanced_activations.ReLU object at
0xa2ed796d8>
```

```
5 : conv_dw_1, <keras.layers.convolutional.DepthwiseConv2D object at
0xa2ed79898>
6 : conv_dw_1_bn, <keras.layers.normalization.BatchNormalization
object at 0xa2ed79550>
7 : conv_dw_1_relu, <keras.layers.advanced_activations.ReLU object at
0xa2eff9e80>

...

88 : reshape_1, <keras.layers.core.Reshape object at 0xa309d9a90>
89 : conv_preds, <keras.layers.convolutional.Conv2D object at
0xa309c3278>
90 : reshape_2, <keras.layers.core.Reshape object at 0xa30ad2978>
91 : act_softmax, <keras.layers.core.Activation object at 0xa30ad2f60>
```

Code

```
coreml_model.save(os.path.join(save_path,"mobilenet.mlmodel"))
```

13.5.2 Creating an IOS CoreML Application

We will now use the neural network created in the last section to create an IOS application that will classify what its camera sees. This application will be a single image classification, not the multi-image classification that we saw with YOLO. Figure 13.4 shows this application running on my iPhone here:

You can find the complete source code (in XCode) for this application at the following URL:

- GitHub: IOS Classify

To create this application from scratch (in XCode), follow these steps:

- Install XCode
- Register for Apple Developer account (if you wish to deploy to iOS device)
- Create a new XCode Project
- Delete storyboard
- Remove project references to storyboard
- Add camera prompt to security settings
- Replace the contents of the view controller with the included
- Test on IOS device

The YouTube video for this module goes through the above process.

13.5.3 More Reading

There are several excellent tutorials on IOS and CoreML development. The following articles were beneficial in the creation of this material.

- Running Keras models on iOS with CoreML
- How to build an image recognition iOS app with Apple's CoreML and Vision APIs

Figure 13.4: IOS Image Classify

Chapter 14

Other Neural Network Techniques

14.1 Part 14.1: What is AutoML

Automatic Machine Learning (AutoML) attempts to use machine learning to automate itself. Data is passed to the AutoML application in raw form and models are automatically generated.

14.1.1 AutoML from your Local Computer

The following AutoML applications are commercial.

- Rapid Miner - Free student version available.
- Dataiku - Free community version available.
- DataRobot - Commercial
- H2O Driverless - Commercial

14.1.2 AutoML from Google Cloud

- Google Cloud AutoML Tutorial

14.1.3 A Simple AutoML System

The following program is a very simple implementation of AutoML. It is able to take RAW tabular data and construct a neural network.

We begin by defining a class that abstracts the differences between reading CSV over local file system or HTTP/HTTPS.

Code

```
import requests
import csv

class CSVSource():
    def __init__(self, filename):
        self.filename = filename
    def __enter__(self):
        if self.filename.lower().startswith("https:")
```

```
            or self.filename.lower().startswith("https:"):
            r = requests.get(self.filename, stream=True)
            self.infile = (line.decode('utf-8') for line in r.iter_lines())
            return csv.reader(self.infile)
        else:
            self.infile = codecs.open(self.filename, "r", "utf-8")
            return csv.reader(self.infile)
    def __exit__(self, type, value, traceback):
        self.infile.close()
```

The following code analyzes the tabular data and determines a way of encoding the feature vector.

Code

```
import csv
import codecs
import math
import os
import re
from numpy import genfromtxt

MAX_UNIQUES = 200

INPUT_ENCODING = 'latin-1'

CMD_CAT_DUMMY = 'dummy-cat'
CMD_CAT_NUMERIC = 'numeric-cat'
CMD_IGNORE = 'ignore'
CMD_MAP = 'map'
CMD_PASS = 'pass'
CMD_BITS = 'bits'

CONTROL_INDEX = 'index'
CONTROL_NAME = 'name'
CONTROL_COMMAND = 'command'
CONTROL_TYPE = 'type'
CONTROL_LENGTH = 'length'
CONTROL_UNIQUE_COUNT = 'unique_count'
CONTROL_UNIQUE_LIST = 'unique_list'
CONTROL_MISSING = 'missing'
CONTROL_MEAN = 'mean'
CONTROL_SDEV = 'sdev'


MAP_SKIP = True
MISSING_SKIP = False

current_row = 0
```

```
def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False


def isna(s):
    return s.upper() == 'NA' or s.upper() == 'N/A'
        or s.upper() == 'NULL' or len(s) < 1 or s.upper() == '?'

def analyze(filename):
    fields = []
    first_header = None

    # Pass 1 (very short. First, look at the first row of each of the
    # provided files.
    # Build field blocks from the first file, and ensure that other files
    # match the first one.

    with CSVSource(filename) as reader:
        header = next(reader)

        if first_header is None:
            first_header = header

            for idx, field_name in enumerate(header):
                fields.append({
                    'name': field_name,
                    'command': '?',
                    'index': idx,
                    'type': None,
                    'missing': False,
                    'unique': {},
                    'count': 0,
                    'mean': '',
                    'sum': 0,
                    'sdev': '',
                    'length': 0})
        else:
            for x, y in zip(header, first_header):
                if x != y:
                    raise ValueError(\
                        'The␣headers␣do␣not␣match␣on␣the␣input␣files')


    # Pass 2 over the files
```

```
    with CSVSource(filename) as reader:
        next(reader)

        # Determine types and calculate sum
        for row in reader:
            if len(row) != len(fields):
                continue
            for data, field_info in zip(row, fields):
                data = data.strip()
                field_info['length'] = max(len(data),field_info['length'])
                if len(data) < 1 or data.upper() == 'NULL' or isna(data):
                    field_info[CONTROL_MISSING] = True
                else:
                    if not is_number(data):
                        field_info['type'] = 'text'

                    # Track the unique values and counts per unique item
                    cat_map = field_info['unique']
                    if data in cat_map:
                        cat_map[data]['count']+=1
                    else:
                        cat_map[data] = {'name':data,'count':1}

                    if field_info['type'] != 'text':
                        field_info['count'] += 1
                        field_info['sum'] += float(data)

    # Finalize types
    for field in fields:
        if field['type'] is None:
            field['type'] = 'numeric'
        field[CONTROL_UNIQUE_COUNT] = len(field['unique'])

    # Calculate mean
    for field in fields:
        if field['type'] == 'numeric' and field['count'] > 0:
            field['mean'] = field['sum'] / field['count']


    # Pass 3 over the files, calculate standard deviation and
    # finailize fields.
    sums = [0] * len(fields)

    with CSVSource(filename) as reader:
        next(reader)

        for row in reader:
            if len(row) != len(fields):
```

```
                continue
            for data, field_info in zip(row, fields):
                data = data.strip()
                if field_info['type'] == 'numeric'
                        and len(data) > 0 and not isna(data):
                    sums[field_info['index']] += (float(data) - \
                                    field_info['mean']) ** 2

    # Examine fields
    for idx, field in enumerate(fields):
        if field['type'] == 'numeric' and field['count'] > 0:
            field['sdev'] = math.sqrt(sums[field['index']] / field['count'])

        # Assign a default command
        if field['name'] == 'ID' or field['name'] == 'FOLD':
            field['command'] = 'pass'
        elif "DATE" in field['name'].upper():
            field['command'] = 'date'
        elif field['unique_count'] == 2 and field['type'] == 'numeric':
            field['command'] = CMD_PASS
        elif field['type'] == 'numeric' and field['unique_count'] < 25:
            field['command'] = CMD_CAT_DUMMY
        elif field['type'] == 'numeric':
            field['command'] = 'zscore'
        elif field['type'] == 'text' and field['unique_count'] \
                <= MAX_UNIQUES:
            field['command'] = CMD_CAT_DUMMY
        else:
            field['command'] = CMD_IGNORE

    return fields

def write_control_file(filename, fields):
    with codecs.open(filename, "w", "utf-8") as outfile:
        writer = csv.writer(outfile,quoting=csv.QUOTE_NONNUMERIC)

        writer.writerow([CONTROL_INDEX, CONTROL_NAME, CONTROL_COMMAND,
                         CONTROL_TYPE, CONTROL_LENGTH, CONTROL_UNIQUE_COUNT,
                         CONTROL_MISSING, CONTROL_MEAN, CONTROL_SDEV])
        for field in fields:

            # Write the main row for the field (left-justified)
            writer.writerow([field[CONTROL_INDEX], field[CONTROL_NAME],
                             field[CONTROL_COMMAND], field[CONTROL_TYPE],
                              field[CONTROL_LENGTH],
                                  field[CONTROL_UNIQUE_COUNT],
                              field[CONTROL_MISSING], field[CONTROL_MEAN],
                              field[CONTROL_SDEV]])
```

```
            # Write out any needed category information
            if field[CONTROL_UNIQUE_COUNT] <= MAX_UNIQUES:
                sorted_cat = field['unique'].values()
                sorted_cat = sorted(sorted_cat, key=lambda k:
                                    k[CONTROL_NAME])
                for category in sorted_cat:
                    writer.writerow(["","",
                        category[CONTROL_NAME],
                            category['count']])
            else:
                catagories = ""



def read_control_file(filename):
    with codecs.open(filename, "r", "utf-8") as infile:
        reader = csv.reader(infile)
        header = next(reader)

        lookup = {}
        for i, name in enumerate(header):
            lookup[name] = i

        fields = []
        categories = {}

        for row in reader:
            if row[0] == '':
                name = row[2]
                mp = '' if len(row)<=4 else row[4]
                categories[name] = {'name':name,'count':int(row[3]),
                                    'map':mp}
                if len(categories)>0:
                    field[CONTROL_UNIQUE_LIST] = \
                        sorted(categories.keys())
            else:
                # New field
                field = {}
                categories = {}
                field['unique'] = categories
                for key in lookup.keys():
                    value = row[lookup[key]]
                    if key in ['unique_count', 'count', 'index', 'length']:
                        value = int(value)
                    elif key in ['sdev', 'mean', 'sum']:
                        if len(value) > 0:
                            value = float(value)
```

```
                    field[key] = value

                field['len'] = -1
                fields.append(field)
        return fields

def header_cat_dummy(field, header):
    name = str(field['name'])

    for c in field['unique']:
        dname = "{}-D:{}".format(name, c)
        header.append(dname)

def header_bits(field, header):
    for i in range(field['length']):
        header.append("{}-B:{}".format(field['name'], i))


def header_other(field, header):
    header.append(field['name'])


def column_zscore(field,write_row,value,has_na):
    if isna(value) or field['sdev'] == 0:
        #write_row.append('NA')
        #has_na = True
        write_row.append(0)
    elif not is_number(value):
        raise ValueError("Row {}: Non-numeric for zscore: {}"\
            " on field {}".format(current_row,value,field['name']))
    else:
        value = (float(value) - field['mean']) / field['sdev']
        write_row.append(value)
    return has_na

def column_cat_numeric(field,write_row,value,has_na):
    if CONTROL_UNIQUE_LIST not in field:
        raise ValueError("No value list, can't encode {}"\
            " to numeric categorical.".format(field[CONTROL_NAME]))

    if value not in field[CONTROL_UNIQUE_LIST]:
        write_row.append("NA")
        has_na = True
    else:
        idx = field[CONTROL_UNIQUE_LIST].index(value)
        write_row.append('class-' + str(idx))
    return has_na
```

```
def column_map(field,write_row,value,has_na):
    if value in field['unique']:
        mapping = field['unique'][value]['map']
        write_row.append(mapping)
    else:
        write_row.append("NA")
        return True
    return has_na


def column_cat_dummy(field,write_row,value,has_na):
    for c in field['unique']:
        write_row.append(0 if value != c else 1)
    return has_na

def column_bits(field,write_row,value,has_na):
    if len(value)!=field['length']:
        raise ValueError("Invalid bits length: {}, expected: {}".format(
            len(value),field['length']))

    for c in value:
        if c == 'Y':
            write_row.append(1)
        elif c == 'N':
            write_row.append(-1)
        else:
            write_row.append(0)
    return has_na

def transform_file(input_file, output_file, fields):
    print("**Transforming to file: {}".format(output_file))
    with CSVSource(input_file) as reader, \
            codecs.open(output_file, "w", "utf-8") as outfile:
        writer = csv.writer(outfile)

        next(reader)
        header = []

        # Write the header
        for field in fields:
            if field['command'] == CMD_IGNORE:
                pass
            elif field['command'] == CMD_CAT_DUMMY:
                header_cat_dummy(field,header)
            elif field['command'] == CMD_BITS:
                header_bits(field,header)
            else:
                header_other(field,header)
```

```
        print("Columns generated: {}".format(len(header)))

        writer.writerow(header)
        line_count = 0
        lines_skipped = 0

        # Process the actual file
        current_row = -1
        header_len = len(header)
        for row in reader:
            if len(row) != len(fields):
                continue

            current_row+=1
            has_na = False
            write_row = []
            for field in fields:
                value = row[field['index']].strip()

                cmd = field['command']
                if cmd == 'zscore':
                    has_na = column_zscore(field,write_row,value, has_na)
                elif cmd == CMD_CAT_NUMERIC:
                    has_na = column_cat_numeric(field,write_row,value, \
                                                has_na)
                elif cmd == CMD_IGNORE:
                    pass
                elif cmd == CMD_MAP:
                    has_na = column_map(field,write_row,value, has_na)
                elif cmd == CMD_PASS:
                    write_row.append(value)
                elif cmd == 'date':
                    write_row.append(str(value[-4:]))
                elif cmd == CMD_CAT_DUMMY:
                    has_na = column_cat_dummy(field,write_row,value,
                                              has_na)
                elif cmd == CMD_BITS:
                    has_na = column_bits(field,write_row,value,has_na)
                else:
                    raise ValueError(\
                        "Unknown command: {}, stopping.".format(cmd))


            if MISSING_SKIP and has_na:
                lines_skipped += 1
                pass
            else:
```

```
                line_count += 1
                writer.writerow(write_row)

                # Double check!
                if len(write_row) != header_len:
                    raise ValueError("Inconsistant column "\
                        "count near line: {}, only had: {}" \
                        .format(line_count,len(write_row)))

    print("Data rows written: {}, skipped: {}"\
          .format(line_count,lines_skipped))
    print()

def find_field(control, name):
    for field in control:
        if field['name'] == name:
            return field
    return None

def find_transformed_fields(header, name):
    y = []
    x = []
    for idx, field in enumerate(header):
        if field.startswith(name + '-') or field==name:
            y.append(idx)
        else:
            x.append(idx)

    return x,y

def process_for_fit(control, transformed_file, target):

    with CSVSource(transformed_file) as reader:
        header = next(reader)

    field = find_field(control, target)
    if field is None:
        raise ValueError(f"Unknown target column specified:{target}")

    if field['command'] == 'dummy-cat':
        print(f"Performing classification on: {target}")
    else:
        print(f"Performing regression on: {target}")

    x_ids, y_ids = find_transformed_fields(header, target)

    x = genfromtxt("transformed.csv", delimiter=',', skip_header=1)
    y = x[:,y_ids]
```

```
    x = x[:,x_ids]
    return x,y
```

The following code takes the data processed from above and trains a neural network.

Code

```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn import metrics
from sklearn.model_selection import KFold

def generate_network(x,y,task):
    model = Sequential()
    model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
    model.add(Dense(25, activation='relu')) # Hidden 2

    if task == 'classify':
        model.add(Dense(y.shape[1],activation='softmax')) # Output
        model.compile(loss='categorical_crossentropy', optimizer='adam')
    else:
        model.add(Dense(1))
        model.compile(loss='mean_squared_error', optimizer='adam')

    return model

def cross_validate(x,y,folds,task):

    if task == 'classify':
        cats = y.argmax(axis=1)
        kf = StratifiedKFold(folds, shuffle=True, random_state=42).split(\
                x,cats)
    else:
        kf = KFold(folds, shuffle=True, random_state=42).split(x)

    oos_y = []
    oos_pred = []
    fold = 0

    for train, test in kf:
        fold+=1
        print(f"Fold #{fold}")

        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
```

```
        y_test = y[test]

        model = generate_network(x,y,task)
        model.fit(x_train,y_train,validation_data=(x_test,y_test),verbose=0,
                  epochs=500)

        pred = model.predict(x_test)

        oos_y.append(y_test)

        if task == 'classify':
            # raw probabilities to chosen class (highest probability)
            pred = np.argmax(pred,axis=1)
        oos_pred.append(pred)

        if task == 'classify':
            # Measure this fold's accuracy
            y_compare = np.argmax(y_test,axis=1) # For accuracy calculation
            score = metrics.accuracy_score(y_compare, pred)
            print(f"Fold score (accuracy): {score}")
        else:
            score = np.sqrt(metrics.mean_squared_error(pred,y_test))
            print(f"Fold score (RMSE): {score}")


    # Build the oos prediction list and calculate the error.
    oos_y = np.concatenate(oos_y)
    oos_pred = np.concatenate(oos_pred)

    if task == 'classify':
        oos_y_compare = np.argmax(oos_y,axis=1) # For accuracy calculation
        score = metrics.accuracy_score(oos_y_compare, oos_pred)
        print(f"Final score (accuracy): {score}")
    else:
        score = np.sqrt(metrics.mean_squared_error(oos_y, oos_pred))
        print(f"Final score (RMSE): {score}")
```

14.1.4 Running My Sample AutoML Program

These three variables are all you really need to define.

Code

```
SOURCE_DATA = \
'https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv'
TARGET_FIELD = 'product'
TASK = 'classify'
```

```
#SOURCE_DATA = 'https://data.heatonresearch.com/data/t81-558/iris.csv'
#TARGET_FIELD = 'species'
#TASK = 'classify'

#SOURCE_DATA = 'https://data.heatonresearch.com/data/t81-558/auto-mpg.csv'
#TARGET_FIELD = 'mpg'
#TASK = 'reg'
```

The following lines of code analyze your source data file and figure out how to encode each column. The result is a control file that you can modify to control how each column is handled. The below code should only be run ONCE to generate a control file as a starting point for you to modify.

Code

```
import csv
import requests
import codecs

control = analyze(SOURCE_DATA)
write_control_file("control.csv",control)
```

If your control file is already create, you can start here (after defining the above constants). Do not rerun the previous section, as it will overwrite your control file. Now transform the data.

Code

```
control = read_control_file("control.csv")
transform_file(SOURCE_DATA,"transformed.csv",control)
```

Output

```
**Transforming to file: transformed.csv
Columns generated: 59
Data rows written: 2000, skipped: 0
```

Load the transformed data into properly preprocessed x and y.

Code

```
x,y = process_for_fit(control, "transformed.csv", TARGET_FIELD)
print(x.shape)
print(y.shape)
```

Output

```
Performing classification on: product
(2000, 52)
(2000, 7)
```

Double check to be sure there are no missing values remaining.

Code

```
import numpy as np
np.isnan(x).any()
```

Output

```
False
```

We are now ready to cross validate and train.

Code

```
cross_validate(x,y,5,TASK)
```

Output

```
Fold #1
Fold score (accuracy): 0.6915422885572139
Fold #2
Fold score (accuracy): 0.7064676616915423
Fold #3
Fold score (accuracy): 0.6807980049875312
Fold #4
Fold score (accuracy): 0.6658291457286433
Fold #5
Fold score (accuracy): 0.6675062972292192
Final score (accuracy): 0.6825
```

14.2 Part 14.2: Using Denoising AutoEncoders in Keras

14.2.1 Function Approximation

Function approximation is perhaps the original task of machine learning. Long before there were computers and even a notion of machine learning, scientists were coming up with equations to fit their observations of nature. Scientists find equations to demonstrate correlations between observations. For example, a variety of equations relate mass, acceleration, and force.

Looking at complex data and deriving an equation does take some technical expertise. The goal of function approximation is to remove intuition from the process and instead depend on an algorithmic method to generate an equation that describes data automatically. A regression neural network performs this task.

We begin by creating a function that we will use to chart a regression function.

Code

```
# Regression chart.
```

```
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()
```

Next, we will attempt to approximate a slightly random variant of the trigonometric sine function.

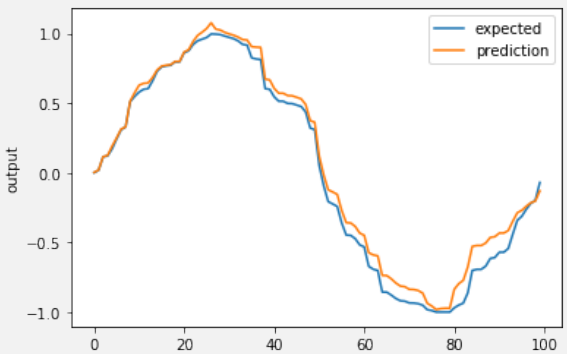
Code

```
import tensorflow as tf
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt

rng = np.random.RandomState(1)
x = np.sort((360 * rng.rand(100, 1)), axis=0)
y = np.array([np.sin(x*(np.pi/180.0)).ravel()]).T

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=0,batch_size=len(x),epochs=25000)

pred = model.predict(x)

print("Actual")
print(y[0:5])

print("Pred")
print(pred[0:5])

chart_regression(pred.flatten(),y,sort=False)
```

Output

```
Actual
[[0.00071864]
```

```
 [0.01803382]
 [0.11465593]
 [0.1213861 ]
 [0.1712333 ]]
Pred
[[0.00317574]
 [0.02026811]
 [0.11543235]
 [0.12497211]
 [0.19133109]]
```



As you can see, the neural network creates a reasonably close approximation of the random sine function.

14.2.2 Multi-Output Regression

Unlike most models, neural networks can provide multiple regression outputs. This feature allows a neural network to generate multiple outputs for the same input. For example, you might train the MPG data set to predict both MPG and horsepower. One area that multiple regression outputs can be useful for is autoencoders. The following diagram shows a multi-regression neural network. As you can see, there are multiple output neurons. Usually, you will use multiple output neurons for classification. Each output neuron will represent the probability of one of the classes. However, in this case, it is a regression neural network. Figure 13.MRG shows multi-output regression.

The following program uses a multi-output regression to predict both sin and cos from the same input data.

Code

```
from sklearn import metrics

rng = np.random.RandomState(1)
x = np.sort((360 * rng.rand(100, 1)), axis=0)
y = np.array([np.pi * np.sin(x*(np.pi/180.0)).ravel(), np.pi \
              * np.cos(x*(np.pi/180.0)).ravel()]).T

model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))
```

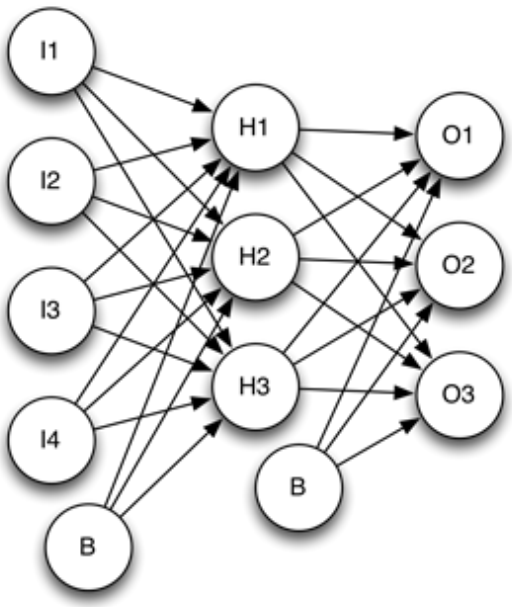


Figure 14.1: Multi-Output Regression

```
model.add(Dense(2)) # Two output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=0,batch_size=len(x),epochs=25000)


# Fit regression DNN model.
pred = model.predict(x)

score = np.sqrt(metrics.mean_squared_error(pred, y))
print("Score (RMSE): {}".format(score))

np.set_printoptions(suppress=True)

print("Predicted:")
print(np.array(pred[20:25]))

print("Expected:")
print(np.array(y[20:25]))
```

Output

```
Score (RMSE): 0.0607553517426597
Predicted:
```

```
[[2.6972563  1.5949173 ]
 [2.7385583  1.5186492 ]
 [2.884924   1.2355493 ]
 [2.9632306  1.0122485 ]
 [3.0012193  0.88598156]]
Expected:
[[2.70765313 1.59317888]
 [2.75138445 1.51640628]
 [2.89299999 1.22480835]
 [2.97603942 1.00637655]
 [3.01381723 0.88685404]]
```

14.2.3 Simple Autoencoder

An autoencoder is a neural network that has the same number of input neurons as it does outputs. The hidden layers of the neural network will have fewer neurons than the input/output neurons. Because there are fewer neurons, the auto-encoder must learn to encode the input to the fewer hidden neurons. The predictors (x) and output (y) are exactly the same in an autoencoder. Because of this, we consider autoencoders to be unsupervised. Figure 14.2 shows an autoencoder.

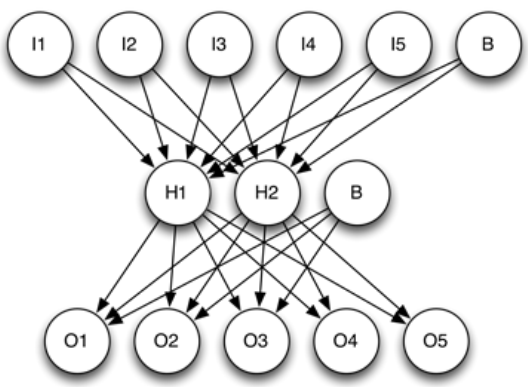


Figure 14.2: Simple Auto Encoder

The following program demonstrates a very simple autoencoder that learns to encode a sequence of numbers. Fewer hidden neurons will make it much more difficult for the autoencoder to learn.

Code

```
from sklearn import metrics
import numpy as np
import pandas as pd
from IPython.display import display, HTML
import tensorflow as tf

x = np.array([range(10)]).astype(np.float32)
```

```
print(x)

model = Sequential()
model.add(Dense(3, input_dim=x.shape[1], activation='relu'))
model.add(Dense(x.shape[1])) # Multiple output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,x,verbose=0,epochs=1000)

pred = model.predict(x)
score = np.sqrt(metrics.mean_squared_error(pred,x))
print("Score (RMSE): {}".format(score))
np.set_printoptions(suppress=True)
print(pred)
```

Output

```
[[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]]
Score (RMSE): 0.8473137021064758
[[0.0133801 1.153795  2.0281403 3.0298407 4.1182265 5.0704904 6.360253
  7.321521  8.555862  6.4323545]]
```

14.2.4 Autoencode (single image)

We are now ready to build a simple image autoencoder. The program below learns a capable encoding for the image. You can see the distortions that occur.

Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
from tensorflow.keras.optimizers import SGD
import requests
from io import BytesIO

url = "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))
img.load()
img = img.resize((128,128), Image.ANTIALIAS)
img_array = np.asarray(img)
img_array = img_array.flatten()
img_array = np.array([ img_array ])
img_array = img_array.astype(np.float32)
print(img_array.shape[1])
print(img_array)
```

```
model = Sequential()
model.add(Dense(10, input_dim=img_array.shape[1], activation='relu'))
model.add(Dense(img_array.shape[1])) # Multiple output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(img_array,img_array,verbose=0,epochs=20)

print("Neural network output")
pred = model.predict(img_array)
print(pred)
print(img_array)
cols,rows = img.size
img_array2 = pred[0].reshape(rows,cols,3)
img_array2 = img_array2.astype(np.uint8)
img2 = Image.fromarray(img_array2, 'RGB')
img2
```

Output

```
49152
[[ 84. 134. 177. ...   6.  15.  14.]]
Neural network output
[[114.38295   124.415985  103.903145  ...  11.447948    4.4970107
   19.995102 ]]
[[ 84. 134. 177. ...   6.  15.  14.]]
```

14.2.5 Standardize Images

When processing several images together, it is sometimes essential to standardize them. The following code reads a sequence of images and causes them to all be of the same size and perfectly square. If the input images are not square, cropping will occur.

Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML
```

```
#url = "http://www.heatonresearch.com/images/about-jeff.jpg"

images = [
    "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/ff/"+\
        "WashU_Graham_Chapel.JPG",
    "https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg"
]


def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
        img = img.crop((pad,0,cols,cols))
    else:
        pad = (cols-rows)/2
        img = img.crop((0,pad,rows,rows))

    return img

x = []

for url in images:
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = make_square(img)
    img = img.resize((128,128), Image.ANTIALIAS)
    print(url)
    display(img)
    img_array = np.asarray(img)
    img_array = img_array.flatten()
    img_array = img_array.astype(np.float32)
    img_array = (img_array-128)/128
    x.append(img_array)


x = np.array(x)

print(x.shape)
```

Output

https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg

https://upload.wikimedia.org/wikipedia/commons/f/ff/WashU_Graham_Chapel.JPG

https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg

https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg

https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg

https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg

```
https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg
```

```
(7, 49152)
```

14.2.6 Image Autoencoder (multi-image)

Autoencoders can learn the same encoding for multiple images. The following code learns a single encoding for numerous images.

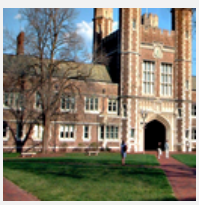
Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO
from sklearn import metrics
import numpy as np
import pandas as pd
import tensorflow as tf
from IPython.display import display, HTML

# Fit regression DNN model.
print("Creating/Training neural network")
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
model.add(Dense(x.shape[1])) # Multiple output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,x,verbose=0,epochs=1000)

print("Score neural network")
pred = model.predict(x)

cols,rows = img.size
for i in range(len(pred)):
```

```
    print(pred[i])
    img_array2 = pred[i].reshape(rows,cols,3)
    img_array2 = (img_array2*128)+128
    img_array2 = img_array2.astype(np.uint8)
    img2 = Image.fromarray(img_array2, 'RGB')
    display(img2)
```

Output

```
Creating/Training neural network
Score neural network
[-0.14099729  0.2504178   0.58697623 ... -0.80495536 -0.8600077
 -0.7507377 ]
```



```
[ 0.1245165   0.05581025 -0.9469281  ... -0.5964093  -0.77362233
 -0.87149537]
```

```
[-0.16287911  0.07083478  0.8028359  ...  0.15529267 -0.1485157
 -0.63872755]
```

```
[-0.40565515  0.08525646  0.6543999  ... -0.12393776 -0.22541034
 -0.34251797]
```

```
[ 0.28899017  0.38252142  0.7866407  ... -0.2032861  -0.2651671
 -0.82580376]
```

```
[ 0.9917432   0.9890273   0.982688   ... -0.18442157 -0.204146
 -0.20103177]
```

```
[-0.78713    -0.38974318  0.52933806 ...  0.6468072   0.20250452
 -0.35054564]
```

14.2.7 Adding Noise to an Image

Autoencoders can handle noise. First, it is essential to see how to add noise to an image. There are many ways to add such noise. The following code adds random black squares to the image to produce noise.

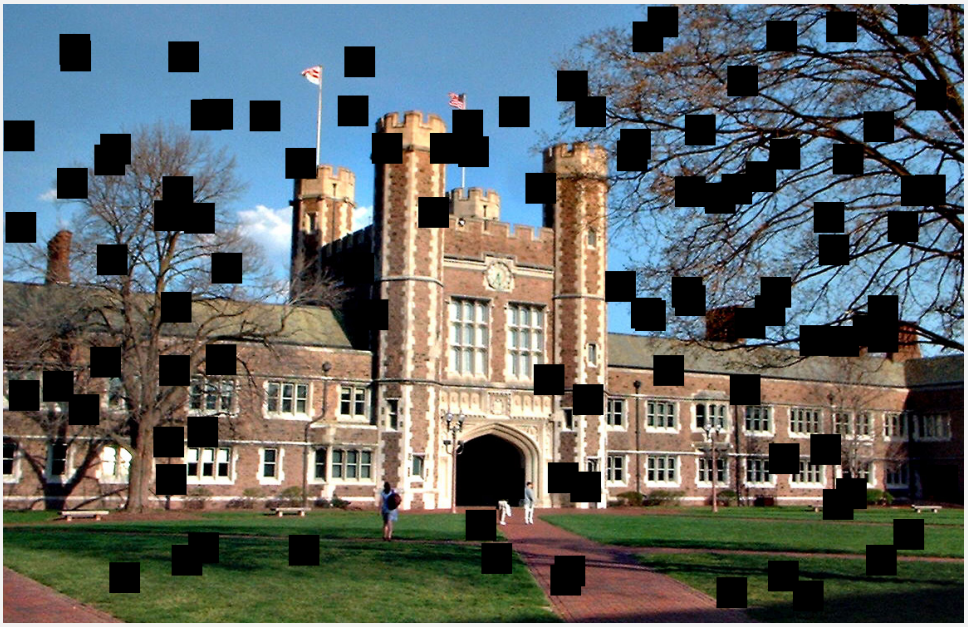
Code

```
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO

%matplotlib inline


def add_noise(a):
    a2 = a.copy()
    rows = a2.shape[0]
    cols = a2.shape[1]
    s = int(min(rows,cols)/20) # size of spot is 1/20 of smallest dimension
```

```
    for i in range(100):
        x = np.random.randint(cols-s)
        y = np.random.randint(rows-s)
        a2[y:(y+s),x:(x+s)] = 0

    return a2

url = "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg"
#url = "http://www.heatonresearch.com/images/about-jeff.jpg"

response = requests.get(url)
img = Image.open(BytesIO(response.content))
img.load()

img_array = np.asarray(img)
rows = img_array.shape[0]
cols = img_array.shape[1]

print("Rows: {}, Cols: {}".format(rows,cols))

# Create new image
img2_array = img_array.astype(np.uint8)
print(img2_array.shape)
img2_array = add_noise(img2_array)
img2 = Image.fromarray(img2_array, 'RGB')
img2
```

Output

```
Rows: 744, Cols: 1157
(744, 1157, 3)
```



14.2.8 Denoising Autoencoder

You design a denoising autoencoder to remove noise from input signals. The y becomes each image/signal (just like a normal autoencoder); however, the x becomes a version of y with noise added. Noise is artificially added to the images to produce x. The following code creates ten noisy versions of each of the images. You train the network to convert noisy data (x) to the original input (y).

Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML

#url = "http://www.heatonresearch.com/images/about-jeff.jpg"

images = [
    "https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/ff/" +
        "WashU_Graham_Chapel.JPG",
    "https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg",
    "https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg"
```

```
]


def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
        img = img.crop((pad,0,cols,cols))
    else:
        pad = (cols-rows)/2
        img = img.crop((0,pad,rows,rows))

    return img

x = []
y = []
loaded_images = []

for url in images:
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url)
    img = Image.open(BytesIO(response.content))
    img.load()
    img = make_square(img)
    img = img.resize((128,128), Image.ANTIALIAS)

    loaded_images.append(img)
    print(url)
    display(img)
    for i in range(10):
        img_array = np.asarray(img)
        img_array_noise = add_noise(img_array)

        img_array = img_array.flatten()
        img_array = img_array.astype(np.float32)
        img_array = (img_array-128)/128

        img_array_noise = img_array_noise.flatten()
        img_array_noise = img_array_noise.astype(np.float32)
        img_array_noise = (img_array_noise-128)/128

        x.append(img_array_noise)
        y.append(img_array)

x = np.array(x)
y = np.array(y)
```

```
print(x.shape)
print(y.shape)
```

Output

```
https://upload.wikimedia.org/wikipedia/commons/9/92/Brookings.jpg
```

```
https://upload.wikimedia.org/wikipedia/commons/f/ff/WashU_Graham_Chapel.JPG
```

```
https://upload.wikimedia.org/wikipedia/commons/9/9e/SeigleHall.jpg
```

```
https://upload.wikimedia.org/wikipedia/commons/a/aa/WUSTLKnight.jpg
```

```
https://upload.wikimedia.org/wikipedia/commons/3/32/WashUABhall.jpg
```

```
https://upload.wikimedia.org/wikipedia/commons/c/c0/Brown_Hall.jpg
```

```
https://upload.wikimedia.org/wikipedia/commons/f/f4/South40.jpg
```

```
(70, 49152)
(70, 49152)
```

We now train the autoencoder neural network to transform the noisy images into clean images.

Code

```
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
from io import BytesIO
from sklearn import metrics
import numpy as np
import pandas as pd
import tensorflow as tf
from IPython.display import display, HTML

# Fit regression DNN model.
print("Creating/Training neural network")
model = Sequential()
model.add(Dense(100, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(x.shape[1])) # Multiple output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=1,epochs=20)

print("Neural network trained")
```

Output

```
Creating/Training neural network
Train on 70 samples
Epoch 1/20
70/70 [==============================] - 0s 5ms/sample - loss: 0.2747
Epoch 2/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.2006
Epoch 3/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.1687
Epoch 4/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.1522
Epoch 5/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.1279
Epoch 6/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.1066
Epoch 7/20

...

Epoch 19/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.0026
Epoch 20/20
70/70 [==============================] - 0s 3ms/sample - loss: 0.0033
Neural network trained
```

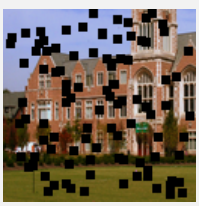
Code

```
for z in range(10):
    print("*** Trial {}".format(z+1))

    # Choose random image
    i = np.random.randint(len(loaded_images))
    img = loaded_images[i]
    img_array = np.asarray(img)
    cols, rows = img.size

    # Add noise
    img_array_noise = add_noise(img_array)

    #Display noisy image
    img2 = img_array_noise.astype(np.uint8)
    img2 = Image.fromarray(img2, 'RGB')
    print("With noise:")
    display(img2)

    # Present noisy image to auto encoder
    img_array_noise = img_array_noise.flatten()
    img_array_noise = img_array_noise.astype(np.float32)
    img_array_noise = (img_array_noise-128)/128
```

```
    img_array_noise = np.array([img_array_noise])
    pred = model.predict(img_array_noise)[0]

    # Display neural result
    img_array2 = pred.reshape(rows,cols,3)
    img_array2 = (img_array2*128)+128
    img_array2 = img_array2.astype(np.uint8)
    img2 = Image.fromarray(img_array2, 'RGB')
    print("After auto encode noise removal")
    display(img2)
```

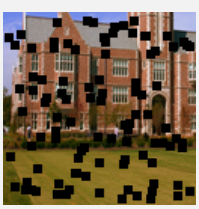
Output

```
*** Trial 1
With noise:
```

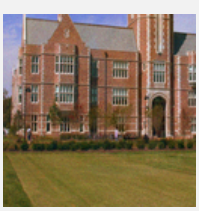


```
After auto encode noise removal
```

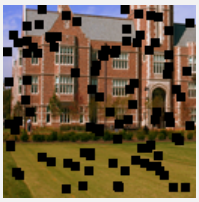
```
*** Trial 2
With noise:
```



```
After auto encode noise removal
```



```
*** Trial 3
With noise:
```



```
After auto encode noise removal
```

```
*** Trial 4
With noise:
```

```
After auto encode noise removal
```

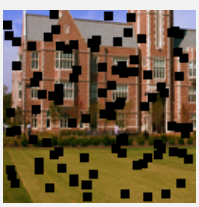
```
*** Trial 5
With noise:
```

```
After auto encode noise removal
```

*** Trial 6
With noise:

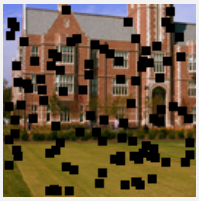
After auto encode noise removal

*** Trial 7
With noise:



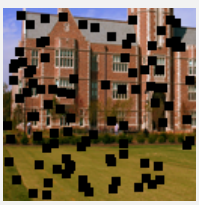
After auto encode noise removal

*** Trial 8
With noise:



After auto encode noise removal

*** Trial 9
With noise:



After auto encode noise removal

*** Trial 10
With noise:

After auto encode noise removal

14.3 Part 14.3: Anomaly Detection in Keras

Anomaly detection is an unsupervised training technique that analyzes the degree to which incoming data is different than data that the data you used to train the neural network. Traditionally, cybersecurity experts have used anomaly detection to ensure network security. However, you can use anomoly in data science to detect input that you have not trained your neural network for.

There are several data sets that are commonly used to demonstrate anomaly detection. In this part, we will look at the KDD-99 dataset.

- Stratosphere IPS Dataset
- The ADFA Intrusion Detection Datasets (2013) - for HIDS
- ITOC CDX (2009)
- KDD-99 Dataset

14.3.1 Read in KDD99 Data Set

Although KDD99 dataset is over 20 years old, it is still widely used to demonstrate Intrusion Detection Systems (IDS) and Anomaly detection. KDD99 is the data set used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Fifth International Conference on Knowledge Discovery and Data Mining. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between "bad" connections, called intrusions or attacks, and "good" normal connections. This database contains a standard set of data to be audited, including a wide variety of intrusions simulated in a military network environment.

The following code reads the KDD99 CSV dataset into a Pandas data frame. The standard format of KDD99 does not include column names. Because of that, the program adds them.

Code

```
import pandas as pd
from tensorflow.keras.utils import get_file

pd.set_option('display.max_columns', 6)
pd.set_option('display.max_rows', 5)

try:
    path = get_file('kddcup.data_10_percent.gz', origin=\
    'http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_percent.gz')
except:
    print('Error downloading')
    raise

print(path)

# This file is a CSV, just no CSV extension or headers
# Download from: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
df = pd.read_csv(path, header=None)

print("Read {} rows.".format(len(df)))
# df = df.sample(frac=0.1, replace=False) # Uncomment this line to
# sample only 10% of the dataset
```

```
df.dropna(inplace=True,axis=1)
# For now, just drop NA's (rows with missing values)

# The CSV file has no column heads, so add them
df.columns = [
    'duration',
    'protocol_type',
    'service',
    'flag',
    'src_bytes',
    'dst_bytes',
    'land',
    'wrong_fragment',
    'urgent',
    'hot',
    'num_failed_logins',
    'logged_in',
    'num_compromised',
    'root_shell',
    'su_attempted',
    'num_root',
    'num_file_creations',
    'num_shells',
    'num_access_files',
    'num_outbound_cmds',
    'is_host_login',
    'is_guest_login',
    'count',
    'srv_count',
    'serror_rate',
    'srv_serror_rate',
    'rerror_rate',
    'srv_rerror_rate',
    'same_srv_rate',
    'diff_srv_rate',
    'srv_diff_host_rate',
    'dst_host_count',
    'dst_host_srv_count',
    'dst_host_same_srv_rate',
    'dst_host_diff_srv_rate',
    'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate',
    'dst_host_serror_rate',
    'dst_host_srv_serror_rate',
    'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate',
    'outcome'
]
```

```
# display 5 rows
pd.set_option('display.max_columns', 5)
pd.set_option('display.max_rows', 5)
df
```

Output

	duration	protocol_type	...	dst_host_srv_rerror_rate	outcome
0	0	tcp	...	0.0	normal.
1	0	tcp	...	0.0	normal.
...	...	...	...	...	...
494019	0	tcp	...	0.0	normal.
494020	0	tcp	...	0.0	normal.

```
C:\Users\jeffh\.keras\datasets\kddcup.data_10_percent.gz
Read 494021 rows.
```

The KDD99 dataset contains many columns that define the network state over time intervals during which a cyber attack might have taken place. The column labeled "outcome" specifies either "normal," indicating no attack, or the type of attack performed. The following code displays the counts for each type of attack, as well as "normal".

Code

```
df.groupby('outcome')['outcome'].count()
```

Output

```
outcome
back.              2203
buffer_overflow.     30
                   ...
warezclient.       1020
warezmaster.         20
Name: outcome, Length: 23, dtype: int64
```

14.3.2 Preprocessing

Before we can feed the KDD99 data into the neural network we must perform some preprocessing. We provide the following two functions to assist with preprocessing. The first function converts numeric columns into Z-Scores. The second function replaces categorical values with dummy variables.

Code

```
# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
```

```
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd

# Encode text values to dummy variables(i.e. [1,0,0],[0,1,0],[0,0,1]
# for red,green,blue)
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = f"{name}-{x}"
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)
```

We now use these functions to preprocess each of the columns. Once the program preprocesses the data we display the results. This code converts all numeric columns to Z-Scores and all textual columns to dummy variables.

Code

```
# Now encode the feature vector

pd.set_option('display.max_columns', 6)
pd.set_option('display.max_rows', 5)

encode_numeric_zscore(df, 'duration')
encode_text_dummy(df, 'protocol_type')
encode_text_dummy(df, 'service')
encode_text_dummy(df, 'flag')
encode_numeric_zscore(df, 'src_bytes')
encode_numeric_zscore(df, 'dst_bytes')
encode_text_dummy(df, 'land')
encode_numeric_zscore(df, 'wrong_fragment')
encode_numeric_zscore(df, 'urgent')
encode_numeric_zscore(df, 'hot')
encode_numeric_zscore(df, 'num_failed_logins')
encode_text_dummy(df, 'logged_in')
encode_numeric_zscore(df, 'num_compromised')
encode_numeric_zscore(df, 'root_shell')
encode_numeric_zscore(df, 'su_attempted')
encode_numeric_zscore(df, 'num_root')
encode_numeric_zscore(df, 'num_file_creations')
encode_numeric_zscore(df, 'num_shells')
encode_numeric_zscore(df, 'num_access_files')
encode_numeric_zscore(df, 'num_outbound_cmds')
encode_text_dummy(df, 'is_host_login')
```

```
encode_text_dummy(df, 'is_guest_login')
encode_numeric_zscore(df, 'count')
encode_numeric_zscore(df, 'srv_count')
encode_numeric_zscore(df, 'serror_rate')
encode_numeric_zscore(df, 'srv_serror_rate')
encode_numeric_zscore(df, 'rerror_rate')
encode_numeric_zscore(df, 'srv_rerror_rate')
encode_numeric_zscore(df, 'same_srv_rate')
encode_numeric_zscore(df, 'diff_srv_rate')
encode_numeric_zscore(df, 'srv_diff_host_rate')
encode_numeric_zscore(df, 'dst_host_count')
encode_numeric_zscore(df, 'dst_host_srv_count')
encode_numeric_zscore(df, 'dst_host_same_srv_rate')
encode_numeric_zscore(df, 'dst_host_diff_srv_rate')
encode_numeric_zscore(df, 'dst_host_same_src_port_rate')
encode_numeric_zscore(df, 'dst_host_srv_diff_host_rate')
encode_numeric_zscore(df, 'dst_host_serror_rate')
encode_numeric_zscore(df, 'dst_host_srv_serror_rate')
encode_numeric_zscore(df, 'dst_host_rerror_rate')
encode_numeric_zscore(df, 'dst_host_srv_rerror_rate')

# display 5 rows

df.dropna(inplace=True,axis=1)
df[0:5]
```

Output

	duration	src_bytes	dst_bytes	...	is_host_login-0	is_guest_login-0	is_guest_login-1
0	-0.067792	-0.002879	0.138664	...	1	1	0
1	-0.067792	-0.002820	-0.011578	...	1	1	0
2	-0.067792	-0.002824	0.014179	...	1	1	0
3	-0.067792	-0.002840	0.014179	...	1	1	0
4	-0.067792	-0.002842	0.035214	...	1	1	0

To perform anomaly detection, we divide the data into two groups "normal" and the various attacks. The following code divides the data into two dataframes and displays each of these two groups' sizes.

Code

```
normal_mask = df['outcome']=='normal.'
attack_mask = df['outcome']!='normal.'

df.drop('outcome',axis=1,inplace=True)

df_normal = df[normal_mask]
df_attack = df[attack_mask]
```

```
print(f"Normal count: {len(df_normal)}")
print(f"Attack count: {len(df_attack)}")
```

Output

```
Normal count: 97278
Attack count: 396743
```

Next, we convert these two dataframes into Numpy arrays. Keras requires this format for data.

Code

```
# This is the numeric feature vector, as it goes to the neural net
x_normal = df_normal.values
x_attack = df_attack.values
```

14.3.3 Training the Autoencoder

It is important to note that we are not using the outcome column as a label to predict. This anomaly detection is unsupervised; there is no target (y) value to predict. We will train an autoencoder on the normal data and see how well it can detect that the data not flagged as "normal" represents an anomaly.

Next, we split the normal data into a 25% test set and a 75% train set. The program will use the test data to facilitate early stopping.

Code

```
from sklearn.model_selection import train_test_split

x_normal_train, x_normal_test = train_test_split(
    x_normal, test_size=0.25, random_state=42)
```

We display the size of the train and test sets.

Code

```
print(f"Normal train count: {len(x_normal_train)}")
print(f"Normal test count: {len(x_normal_test)}")
```

Output

```
Normal train count: 72958
Normal test count: 24320
```

We are now ready to train the autoencoder on the normal data. The autoencoder will learn to compress the data to a vector of just three numbers. The autoencoder should be able to also decompress with reasonable accuracy. As is typical for autoencoders, we are merely training the neural network to produce the same output values as were fed to the input layer.

Code

```
from sklearn import metrics
import numpy as np
import pandas as pd
from IPython.display import display, HTML
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(25, input_dim=x_normal.shape[1], activation='relu'))
model.add(Dense(3, activation='relu')) # size to compress to
model.add(Dense(25, activation='relu'))
model.add(Dense(x_normal.shape[1])) # Multiple output neurons
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x_normal_train,x_normal_train,verbose=1,epochs=100)
```

Output

```
Train on 72958 samples
Epoch 1/100
72958/72958 [==============================] - ETA: 0s - loss: 0.317 -
5s 72us/sample - loss: 0.3163
Epoch 2/100
72958/72958 [==============================] - 5s 68us/sample - loss:
0.2562
Epoch 3/100
72958/72958 [==============================] - 5s 70us/sample - loss:
0.2420
Epoch 4/100
72958/72958 [==============================] - 5s 70us/sample - loss:
0.2247
Epoch 5/100
72958/72958 [==============================] - 5s 67us/sample - loss:

...

0.0861
Epoch 100/100
72958/72958 [==============================] - 4s 53us/sample - loss:
0.0860
<tensorflow.python.keras.callbacks.History at 0x22d8990ec48>
```

14.3.4 Detecting an Anomaly

We are now ready to see if the abnormal data registers as an anomaly. The first two scores show the in-sample and out of sample RMSE errors. Both of these two scores are relatively low at around 0.33 because they resulted from normal data. The much higher 0.76 error occurred from the abnormal data. The autoencoder is not as capable of encoding data that represents an attack. This higher error indicates an anomaly.

Code

```
pred = model.predict(x_normal_test)
score1 = np.sqrt(metrics.mean_squared_error(pred,x_normal_test))
pred = model.predict(x_normal)
score2 = np.sqrt(metrics.mean_squared_error(pred,x_normal))
pred = model.predict(x_attack)
score3 = np.sqrt(metrics.mean_squared_error(pred,x_attack))
print(f"Out of Sample Normal Score (RMSE): {score1}")
print(f"Insample Normal Score (RMSE): {score2}")
print(f"Attack Underway Score (RMSE): {score3}")
```

Output

```
Out of Sample Normal Score (RMSE): 0.2767440138449814
Insample Normal Score (RMSE): 0.286465073536387
Attack Underway Score (RMSE): 0.5492403830704743
```

14.4 Part 14.4: Training an Intrusion Detection System with KDD99

The KDD-99 dataset is very famous in the security field and almost a "hello world" of Intrusion Detection Systems (IDS) in machine learning. An intrusion detection system (IDS) is program that monitors computers and network systems for malicious activity or policy violations. Any intrusion activity or violation is typically reported either to an administrator or collected centrally. IDS types range in scope from single computers to large networks. Although KDD99 dataset is over 20 years old, it is still widely used to demonstrate Intrusion Detection Systems (IDS). KDD99 is the data set used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Fifth International Conference on Knowledge Discovery and Data Mining. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between "bad" connections, called intrusions or attacks, and "good" normal connections. This database contains a standard set of data to be audited, including a wide variety of intrusions simulated in a military network environment.

14.4.1 Read in Raw KDD-99 Dataset

The following code reads the KDD99 CSV dataset into a Pandas data frame. The standard format of KDD99 does not include column names. Because of that, the program adds them.

Code

```
import pandas as pd
from tensorflow.keras.utils import get_file
```

```
try:
    path = get_file('kddcup.data_10_percent.gz', origin=
    'http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_percent.gz')
except:
    print('Error downloading')
    raise

print(path)

# This file is a CSV, just no CSV extension or headers
# Download from: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
df = pd.read_csv(path, header=None)

print("Read {} rows.".format(len(df)))
# df = df.sample(frac=0.1, replace=False) # Uncomment this line to
# sample only 10% of the dataset
df.dropna(inplace=True,axis=1) # For now, just drop NA's
# (rows with missing values)

# The CSV file has no column heads, so add them
df.columns = [
    'duration',
    'protocol_type',
    'service',
    'flag',
    'src_bytes',
    'dst_bytes',
    'land',
    'wrong_fragment',
    'urgent',
    'hot',
    'num_failed_logins',
    'logged_in',
    'num_compromised',
    'root_shell',
    'su_attempted',
    'num_root',
    'num_file_creations',
    'num_shells',
    'num_access_files',
    'num_outbound_cmds',
    'is_host_login',
    'is_guest_login',
    'count',
    'srv_count',
    'serror_rate',
    'srv_serror_rate',
```

```
    'rerror_rate',
    'srv_rerror_rate',
    'same_srv_rate',
    'diff_srv_rate',
    'srv_diff_host_rate',
    'dst_host_count',
    'dst_host_srv_count',
    'dst_host_same_srv_rate',
    'dst_host_diff_srv_rate',
    'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate',
    'dst_host_serror_rate',
    'dst_host_srv_serror_rate',
    'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate',
    'outcome'
]

pd.set_option('display.max_columns', 5)
pd.set_option('display.max_rows', 5)
# display 5 rows
display(df[0:5])
```

Output

	duration	protocol_type	...	dst_host_srv_rerror_rate	outcome
0	0	tcp	...	0.0	normal.
1	0	tcp	...	0.0	normal.
2	0	tcp	...	0.0	normal.
3	0	tcp	...	0.0	normal.
4	0	tcp	...	0.0	normal.

```
C:\Users\jeffh\.keras\datasets\kddcup.data_10_percent.gz
Read 494021 rows.
```

14.4.2 Analyzing a Dataset

Before we preprocess the KDD99 dataset let's have a look at the individual columns and distributions. You can use the following script to give a high-level overview of how a dataset appears.

Code

```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
```

```
def expand_categories(values):
    result = []
    s = values.value_counts()
    t = float(len(values))
    for v in s.index:
        result.append("{}:{}%".format(v,round(100*(s[v]/t),2)))
    return "[{}]".format(",".join(result))

def analyze(df):
    print()
    cols = df.columns.values
    total = float(len(df))

    print("{} rows".format(int(total)))
    for col in cols:
        uniques = df[col].unique()
        unique_count = len(uniques)
        if unique_count>100:
            print("** {}:{} ({}%)".format(col,unique_count,int(((unique_count)/tota
        else:
            print("** {}:{}".format(col,expand_categories(df[col])))
            expand_categories(df[col])
```

The analysis looks at how many unique values are present. For example, duration, which is a numeric value, has 2495 unique values, and there is a 0% overlap. A text/categorical value such as protocol_type only has a few unique values, and the program shows the percentages of each. Columns with a large number of unique values do not have their item counts shown to save display space.

Code

```
# Analyze KDD-99
analyze(df)
```

Output

```
494021 rows
** duration:2495 (0%)
** protocol_type:[icmp:57.41%,tcp:38.47%,udp:4.12%]
** service:[ecr_i:56.96%,private:22.45%,http:13.01%,smtp:1.97%,other:1
.46%,domain_u:1.19%,ftp_data:0.96%,eco_i:0.33%,ftp:0.16%,finger:0.14%,
urp_i:0.11%,telnet:0.1%,ntp_u:0.08%,auth:0.07%,pop_3:0.04%,time:0.03%,
csnet_ns:0.03%,remote_job:0.02%,gopher:0.02%,imap4:0.02%,domain:0.02%,
discard:0.02%,iso_tsap:0.02%,systat:0.02%,shell:0.02%,echo:0.02%,rje:0
.02%,whois:0.02%,sql_net:0.02%,printer:0.02%,courier:0.02%,nntp:0.02%,
netbios_ssn:0.02%,mtp:0.02%,sunrpc:0.02%,klogin:0.02%,vmnet:0.02%,bgp:
0.02%,uucp:0.02%,uucp_path:0.02%,ssh:0.02%,nnsp:0.02%,supdup:0.02%,hos
tnames:0.02%,login:0.02%,efs:0.02%,daytime:0.02%,link:0.02%,netbios_ns
```

```
:0.02%,ldap:0.02%,pop_2:0.02%,netbios_dgm:0.02%,http_443:0.02%,exec:0.
02%,name:0.02%,kshell:0.02%,ctf:0.02%,netstat:0.02%,Z39_50:0.02%,IRC:0
.01%,urh_i:0.0%,X11:0.0%,tim_i:0.0%,tftp_u:0.0%,red_i:0.0%,pm_dump:0.0

...

** outcome:[smurf.:56.84%,neptune.:21.7%,normal.:19.69%,back.:0.45%,sa
tan.:0.32%,ipsweep.:0.25%,portsweep.:0.21%,warezclient.:0.21%,teardrop
.:0.2%,pod.:0.05%,nmap.:0.05%,guess_passwd.:0.01%,buffer_overflow.:0.0
1%,land.:0.0%,warezmaster.:0.0%,imap.:0.0%,rootkit.:0.0%,loadmodule.:0
.0%,ftp_write.:0.0%,multihop.:0.0%,phf.:0.0%,perl.:0.0%,spy.:0.0%]
```

14.4.3 Encode the feature vector

We use the same two functions provided earlier to preprocess the data. The first encodes Z-Scores, and the second creates dummy variables from categorical columns.

Code

```
# Encode a numeric column as zscores
def encode_numeric_zscore(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd

# Encode text values to dummy variables(i.e. [1,0,0],
# [0,1,0],[0,0,1] for red,green,blue)
def encode_text_dummy(df, name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = f"{name}-{x}"
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)
```

Again, just as we did for anomaly detection, we preprocess the data set. We convert all numeric values to Z-Score, and we translate all categorical to dummy variables.

Code

```
# Now encode the feature vector

encode_numeric_zscore(df, 'duration')
encode_text_dummy(df, 'protocol_type')
encode_text_dummy(df, 'service')
```

```
encode_text_dummy(df, 'flag')
encode_numeric_zscore(df, 'src_bytes')
encode_numeric_zscore(df, 'dst_bytes')
encode_text_dummy(df, 'land')
encode_numeric_zscore(df, 'wrong_fragment')
encode_numeric_zscore(df, 'urgent')
encode_numeric_zscore(df, 'hot')
encode_numeric_zscore(df, 'num_failed_logins')
encode_text_dummy(df, 'logged_in')
encode_numeric_zscore(df, 'num_compromised')
encode_numeric_zscore(df, 'root_shell')
encode_numeric_zscore(df, 'su_attempted')
encode_numeric_zscore(df, 'num_root')
encode_numeric_zscore(df, 'num_file_creations')
encode_numeric_zscore(df, 'num_shells')
encode_numeric_zscore(df, 'num_access_files')
encode_numeric_zscore(df, 'num_outbound_cmds')
encode_text_dummy(df, 'is_host_login')
encode_text_dummy(df, 'is_guest_login')
encode_numeric_zscore(df, 'count')
encode_numeric_zscore(df, 'srv_count')
encode_numeric_zscore(df, 'serror_rate')
encode_numeric_zscore(df, 'srv_serror_rate')
encode_numeric_zscore(df, 'rerror_rate')
encode_numeric_zscore(df, 'srv_rerror_rate')
encode_numeric_zscore(df, 'same_srv_rate')
encode_numeric_zscore(df, 'diff_srv_rate')
encode_numeric_zscore(df, 'srv_diff_host_rate')
encode_numeric_zscore(df, 'dst_host_count')
encode_numeric_zscore(df, 'dst_host_srv_count')
encode_numeric_zscore(df, 'dst_host_same_srv_rate')
encode_numeric_zscore(df, 'dst_host_diff_srv_rate')
encode_numeric_zscore(df, 'dst_host_same_src_port_rate')
encode_numeric_zscore(df, 'dst_host_srv_diff_host_rate')
encode_numeric_zscore(df, 'dst_host_serror_rate')
encode_numeric_zscore(df, 'dst_host_srv_serror_rate')
encode_numeric_zscore(df, 'dst_host_rerror_rate')
encode_numeric_zscore(df, 'dst_host_srv_rerror_rate')

# display 5 rows

df.dropna(inplace=True,axis=1)
df[0:5]
# This is the numeric feature vector, as it goes to the neural net


# Convert to numpy - Classification
x_columns = df.columns.drop('outcome')
```

```
x = df[x_columns].values
dummies = pd.get_dummies(df['outcome']) # Classification
outcomes = dummies.columns
num_classes = len(outcomes)
y = dummies.values
```

We will attempt to predict what type of attack is underway. The outcome column specifies the attack type. A value of normal indicates that there is no attack underway. We display the outcomes; some attack types are much rarer than others.

Code

```
df.groupby('outcome')['outcome'].count()
```

Output

```
outcome
back.                2203
buffer_overflow.       30
                     ...
warezclient.         1020
warezmaster.           20
Name: outcome, Length: 23, dtype: int64
```

14.4.4 Train the Neural Network

We now train the neural network to classify the different KDD99 outcomes. The code provided here implements a relatively simple neural with two hidden layers. We train it with the provided KDD99 data.

Code

```
import pandas as pd
import io
import requests
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

# Create a test/train split.  25% test
# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Create neural net
```

```
model = Sequential()
model.add(Dense(10, input_dim=x.shape[1], activation='relu'))
model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
model.add(Dense(10, input_dim=x.shape[1], activation='relu'))
model.add(Dense(1, kernel_initializer='normal'))
model.add(Dense(y.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto',
                           restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=2,epochs=1000)
```

Output

```
Train on 370515 samples, validate on 123506 samples
Epoch 1/1000
370515/370515 - 24s - loss: 0.1942 - val_loss: 0.0408
Epoch 2/1000
370515/370515 - 24s - loss: 0.1164 - val_loss: 0.0293
Epoch 3/1000
370515/370515 - 24s - loss: 0.0780 - val_loss: 0.0414
Epoch 4/1000
370515/370515 - 24s - loss: 0.0524 - val_loss: 0.0251
Epoch 5/1000
370515/370515 - 24s - loss: 0.0248 - val_loss: 0.0250
Epoch 6/1000
370515/370515 - 24s - loss: 0.0224 - val_loss: 0.0220
Epoch 7/1000
370515/370515 - 24s - loss: 0.0211 - val_loss: 0.0217

...

Epoch 21/1000
Restoring model weights from the end of the best epoch.
370515/370515 - 25s - loss: 0.0141 - val_loss: 0.0149
Epoch 00021: early stopping
<tensorflow.python.keras.callbacks.History at 0x2286db53888>
```

We can now evaluate the neural network. As you can see, the neural network achieves a 99% accuracy rate.

Code

```
# Measure accuracy
pred = model.predict(x_test)
pred = np.argmax(pred,axis=1)
y_eval = np.argmax(y_test,axis=1)
```

```
score = metrics.accuracy_score(y_eval, pred)
print("Validation score: {}".format(score))
```

Output

```
Validation score: 0.998234903567438
```

14.5 Part 14.5: New Technologies

This course changes often to keep up with the rapidly evolving landscape that is deep learning. If you would like to continue to monitor this class, I suggest following me on the following:

- GitHub - I post all changes to GitHub.
- Jeff Heaton's YouTube Channel - I add new videos for this class at my channel.

Currently, four technologies are mainly on my radar for possible future inclusion in this course:

- Neural Structured Learning (NSL)
- Bert, AlBert, and Other NLP Technologies
- Explainability Frameworks

This section seeks only to provide a high-level overview of these emerging technologies. I provide links to supplemental material and code in each subsection. I describe these technologies in the following sections.

14.5.1 Neural Structured Learning (NSL)

Neural Structured Learning (NSL) provides additional training information to the neural network.[3]This training information is in the form of a graph that relates individual training cases (rows in your training set) among each other. This technique allows the neural networks to train with greater accuracy with a smaller number of labeled data. When you eventually use the neural network for scoring and prediction, once training completes, the neural network no longer uses the graph.

There are two primary sources that this graph data comes from:

- Existing Graph Relationships in Data
- Automatic Adversarial Modification of Images

Often existing graph relationships may exist in data beyond just the labels that describe what individual data items are. Consider many photo collections. There may be collections of images placed into specific albums. This album placement can form additional training information beyond the actual image data and labels.

Sometimes graph data cannot be directly obtained for a data set. Just because you do not have a graph NSL still might be an option. In such cases, you can make adversarial-like modifications to the data. You can introduce additional examples and linked them to the original images in the training set. This technique might make the final trained neural network more resilient to adversarial example attacks.

Built into TF 2.0, supports any type of ANN.

```
pip install neural_structured_learning
```

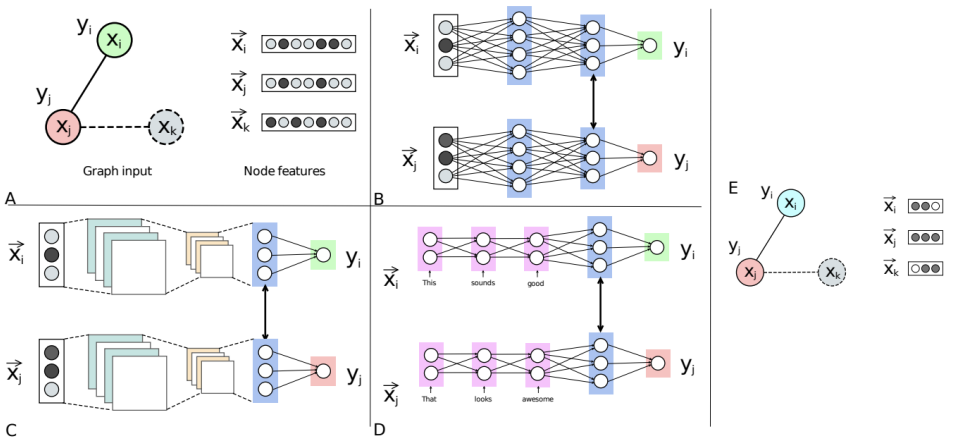


Figure 14.3: Neural Structured Learning (NSL)

Figure 14.4 is from the original NSL paper.[3]

A: An example of a graph and feature inputs. In this case, there are two labeled nodes (x_i, x_j) and one unlabeled
node (x_k), and two edges. The feature vectors, one for each node, are used as neural network inputs.

B, C, and D: Illustration of Neural Graph Machine for feed-forward, convolution, and recurrent networks respectively: the training flow ensures the neural net to make accurate node-level predictions and biases the hidden representations/embeddings of neighboring nodes to be similar. In this example, we force h_i and h_j to be similar as there is an edge connecting x_i and x_j nodes.

E: Illustration of how we can construct inputs to the neural network using the adjacency matrix. In this example, we have three nodes and two edges. The feature vector created for each node (shown on the right) has 1's at its index and indices of nodes adjacent.

Figure 14.4 shows that NSL can help when there are fewer training elements, it is from a TensorFlow presentation by Google. This video is a great starting point if you wish to do more with NSL.

The x-axis shows the amount of training data available, and the y-axis shows the accuracy attained by the model. The two graphs show NSL applied to different neural network architectures. You can use NSL with most supervised neural network architectures. As the number of training elements decreases, there is a period where NSL helps keep the accuracy higher.

14.5.2 Bert, AlBert, and Other NLP Technologies

Natural Language Processing (NLP) has seen a tremendous number of advances in the past few years. One recent technology is Bidirectional Encoder Representations from
Transformers (BERT).[6]BERT achieved "state of the art" results in the following key NLP benchmarks:

- GLUE [39]
- MultiNLI [40]
- SQuAD [**?**]

When a framework, such as BERT claims "state of the art" results, it is important to understand what is meant by that. The GLUE benchmark is composed of the following parts:

- Corpus of Linguistic Acceptability (CoLA)

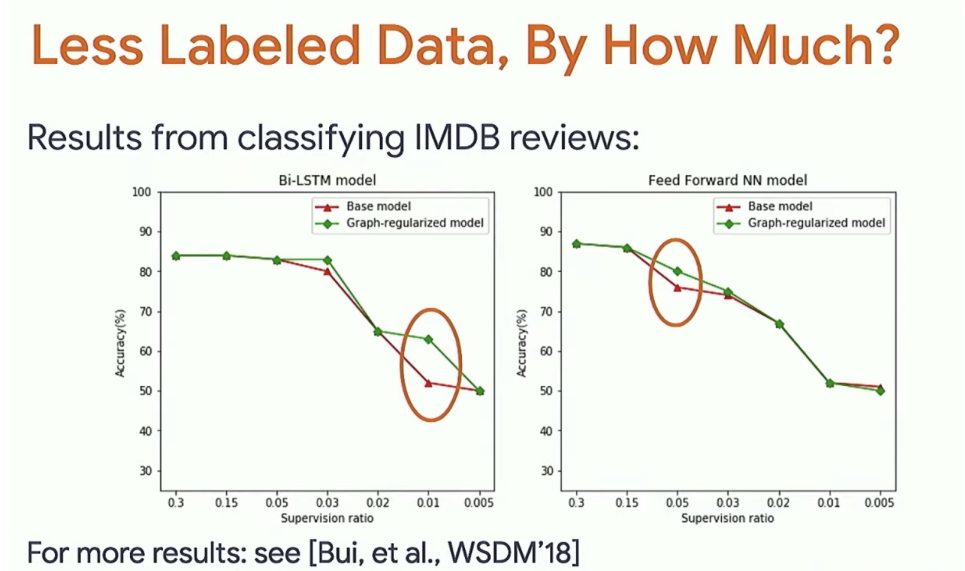


Figure 14.4: Neural Structured Learning (NSL) Results

- Stanford Sentiment Treebank (SST-2)
- Microsoft Research Paraphrase Corpus (MRPC)
- Semantic Textual Similarity Benchmark (STS-B)
- Quora Question Pairs (QQP)
- Multi-Genre Natural Language Inference Corpus matched/mismatched (MNLI-m/MNLI-mm)
- Stanford Question Answering Dataset (QNLI)
- Recognizing Textual Entailment (RTE)
- Winograd Schema Challenge (WNLI)

Single Sentence Tasks

CoLA and SST-2 are both single sentence tasks. CoLA is made up of sample sentences from English grammar textbooks where the authors demonstrated acceptable and unacceptable English grammar usage. The task in CoLA is to classify a sentence as acceptable or unacceptable. Examples include:

- Acceptable: The angrier Sue gets, the more Fred admires her.
- Unacceptable: The most you want, the least you eat.

The task SST-2 is used to analyze sentiment. Sentences are classified by their degree of positivity vs negativity. For example:

- Positive: the greatest musicians
- Negative: lend some dignity to a dumb story

Multi-Sentence Similarity Tasks

The MRPC, QQP, and STS-B tasks look at similarity and paraphrase tasks. The MRPC tests the AIs ability to paraphrase. Each row contains two sentences and a target that indicates whether each pair captures a paraphrase/semantic equivalence relationship. For example, the following two sentences are considered to be equivalent:

- He told The Sun newspaper that Mr. Hussein's daughters had British schools and hospitals in mind when they decided to ask for asylum .
- "Saddam's daughters had British schools and hospitals in mind when they decided to ask for asylum -- especially the schools," he told The Sun.

Conversely, though the following two sentences look similar, they are not considered equivalent:

- Gyorgy Heizler, head of the local disaster unit, said the coach was carrying 38 passengers.
- The head of the local disaster unit, Gyorgy Heizler, said the coach driver had failed to heed red stoplights.

The QQP tasks look at if two questions are asking the same thing. The Quora website provided this data. Examples of sentences that are considered to ask the same question:

- What are the coolest Android hacks and tricks you know?
- What are some cool hacks for Android phones?

Similarly, the following two questions are considered to be different in the QQP dataset.

- If you received a check from Donald Knuth, what did you do, and why did you get it?
- How can I contact Donald Knuth?

The STS-B dataset evaluates how similar two sentences are. If the target/label it 0, then the two sentences are completely dissimilar. A target value of 5 indicates that the two sentences are completely equivalent, as they
mean the same thing. For example:

Two sentences with a label of 0:

- A woman is dancing.
- A man is talking.

Two sentences with a label of 5:

- A plane is taking off.
- An airplane is taking off.

Inference Tasks

The tasks MNLI, QNLI, RTE, and WNLI are all inference tasks. The MNLI task provides two sentences that must be labeled neutral, contradiction, or entailment. For example, the following two sentences are a contradiction:

- At the end of Rue des Francs-Bourgeois is what many consider to be the city's most handsome residential square, the Place des Vosges, with its stone and red brick facades.
- Place des Vosges is constructed entirely of gray marble.

These two sentences are an entailment:

- I burst through a set of cabin doors, and fell to the ground-
- I burst through the doors and fell down.

These two sentences are neutral:

- It's not that the questions they asked weren't exciting or legitimate (though most did fall under the already asked and answered).
- All of the questions were interesting according to a focus group consulted on the subject.

The QNLI task poses a question and supporting sentence. The label states if the supporting information can answer the question. For example, the following two are labeled as "not_entailment":

- Question: Which missile batteries often have individual launchers several kilometers from one another?
- Answer: When MANPADS is operated by specialists, batteries may have several dozen teams deploying separately in small sections; self-propelled air defense guns may deploy in pairs.

Similarly, the following two sentences are labeled as "entailment":

- Question: What two things does Popper argue Tarski's theory involves in an evaluation of truth?
- Answer: He bases this interpretation on the fact that examples such as the one described above refer to two things: assertions and the facts to which they refer.

The RTE task is similar and looks at whether one sentence entails another. For example, the following two sentences are labeled as "entailment":

- Lin Piao, after all, was the creator of Mao's "Little Red Book" of quotations.
- Lin Piao wrote the "Little Red Book".

Similarly, the following two sentences are labeled as "not_entailment".

- Oil prices fall back as Yukos oil threat lifted
- Oil prices rise.

The WNLI task checks to see if two sentences agree to a third. For example, the following two agree:

- The foxes are getting in at night and attacking the chickens. They have gotten very bold.
- The foxes have gotten very bold.

Similarly, the following two do not agree:

- Sam pulled up a chair to the piano, but it was broken, so he had to stand instead.
- The piano was broken, so he had to stand instead.

BERT High-Level Overview **

BERT makes use of both pretraining and fine-tuning before it is ready to be used to evaluate data. It is important to understand the different roles of these two functions.

- **Pretraining** - Ideally, this is done once per language. Pretraining is the portion of BERT that most will simply obtain from the original BERT model. These can be downloaded here.
- **Fine-Tuning** - This is where additional layers are added to the base BERT models to adapt it to the intended task.

Figure 14.5 summarizes the pretraining and fine-tuning phases of BERT.

You present sentences to BERT in the method demonstrated in Figure 14.6.

14.5.3 Explainability Frameworks

Neural networks are notorious as black-box models. Such a model may make accurate predictions; however, explanations of why the black box model chose what it did can be elusive. There are two explainability libraries that I occasionally use:

- Lime
- Explain it to Me Like I'm 5 (ELI5)

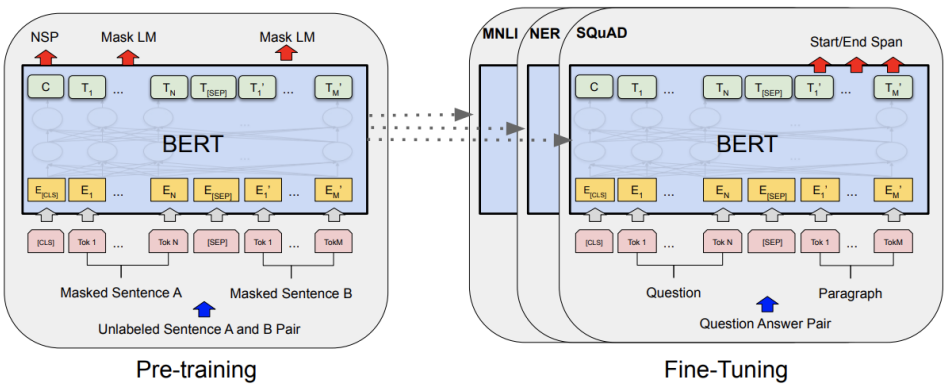


Figure 14.5: Pretraining and Fine-Tuning Phases of BERT

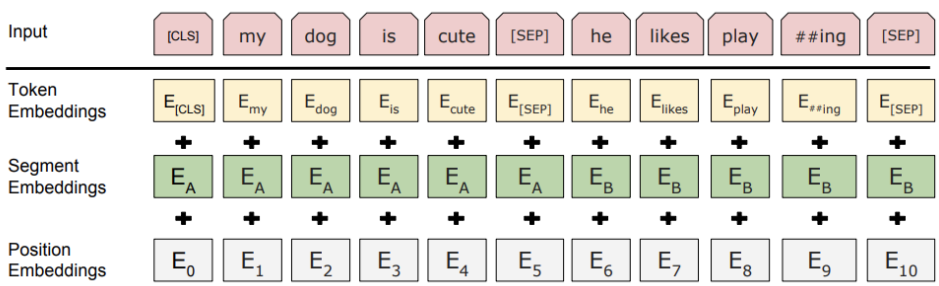


Figure 14.6: Sentences Presented to BERT

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283.

[2] Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, 5 (1983), 834–846.

[3] Bui, T. D., Ravi, S., and Ramavajjala, V. Neural graph learning: Training neural networks using graphs. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (2018), ACM, pp. 64–71.

[4] Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175* (2018).

[5] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[6] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] François, C. Deep learning with python, 2017.

[8] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics 36*, 4 (1980), 193–202.

[9] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 249–256.

[10] Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), pp. 315–323.

[11] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

[12] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems* (2014), pp. 2672–2680.

[13] Heaton, J. Encog: Library of interchangeable machine learning models for java and c#. *arXiv preprint arXiv:1506.04776* (2015).

[14] Heaton, J. Evolving continuous cellular automata for aesthetic objectives. *Genetic Programming and Evolvable Machines 20*, 1 (2019), 93–125.

[15] Heaton, J., McElwee, S., Fraley, J., and Cannady, J. Early stabilizing feature importance for tensorflow deep neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)* (2017), IEEE, pp. 4618–4624.

[16] Heaton, J. T. Automated feature engineering for deep neural networks with genetic programming.

[17] Hinton, G. E., Osindero, S., and Teh, Y.-W. A fast learning algorithm for deep belief nets. *Neural computation 18*, 7 (2006), 1527–1554.

[18] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[19] Howard, J., and Ruder, S. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).

[20] Karpathy, A. *Connecting images and natural language*. PhD thesis, Ph. D. thesis, Stanford University, 2016.

[21] Karras, T., Laine, S., and Aila, T. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), pp. 4401–4410.

[22] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. Analyzing and improving the image quality of stylegan. *arXiv preprint arXiv:1912.04958* (2019).

[23] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] LeCun, Y., Bengio, Y., et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks 3361*, 10 (1995), 1995.

[25] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436–444.

[26] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[27] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *European conference on computer vision* (2014), Springer, pp. 740–755.

[28] McCulloch, W. S., and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics 5*, 4 (1943), 115–133.

[29] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[30] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[31] Ng, A. Y. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning* (2004), p. 78.

[32] Odena, A. Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583* (2016).

[33] Olden, J. D., Joy, M. K., and Death, R. G. An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data. *Ecological modelling 178*, 3-4 (2004), 389–397.

[34] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[35] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 779–788.

[36] Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems* (2012), pp. 2951–2959.

[37] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[38] Sukhbaatar, S., Weston, J., Fergus, R., et al. End-to-end memory networks. In *Advances in neural information processing systems* (2015), pp. 2440–2448.

[39] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[40] Williams, A., Nangia, N., and Bowman, S. R. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426* (2017).