

Applications of Deep Neural Networks with Keras

Jeff Heaton

Fall 2022.0

Publisher: Heaton Research, Inc.
Applications of Deep Neural Networks
May, 2022
Author: [Jeffrey Heaton](<https://orcid.org/0000-0003-1496-4049>)
ISBN: 9798416344269
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 CC 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. LGPL



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 done their best 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 about 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.

DISCLAIMER

The author, Jeffrey Heaton, makes no warranty or representation, either expressed or implied, concerning the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will Jeffrey Heaton, his 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.

Contents

Introduction	xiii
1 Python Preliminaries	1
1.1 Part 1.1: Overview	1
1.1.1 Origins of Deep Learning	1
1.1.2 What is Deep Learning	2
1.1.3 Regression, Classification and Beyond	4
1.1.4 Why Deep Learning?	4
1.1.5 Python for Deep Learning	4
1.1.6 Check your Python Installation	5
1.1.7 Module 1 Assignment	6
1.2 Part 1.2: Introduction to Python	6
1.3 Part 1.3: Python Lists, Dictionaries, Sets, and JSON	12
1.3.1 Lists and Tuples	13
1.3.2 Sets	16
1.3.3 Maps/Dictionaries/Hash Tables	17
1.3.4 More Advanced Lists	19
1.3.5 An Introduction to JSON	22
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	26
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	32
2 Python for Machine Learning	33
2.1 Part 2.1: Introduction to Pandas	33
2.1.1 Missing Values	36

2.1.2	Dealing with Outliers	37
2.1.3	Dropping Fields	38
2.1.4	Concatenating Rows and Columns	38
2.1.5	Training and Validation	40
2.1.6	Converting a Dataframe to a Matrix	41
2.1.7	Saving a Dataframe to CSV	42
2.1.8	Saving a Dataframe to Pickle	43
2.1.9	Module 2 Assignment	44
2.2	Part 2.2: Categorical and Continuous Values	44
2.2.1	Encoding Continuous Values	45
2.2.2	Encoding Categorical Values as Dummies	46
2.2.3	Removing the First Level	49
2.2.4	Target Encoding for Categoricals	51
2.2.5	Encoding Categorical Values as Ordinal	54
2.2.6	High Cardinality Categorical	55
2.3	Part 2.3: Grouping, Sorting, and Shuffling	55
2.3.1	Shuffling a Dataset	55
2.3.2	Sorting a Data Set	56
2.3.3	Grouping a Data Set	57
2.4	Part 2.4: Apply and Map	59
2.4.1	Using Map with Dataframes	59
2.4.2	Using Apply with Dataframes	61
2.4.3	Feature Engineering with Apply and Map	62
2.5	Part 2.5: Feature Engineering	67
2.5.1	Calculated Fields	67
2.5.2	Google API Keys	68
2.5.3	Other Examples: Dealing with Addresses	68
3	Introduction to TensorFlow	73
3.1	Part 3.1: Deep Learning and Neural Network Introduction	73
3.1.1	Classification or Regression	74
3.1.2	Neurons and Layers	74
3.1.3	Types of Neurons	78
3.1.4	Input and Output Neurons	79
3.1.5	Hidden Neurons	79
3.1.6	Bias Neurons	80
3.1.7	Other Neuron Types	80
3.1.8	Why are Bias Neurons Needed?	80
3.1.9	Modern Activation Functions	82
3.1.10	Linear Activation Function	83
3.1.11	Rectified Linear Units (ReLU)	84
3.1.12	Softmax Activation Function	84
3.1.13	Step Activation Function	86
3.1.14	Sigmoid Activation Function	87

3.1.15	Hyperbolic Tangent Activation Function	87
3.1.16	Why ReLU?	88
3.1.17	Module 3 Assignment	89
3.2	Part 3.2: Introduction to Tensorflow and Keras	89
3.2.1	Why TensorFlow	90
3.2.2	Deep Learning Tools	91
3.2.3	Using TensorFlow Directly	91
3.2.4	TensorFlow Linear Algebra Examples	92
3.2.5	TensorFlow Mandelbrot Set Example	93
3.2.6	Introduction to Keras	96
3.2.7	Simple TensorFlow Regression: MPG	96
3.2.8	Introduction to Neural Network Hyperparameters	97
3.2.9	Controlling the Amount of Output	97
3.2.10	Regression Prediction	98
3.2.11	Simple TensorFlow Classification: Iris	99
3.3	Part 3.3: Saving and Loading a Keras Neural Network	105
3.4	Part 3.4: Early Stopping in Keras to Prevent Overfitting	107
3.4.1	Early Stopping with Classification	107
3.4.2	Early Stopping with Regression	110
3.5	Part 3.5: Extracting Weights and Manual Network Calculation	111
3.5.1	Weight Initialization	111
3.5.2	Manual Neural Network Calculation	113
4	Training for Tabular Data	119
4.1	Part 4.1: Encoding a Feature Vector for Keras Deep Learning	119
4.1.1	Generate X and Y for a Classification Neural Network	124
4.1.2	Generate X and Y for a Regression Neural Network	125
4.1.3	Module 4 Assignment	125
4.2	Part 4.2: Multiclass Classification with ROC and AUC	125
4.2.1	Binary Classification and ROC Charts	126
4.2.2	ROC Chart Example	129
4.2.3	Multiclass Classification Error Metrics	131
4.2.4	Calculate Classification Accuracy	133
4.2.5	Calculate Classification Log Loss	134
4.2.6	Confusion Matrix	137
4.3	Part 4.3: Keras Regression for Deep Neural Networks with RMSE	138
4.3.1	Mean Square Error	140
4.3.2	Root Mean Square Error	141
4.3.3	Lift Chart	141
4.4	Part 4.4: Training Neural Networks	142
4.4.1	Momentum Backpropagation	145
4.4.2	Batch and Online Backpropagation	145
4.4.3	Stochastic Gradient Descent	146
4.4.4	Other Techniques	146

4.4.5	ADAM Update	147
4.4.6	Methods Compared	148
4.4.7	Specifying the Update Rule in Keras	148
4.5	Part 4.5: Error Calculation from Scratch	151
4.5.1	Classification	152
5	Regularization and Dropout	155
5.1	Part 5.1: Introduction to Regularization: Ridge and Lasso	155
5.1.1	L1 and L2 Regularization	155
5.1.2	Linear Regression	157
5.1.3	L1 (Lasso) Regularization	158
5.1.4	L2 (Ridge) Regularization	160
5.1.5	ElasticNet Regularization	161
5.2	Part 5.2: Using K-Fold Cross-validation with Keras	162
5.2.1	Regression vs Classification K-Fold Cross-Validation	163
5.2.2	Out-of-Sample Regression Predictions with K-Fold Cross-Validation	163
5.2.3	Classification with Stratified K-Fold Cross-Validation	166
5.2.4	Training with both a Cross-Validation and a Holdout Set	169
5.3	Part 5.3: L1 and L2 Regularization to Decrease Overfitting	172
5.4	Part 5.4: Drop Out for Keras to Decrease Overfitting	176
5.5	Part 5.5: Benchmarking Regularization Techniques	180
5.5.1	Bootstrapping for Regression	181
5.5.2	Bootstrapping for Classification	184
5.5.3	Benchmarking	188
6	Convolutional Neural Networks (CNN) for Computer Vision	195
6.1	Part 6.1: Image Processing in Python	195
6.1.1	Creating Images from Pixels in Python	197
6.1.2	Transform Images in Python (at the pixel level)	198
6.1.3	Standardize Images	199
6.1.4	Adding Noise to an Image	201
6.1.5	Preprocessing Many Images	203
6.1.6	Module 6 Assignment	205
6.2	Part 6.2: Keras Neural Networks for Digits and Fashion MNIST	206
6.2.1	Common Computer Vision Data Sets	206
6.2.2	Convolutional Neural Networks (CNNs)	207
6.2.3	Convolution Layers	209
6.2.4	Max Pooling Layers	211
6.2.5	Regression Convolutional Neural Networks	211
6.2.6	Score Regression Image Data	217
6.2.7	Classification Neural Networks	218
6.2.8	Other Resources	222
6.3	Part 6.3: Transfer Learning for Computer Vision	222
6.3.1	Using the Structure of ResNet	227

6.4	Part 6.4: Inside Augmentation	232
6.5	Part 6.5: Recognizing Multiple Images with YOLO5	238
6.5.1	Using YOLO in Python	238
6.5.2	Installing YOLOv5	240
6.5.3	Running YOLOv5	243
6.5.4	Module 6 Assignment	249
7	Generative Adversarial Networks	251
7.1	Part 7.1: Introduction to GANS for Image and Data Generation	251
7.1.1	Face Generation with StyleGAN and Python	251
7.1.2	Generating High Rez GAN Faces with Google CoLab	253
7.1.3	Run StyleGan From Command Line	254
7.1.4	Run StyleGAN From Python Code	256
7.1.5	Examining the Latent Vector	260
7.1.6	Module 7 Assignment	262
7.2	Part 7.2: Train StyleGAN3 with your Images	263
7.2.1	What Sort of GPU do you Have?	263
7.2.2	Set Up New Environment	264
7.2.3	Find Your Files	264
7.2.4	Convert Your Images	264
7.2.5	Clean Up your Images	265
7.2.6	Perform Initial Training	265
7.2.7	Resume Training	266
7.3	Part 7.3: Exploring the StyleGAN Latent Vector	266
7.3.1	Installing Needed Software	267
7.3.2	Generate and View GANS from Seeds	269
7.3.3	Fine-tune an Image	270
7.4	Part 7.4: GANS to Enhance Old Photographs Deoldify	273
7.4.1	Install Needed Software	273
7.4.2	Initialize Torch Device	274
7.5	Part 7.5: GANs for Tabular Synthetic Data Generation	277
7.5.1	Installing Tabgan	277
7.5.2	Loading the Auto MPG Data and Training a Neural Network	277
7.5.3	Training a GAN for Auto MPG	279
7.5.4	Evaluating the GAN Results	280
8	Kaggle Data Sets	283
8.1	Part 8.1: Introduction to Kaggle	283
8.1.1	Kaggle Ranks	283
8.1.2	Typical Kaggle Competition	283
8.1.3	How Kaggle Competition Scoring	284
8.1.4	Preparing a Kaggle Submission	284
8.1.5	Select Kaggle Competitions	285
8.1.6	Module 8 Assignment	285

8.2	Part 8.2: Building Ensembles with Scikit-Learn and Keras	285
8.2.1	Evaluating Feature Importance	285
8.2.2	Classification and Input Perturbation Ranking	287
8.2.3	Regression and Input Perturbation Ranking	289
8.2.4	Biological Response with Neural Network	291
8.2.5	What Features/Columns are Important	293
8.2.6	Neural Network Ensemble	294
8.3	Part 8.3: Architecting Network: Hyperparameters	297
8.3.1	Number of Hidden Layers and Neuron Counts	298
8.3.2	Activation Functions	299
8.3.3	Advanced Activation Functions	299
8.3.4	Regularization: L1, L2, Dropout	299
8.3.5	Batch Normalization	300
8.3.6	Training Parameters	300
8.4	Part 8.4: Bayesian Hyperparameter Optimization for Keras	300
8.5	Part 8.5: Current Semester's Kaggle	307
8.5.1	Iris as a Kaggle Competition	307
8.5.2	MPG as a Kaggle Competition (Regression)	310
9	Transfer Learning	315
9.1	Part 9.1: Introduction to Keras Transfer Learning	315
9.1.1	Transfer Learning Example	315
9.1.2	Create a New Iris Network	318
9.1.3	Transferring to a Regression Network	319
9.1.4	Module 9 Assignment	322
9.2	Part 9.2: Keras Transfer Learning for Computer Vision	322
9.2.1	Transferring Computer Vision	322
9.2.2	The Kaggle Cats vs. Dogs Dataset	322
9.2.3	Looking at the Data and Augmentations	323
9.2.4	Create a Network and Transfer Weights	326
9.2.5	Fine-Tune the Model	329
9.3	Part 9.3: Transfer Learning for NLP with Keras	330
9.3.1	Benefits of Early Stopping	334
9.4	Part 9.4: Transfer Learning for Facial Points and GANs	338
9.4.1	Upload Starting and Ending Images	339
9.4.2	Install Software	340
9.4.3	Detecting Facial Features	341
9.4.4	Preprocess Images for Best StyleGAN Results	343
9.4.5	Convert Source to a GAN	346
9.4.6	Convert Target to a GAN	346
9.4.7	Build the Video	348
9.4.8	Download your Video	349
9.5	Part 9.5: Transfer Learning for Keras Style Transfer	349
9.5.1	Image Preprocessing and Postprocessing	353

9.5.2	Calculating the Style, Content, and Variation Loss	354
9.5.3	The VGG Neural Network	356
9.5.4	Generating the Style Transferred Image	358
10	Time Series in Keras	361
10.1	Part 10.1: Time Series Data Encoding	361
10.1.1	Module 10 Assignment	366
10.2	Part 10.2: Programming LSTM with Keras and TensorFlow	366
10.2.1	Understanding LSTM	366
10.2.2	Simple Keras LSTM Example	370
10.2.3	Sun Spots Example	372
10.3	Part 10.3: Text Generation with LSTM	376
10.3.1	Additional Information	376
10.3.2	Character-Level Text Generation	376
10.4	Part 10.4: Introduction to Transformers	383
10.4.1	High-Level Overview of Transformers	384
10.4.2	Transformer Hyperparameters	385
10.4.3	Inside a Transformer	385
10.5	Part 10.5: Programming Transformers with Keras	386
11	Natural Language Processing with Hugging Face	395
11.1	Part 11.1: Introduction to Hugging Face	395
11.1.1	Sentiment Analysis	395
11.1.2	Entity Tagging	397
11.1.3	Question Answering	397
11.1.4	Language Translation	398
11.1.5	Summarization	399
11.1.6	Text Generation	399
11.2	Part 11.2: Hugging Face Tokenizers	400
11.3	Part 11.3: Hugging Face Datasets	403
11.4	Part 11.4: Training Hugging Face Models	409
11.5	Part 11.5: What are Embedding Layers in Keras	412
11.5.1	Simple Embedding Layer Example	412
11.5.2	Transferring An Embedding	415
11.5.3	Training an Embedding	416
12	Reinforcement Learning	421
12.1	Part 12.1: Introduction to the OpenAI Gym	421
12.1.1	OpenAI Gym Leaderboard	421
12.1.2	Looking at Gym Environments	422
12.1.3	Render OpenAI Gym Environments from CoLab	425
12.2	Part 12.2: Introduction to Q-Learning	427
12.2.1	Introducing the Mountain Car	428
12.2.2	Programmed Car	431

12.2.3 Reinforcement Learning	433
12.2.4 Running and Observing the Agent	438
12.2.5 Inspecting the Q-Table	439
12.3 Part 12.3: Keras Q-Learning in the OpenAI Gym	440
12.3.1 DQN and the Cart-Pole Problem	441
12.3.2 Hyperparameters	443
12.3.3 Environment	444
12.3.4 Agent	447
12.3.5 Policies	448
12.3.6 Metrics and Evaluation	450
12.3.7 Replay Buffer	451
12.3.8 Data Collection	452
12.3.9 Training the agent	453
12.3.10 Visualization and Plots	455
12.3.11 Videos	456
12.4 Part 12.4: Atari Games with Keras Neural Networks	457
12.4.1 Actual Atari 2600 Specs	457
12.4.2 OpenAI Lab Atari Pong	458
12.4.3 Hyperparameters	459
12.4.4 Atari Environment	460
12.4.5 Agent	462
12.4.6 Metrics and Evaluation	464
12.4.7 Replay Buffer	465
12.4.8 Random Collection	466
12.4.9 Training the Agent	466
12.4.10 Videos	468
12.5 Part 12.5: Application of Reinforcement Learning	469
12.5.1 Create an Environment of your Own	470
12.5.2 Testing the Environment	480
12.5.3 Hyperparameters	481
12.5.4 Instantiate the Environment	482
12.5.5 Metrics and Evaluation	485
12.5.6 Data Collection	486
12.5.7 Training the agent	487
12.5.8 Visualization	489
12.5.9 Videos	489
13 Advanced/Other Topics	491
13.1 Part 13.1: Flask and Deep Learning Web Services	491
13.1.1 Flask Hello World	491
13.1.2 MPG Flask	492
13.1.3 Flask MPG Client	496
13.1.4 Images and Web Services	497
13.2 Part 13.2: Interrupting and Continuing Training	499

13.2.1	Continuing Training	507
13.3	Part 13.3: Using a Keras Deep Neural Network with a Web Application	508
13.4	Part 13.4: When to Retrain Your Neural Network	509
13.4.1	Preprocessing the Sberbank Russian Housing Market Data	512
13.4.2	KS-Statistic	513
13.4.3	Detecting Drift between Training and Testing Datasets by Training	514
13.5	Part 13.5: Tensor Processing Units (TPUs)	517
13.5.1	Preparing Data for TPUs	518
14	Other Neural Network Techniques	521
14.1	Part 14.1: What is AutoML	521
14.1.1	AutoML from your Local Computer	521
14.1.2	AutoML from Google Cloud	521
14.1.3	Using AutoKeras	523
14.2	Part 14.2: Using Denoising AutoEncoders in Keras	526
14.2.1	Multi-Output Regression	528
14.2.2	Simple Autoencoder	530
14.2.3	Autoencode (single image)	531
14.2.4	Standardize Images	533
14.2.5	Image Autoencoder (multi-image)	535
14.2.6	Adding Noise to an Image	537
14.2.7	Denoising Autoencoder	538
14.3	Part 14.3: Anomaly Detection in Keras	544
14.3.1	Read in KDD99 Data Set	544
14.3.2	Preprocessing	546
14.3.3	Training the Autoencoder	548
14.3.4	Detecting an Anomaly	549
14.4	Part 14.4: Training an Intrusion Detection System with KDD99	550
14.4.1	Read in Raw KDD-99 Dataset	550
14.4.2	Analyzing a Dataset	552
14.4.3	Encode the feature vector	553
14.4.4	Train the Neural Network	555
14.5	Part 14.5: New Technologies	557
14.5.1	New Technology Radar	557
14.5.2	Programming Language Radar	558
14.5.3	What About PyTorch?	558
14.5.4	Where to From Here?	558

Introduction

Starting in the spring semester of 2016, I began teaching the T81-558 Applications of Deep Learning course for Washington University in St. Louis. I never liked Microsoft Powerpoint for technical classes, so I placed my course material, examples, and assignments on GitHub. This material started with code and grew to include enough description that this information evolved into the book you see before you.

I license the book's text under the Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) license. Similarly, I offer the book's code under the LGPL license. Though I provide this book both as a relatively inexpensive paperback and Amazon Kindle, you can obtain the book's PDF here:

- <https://arxiv.org/abs/2009.05673>

The book's code is available at the following GitHub repository:

- https://github.com/jeffheaton/t81_558_deep_learning

If you purchased this book from me, you have my sincere thanks for supporting my ongoing projects. I sell the book as a relatively low-cost paperback and Kindle ebook for those who prefer that format or wish to support my projects. I suggest that you look at the above GitHub site, as all of the code for this book is presented there as Jupyter notebooks that are entirely Google CoLab compatible.

This book focuses on the application of deep neural networks. There is some theory; however, I do not focus on recreating neural network fundamentals that tech companies already provide in popular frameworks. The book begins with a quick review of the Python fundamentals needed to learn the subsequent chapters. With Python preliminaries covered, we start with classification and regression neural networks in Keras.

In my opinion, PyTorch, Jax, and Keras are the top three deep learning frameworks. When I first created this course, neither PyTorch nor JAX existed. I began the course based on TensorFlow and migrated to Keras the following semester. I believe TensorFlow remains a good choice for a course focusing on the application of deep learning. Some of the third-party libraries used for this course use PyTorch; as a result, you will see a blend of both technologies. StyleGAN and TabGAN both make use of PyTorch.

The technologies that this course is based on change rapidly. I update the Kindle and paperback books according to this schedule. Formal updates to this book typically occur just before each academic year's fall and spring semesters.

The source document for this book is Jupyter notebooks. I wrote a Python utility that transforms my course Jupyter notebooks into this book. It is entirely custom, and I may release it as a project someday. However, because this book is based on code and updated twice a year, you may find the occasional typo. I

try to minimize errors as much as possible, but please let me know if you see something. I use Grammarly to find textual issues, but due to the frequently updated nature of this book, I do not run it through a formal editing cycle for each release. I also double-check the code with each release to ensure CoLab, Keras, or another third-party library did not make a breaking change.

The book and course continue to be a work in progress. Many have contributed code, suggestions, fixes, and clarifications to the GitHub repository. Please submit a GitHub issue or a push request with a solution if you find an error.

Chapter 1

Python Preliminaries

1.1 Part 1.1: Overview

Deep learning is a group of exciting new technologies for neural networks.[21] 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 book introduces the reader to deep 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. Readers will 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.[9]

1.1.1 Origins of Deep Learning

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 in popularity. The current generation of deep learning began in 2006 with an improved training algorithm by Geoffrey Hinton.[12] 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 1.1.

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

- 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.



Figure 1.1: Neural Network Luminaries

- Yoshua Bengio, University of Montreal and Botler AI. Extensive research into deep learning, neural networks, and machine learning.
- Andrew Ng, Baidu 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.

1.1.2 What is Deep Learning

The focus of this book 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.

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.

- **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.

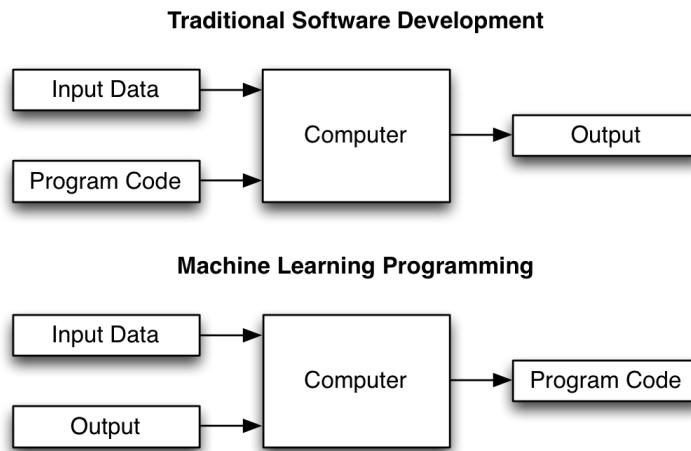


Figure 1.2: ML vs Traditional Software Development

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

- **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.
- **Tabular Data** - Several named input values allow the neural network to predict another named value that becomes the output. For example, we are using four measurements of iris flowers to predict the species. This type of data is often called tabular data.
- **Natural Language Processing (NLP)** - Deep learning transformers have revolutionized NLP, allowing text sequences to generate more text, images, or classifications.
- **Reinforcement Learning** - Reinforcement learning trains a neural network to choose ongoing actions so that the algorithm rewards the neural network for optimally completing a task.

- **Time Series** - The use of machine learning to detect patterns in time. Typical time series applications are financial applications, speech recognition, and even natural language processing (NLP).
- **Generative Models** - Neural networks can learn to produce new original synthetic data from input. We will examine StyleGAN, which learns to create new images similar to those it saw during training.

1.1.3 Regression, Classification and Beyond

Machine learning research looks at problems in broad terms of supervised and unsupervised learning. Supervised learning occurs when you know the correct outcome for each item in the training set. On the other hand, unsupervised learning utilizes training sets where no correct outcome is known. Deep learning supports both supervised and unsupervised learning; however, it also adds reinforcement and adversarial learning. Reinforcement learning teaches the neural network to carry out actions based on an environment. Adversarial learning pits two neural networks against each other to learn when the data provides no correct outcomes. Researchers continue to add new deep learning training techniques.

Machine learning practitioners usually divide supervised learning into classification and regression. Classification networks might accept financial data and classify the investment risk as risk or safe. Similarly, a regression neural network outputs a number and might take the same data and return a risk score. Additionally, neural networks can output multiple regression and classification scores simultaneously.

One of the most powerful aspects of neural networks is that the input and output of a neural network can be of many different types, such as:

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

1.1.4 Why Deep Learning?

For tabular data, neural networks often do not perform significantly better 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 tabular data tasks, deep neural networks do not necessarily add significant accuracy over other model types. However, most state-of-the-art solutions depend on deep neural networks for images, video, text, and audio data.

1.1.5 Python for Deep Learning

We will utilize the Python 3.x programming language for this book. Python has some of the widest support for deep learning as a programming language. The two most popular frameworks for deep learning in Python are:

- TensorFlow/Keras (Google)
- PyTorch (Facebook)

Overall, this book focused on the application of deep neural networks. This book focuses primarily upon Keras, with some applications in PyTorch. For many tasks, we will utilize Keras directly. We will utilize third-party libraries for higher-level tasks, such as reinforcement learning, generative adversarial neural networks, and others. These third-party libraries may internally make use of either PyTorch or Keras. I chose these libraries based on popularity and application, not whether they used PyTorch or Keras.

To successfully use this book, you must 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, with free GPU access.

If you look at this notebook on Github, near the top of the 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 - Windows/Linux
- Installing Python and TensorFlow - Mac Intel
- Installing Python and TensorFlow - Mac M1

1.1.6 Check your Python Installation

Once you've installed Python, you can utilize the following code to check your Python and library versions. If you have a GPU, you can also check to see that Keras recognize it.

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  
  
check_gpu = len(tf.config.list_physical_devices('GPU'))>0  
  
print(f"TensorFlow 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 check_gpu \
    else "NOT AVAILABLE")
```

Output

```
Tensor Flow Version: 2.8.0
Keras Version: 2.8.0
Python 3.7.13 (default, Mar 16 2022, 17:37:17)
[GCC 7.5.0]
Pandas 1.3.5
Scikit-Learn 1.0.2
GPU is available
```

1.1.7 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("HelloWorld") # Say hello
```

Output

```
Hello World
```

Strings are very versatile and allow your program to process textual information. Constant strings, 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 Python 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('HelloWorld')
```

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, note 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 3.

Code

```
for x in range(1, 3): # 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
```

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, 3):
    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
Final sum: 3
```

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, meaning 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 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 collection type in more detail.

1.3.1 Lists and Tuples

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

The primary difference that you will see syntactically is that a list is enclosed by square braces [], and a tuple 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 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.

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, 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, as 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, such as strings. Duplicate values are allowed. Tuples do not allow the program to add additional objects after definition. The programmer must specify an index for the insert function, 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 similarly a **set** is enclosed 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, 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 dictionary 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\u00a0Main" }
# 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 dictionary.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; 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 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 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": "646 555-1234"
        }
    ]
}
```

```
        "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\u00a02nd\u00a0Street",
        "city": "New\u00a0York",
        "state": "NY",
        "postalCode": "10021-3100"
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212\u00a0555-1234"
        },
        {
            "type": "office",
            "number": "646\u00a0555-4567"
        },
        {
            "type": "mobile",
            "number": "123\u00a0456-7890"
        }
    ]
}
```

```

        }
    ],
    "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 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 your code in Google CoLab, you use 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 Sonnet 18 by William Shakespeare 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://data.heatonresearch.com/images/jupyter/brookings.jpeg"

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', 'pineapple']
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', 'pineapple']
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 at <https://data.heatonresearch.com/>.

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

pd.set_option('display.max_columns', 7)
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv")
display(df[0:5])
```

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
2	18.0	8	318.0	...	70	1	plymouth satellite
3	16.0	8	304.0	...	70	1	amc rebel sst
4	17.0	8	302.0	...	70	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. If proper JSON is needed, the program should add these records to a list and call the Python JSON library's **dumps** command.

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. 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. We typically consider outliers to be a value that is several standard deviations from the mean. 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. 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 two standard deviations or more 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

You must drop fields that are of no value to the neural network. 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 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 values 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 splits 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 we train a model on 80% of the data and then validated against the remaining 20%.

Code

```

import os
import pandas as pd
import numpy as np

df = pd.read_csv(

```

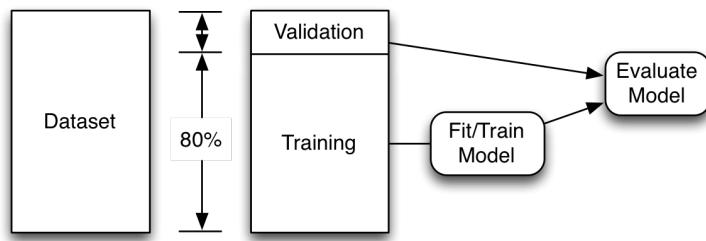


Figure 2.1: Training and Validation

```

"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 a data frame's **values** property 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)
```

Output

Done

2.1.8 Saving a Dataframe to Pickle

A variety of software programs can use text files stored as CSV. However, they take longer to generate and can sometimes lose small amounts of precision in the conversion. Generally, you will output to CSV because it is very compatible, even outside of Python. Another format is Pickle. The code below stores the Dataframe to Pickle. Pickle stores data in the exact binary representation used by Python. The benefit is that there is no loss of data going to CSV format. The disadvantage is that generally, only Python programs can read Pickle files.

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; it must be entirely numeric. It is essential to represent the data so that the neural network can train from it. Before we look at specific ways to preprocess data, it is important to consider four basic types of data, as defined by[34]. Statisticians commonly refer to as the levels of measure:

- Character Data (strings)
 - **Nominal** - Individual discrete items, no order. For example, color, zip code, and 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 could say, "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 normalize numeric inputs 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, the discount is not that good. If your friend bought lunch, 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 also need to calculate the mean(μ or \bar{x}) and the standard deviation (σ). You can calculate the mean with this equation:

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

The standard deviation is calculated as follows:

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

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 more than 3 above or below 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

The *area* column is not numeric, so you must encode it with one-hot encoding. We display the number of areas and individual values. There are just four values in the *area* categorical variable in this case.

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 *area* column. To encode these dummy variables, we would use four columns, each representing 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

We can now encode the actual column.

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]

For the new dummy/one hot encoded values to be of any use, they must be merged back into the data set.

Code

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

To encode the *area* column, we use the following code. 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 the goal is 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 Removing the First Level

The **pd.concat** function also includes a parameter named *drop_first*, which specifies whether to get k-1 dummies out of k categorical levels by removing the first level. Why would you want to remove the first level, in this case, *area_a*? This technique provides a more efficient encoding by using the ordinarily unused encoding of [0,0,0]. We encode the *area* to just three columns and map the categorical value of *a* to [0,0,0]. The following code demonstrates this technique.

Code

```
import pandas as pd

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

Output

	area_b	area_c	area_d
0	0	0	0
1	1	0	0
2	0	1	0
3	0	0	1

As you can see from the above data, the *area_a* column is missing, as it `get_dummies` replaced it by the encoding of [0,0,0]. The following code shows how to apply this technique to a dataframe.

Code

```
import pandas as pd

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

# encode the area column as dummy variables
dummies = pd.get_dummies(df['area'], drop_first=True, prefix='area')
df = pd.concat([df, dummies], axis=1)
df.drop('area', axis=1, inplace=True)

# display the encoded dataframe
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

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

Output

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

2.2.4 Target Encoding for Categoricals

Target encoding is a popular technique for Kaggle competitions. 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 of using this method.

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 corresponding 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, 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

We want to change them to a number rather than creating dummy variables for "dog" and "cat," we would like to change them to a number. We could use 0 for a cat and 1 for a 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 (y) for training. This technique will potentially lead to overfitting. The possibility of overfitting is even greater if a small number of a particular category. To prevent this from happening, we use a weighting factor. The stronger the weight, the more categories with fewer 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

2.2.5 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 and would take 21 dummy variables to encode. However, simply encoding this to dummies would lose the order information. Perhaps the most straightforward approach would be to simply number them and assign the category a single number equal to the value in the parenthesis above.

However, we might be able to do even better. A graduate student is likely more than a year so you might increase one value.

2.2.6 High Cardinality Categorical

If there were many, perhaps thousands or tens of thousands, then one-hot encoding is no longer a good choice. We call these cases high cardinality categorical. We generally encode such values with an embedding layer, which we will discuss later when introducing natural language processing (NLP).

2.3 Part 2.3: Grouping, Sorting, and Shuffling

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 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 program has not reset the row indexes' first column. Generally, this will not cause any issues and allows tracing 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 good 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.

We use the Auto MPG dataset 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

You can use the above data set with the group by 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, uses 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 you create to a data frame. Pandas supports functional programming techniques that allow you to use functions across an 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 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 that 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, that 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 the government data.

Code

```
import pandas as pd

df=pd.read_csv('https://www.irs.gov/pub/irs-soi/16zpalagi.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 across the groups and calculates the AGI estimate.

Code

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

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' ]  
  
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

	zipcode	agi_estimate
19909	63017	88689.892051

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 APIs 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 there. Google will ask for a credit card, but there will be no actual cost unless you use a massive number of lookups. YOU ARE NOT required to get a Google API key for this class; this only shows you how. If you want to get a Google API key, visit this site and obtain one for **geocode**.

You can obtain your key from this link: [Google API Keys](#).

Code

```
if 'GOOGLE_API_KEY' in os.environ:
    # If the API key is defined in an environmental variable ,
    # the use the env variable .
    GOOGLE_KEY = os.environ['GOOGLE_API_KEY']
else:
    # If you have a Google API key of your own, you can also just
    # put it here:
    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': 38.6481653, 'lng': -90.3049506}
```

They might not be overly helpful if you feed latitude and longitude into the neural network as two features. 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.

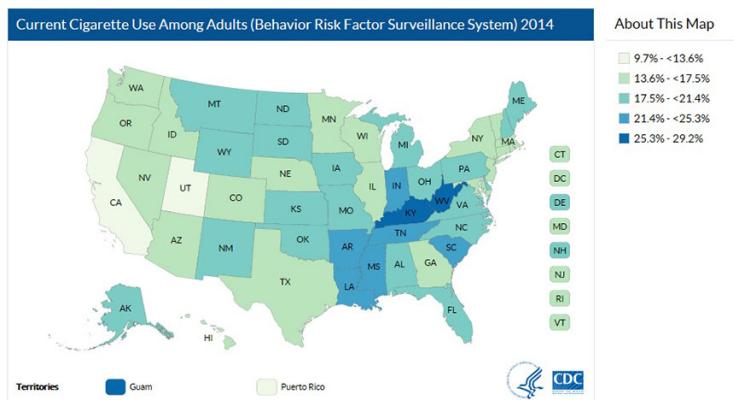


Figure 2.2: Smokers by State

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:

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

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

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:
        raise ValueError("Google API error on: {}".format(address))
    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, Ft 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: 1685.3019808607426 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' ] ,
```

```
[ "JohnsHopkinsUniversity", "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: 1354.4830895052746  
School 'Harvard', distance to wustl is: 1670.6297027161022  
School 'University of Chicago', distance to wustl is:  
418.0815972177934  
School 'Yale', distance to wustl is: 1508.217831712127  
School 'Columbia University', distance to wustl is: 1418.2264083295695  
School 'Stanford', distance to wustl is: 2780.6829398114114  
School 'MIT', distance to wustl is: 1672.4444489665696  
School 'Duke University', distance to wustl is: 1046.7970984423719  
School 'University of Pennsylvania', distance to wustl is:  
1307.19541200423  
School 'Johns Hopkins', distance to wustl is: 1184.3831076555425
```

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. This course is only concerned with the latest, most current state-of-the-art techniques for deep neural networks. I will not spend much time discussing the history 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 usually 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. Later we will learn about convolutional neural networks (CNNs), which can allow the input size to change without retraining the neural network. 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 CNN.
- **3D Matrix** - Color image input to a CNN.
- **nD Matrix** - Higher-order input to a CNN.

Before CNNs, programs either encoded images to an intermediate form or 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 matrix passes through the neural network layers.

Initially, this book will focus on 1D input to neural networks. However, later modules will focus more heavily on higher dimension input.

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 CNNs, the input has dimensions. 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 with 64x64 pixels. This configuration would result in 4,096 input neurons. This network is either 2D or 4,096D, depending on which dimensions you reference.

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.

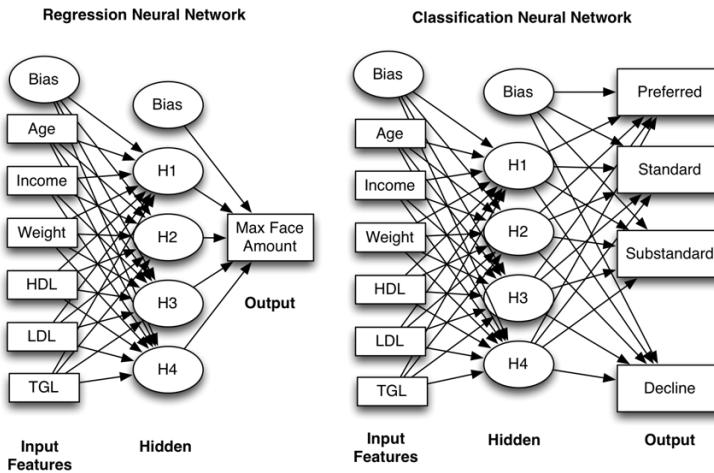


Figure 3.1: Neural Network Classification and Regression

Notice that the output of the regression neural network is numeric, and the classification output 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 neural networks exist, and programmers introduce experimental neural network structures. Consequently, it is not possible to cover every neural network architecture. However, there are some commonalities among neural network implementations. A neural network algorithm would 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.

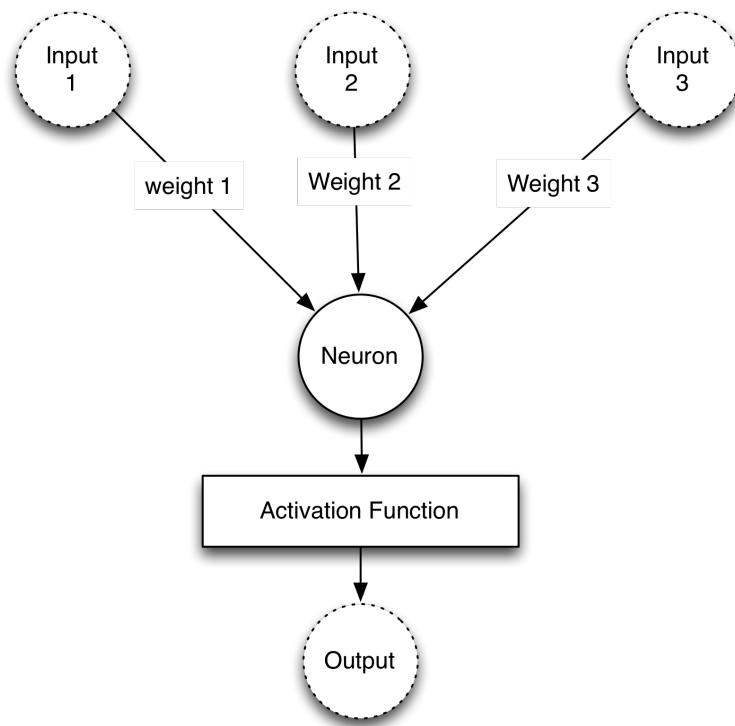


Figure 3.2: An Artificial Neuron

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 information 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:

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

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, 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:

$$0.1 \cdot 1 + 0.2 \cdot 2 + 0.3 \cdot 1 = 0.8$$

The program passes a value of 0.8 to the ϕ (phi) function, representing 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:

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 detail seen in the previous figure. We can omit the activation functions and intermediate outputs to simplify the chart, resulting in Figure 3.4.

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. 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.

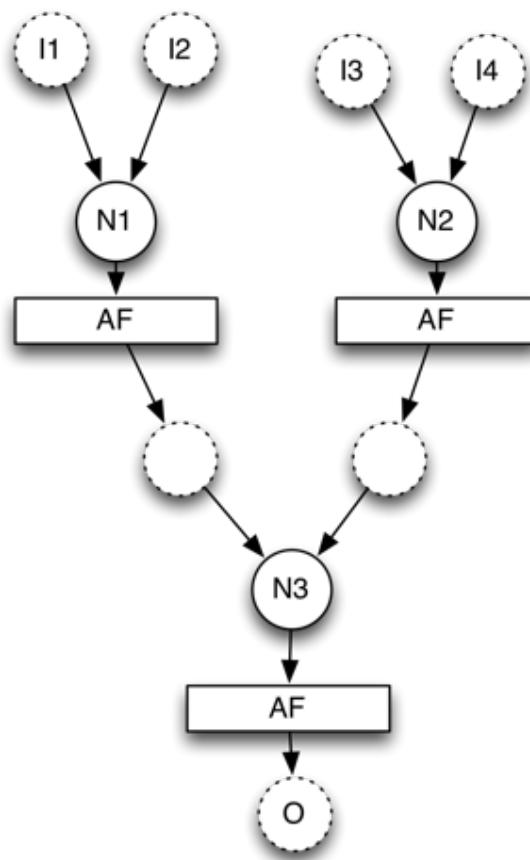


Figure 3.3: Three Neuron Neural Network

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 by 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. Without implementing 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.

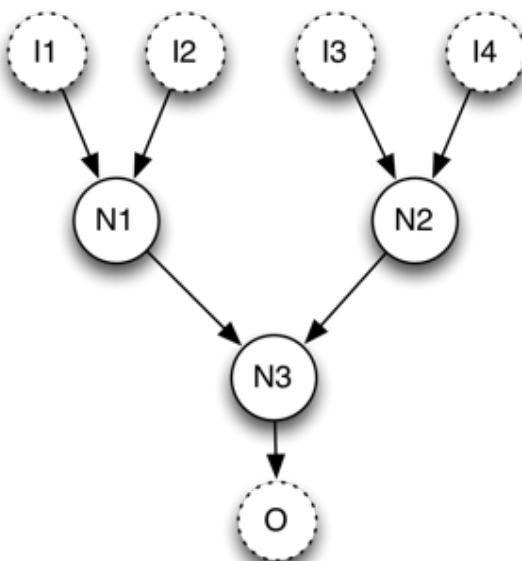


Figure 3.4: Three Neuron Neural Network

Later in this course, we will see recurrent neural networks that form inverted loops among the neurons. This type of neural network is called a feedforward neural network.

3.1.3 Types of Neurons

In the last section, we briefly introduced the idea that different types of neurons exist. Not every neural network will use every kind of neuron. It is also possible for a single neuron to fill the role of several different neuron types. Now we will explain all the neuron types described in the course.

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

- **Input Neurons** - We map each input neuron 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.
- **Bias Neurons** - Work similar to the y-intercept of a linear equation.

We place each neuron into a layer:

- **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 between the input and output layers. Each hidden layer will usually have a bias neuron.

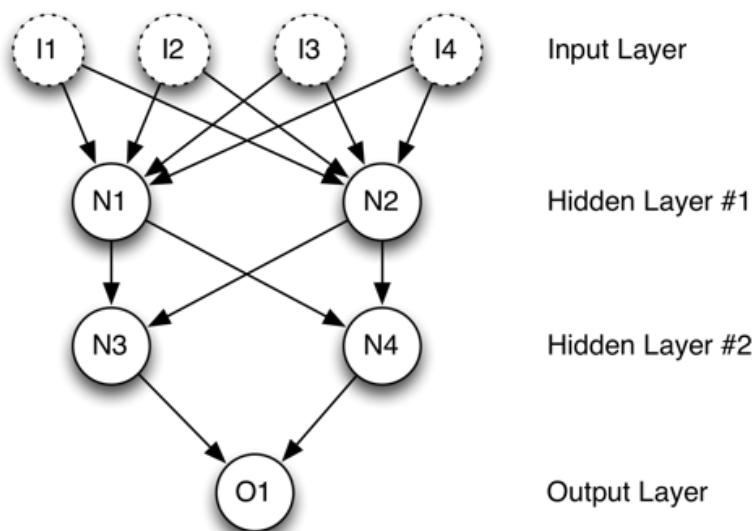


Figure 3.5: Fully Connected Neural Network Diagram

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 layers. The program normally represents the input to a neural network as an array or vector. The number of elements contained in the vector must equal 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. To be consistent, we will represent the output of a single output neuron network as a single-element vector. Likewise, neural networks will output a vector with a length equal to the number of output neurons. The output will often be a single value from a single output neuron.

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

form the output. Programmers often group hidden neurons into fully connected hidden layers. However, these hidden layers do not directly process the incoming data or the eventual output.

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.[14]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.

Training refers to the process that determines good weight values. Before the advent of deep learning, researchers feared additional layers would lengthen training time or encourage overfitting. Both concerns are true; however, increased hardware speeds and clever techniques can mitigate these concerns. Before researchers introduced deep learning techniques, we did not have an efficient way to train a deep network, which is a neural network with many 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, 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. Except for the output layer, every level 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 discussing activation functions.

3.1.7 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. You will almost always construct neural networks of weighted connections between these units.

3.1.8 Why are Bias Neurons Needed?

The activation functions from the previous section specify 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.

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

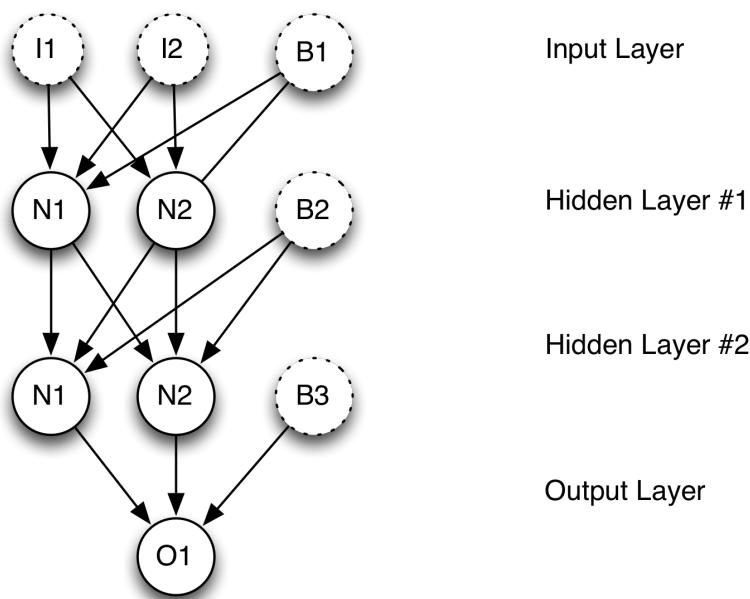


Figure 3.6: Neural Network with Bias Neurons

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 combines the weighted sum of the inputs and the sigmoid activation function. For this section, we will consider the sigmoid function because it demonstrates a bias neuron's effect.

The weights of the neuron allow you to adjust the slope or shape of the activation function. Figure 3.7 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:

$$f(x, 0.5, 0.0)$$

$$f(x, 1.0, 0.0)$$

$$f(x, 1.5, 0.0)$$

$$f(x, 2.0, 0.0)$$

We did not use bias to produce the curves, 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 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.

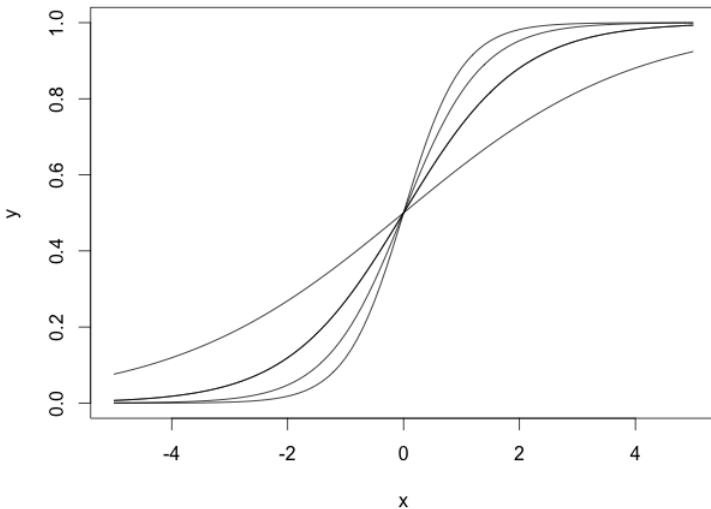


Figure 3.7: Neuron Weight Shifting

Bias does shift the sigmoid curve, which allows values other than 0.5 when x is near 0. Figure 3.8 shows the effect of using a weight of 1.0 with several different biases:

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

$$\begin{aligned} &f(x, 1.0, 1.0) \\ &f(x, 1.0, 0.5) \\ &f(x, 1.0, 1.5) \\ &f(x, 1.0, 2.0) \end{aligned}$$

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. 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.

3.1.9 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

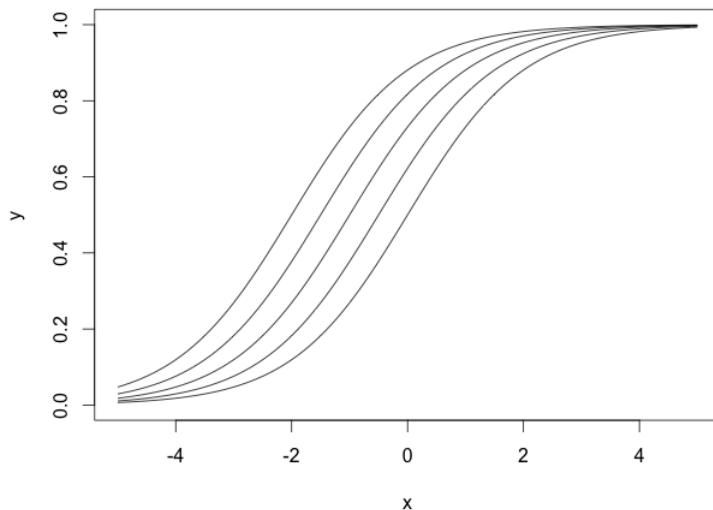


Figure 3.8: Neuron Bias Shifting

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.[8]
- **Softmax** - Used for the output of classification neural networks.
- **Linear** - Used for the output of regression neural networks (or 2-class classification).

3.1.10 Linear Activation Function

The most basic activation function is the linear function because it does not change the neuron output. 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 3.9 shows the graph for a linear activation function:

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

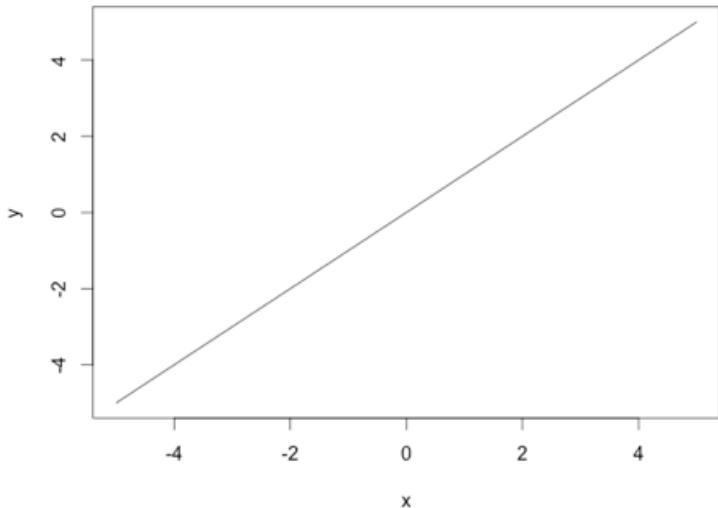


Figure 3.9: Linear Activation Function

3.1.11 Rectified Linear Units (ReLU)

Since its introduction, researchers have rapidly adopted the rectified linear unit (ReLU).[25] 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:

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

Figure 3.10 shows the graph of the ReLU activation function:

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

3.1.12 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 with the highest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the neural network's output to represent the probability that the input falls into each of the classes.

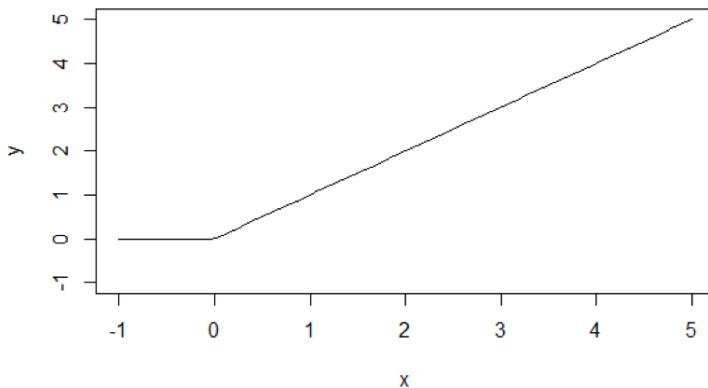


Figure 3.10: Rectified Linear Units (ReLU)

The neuron's outputs are numeric values without the softmax, 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 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 likelihood of a flower being each of the three species. To get the probability, use the softmax function in the following equation:

$$\phi_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

In the above equation, i represents the index of the output neuron (ϕ) that the program is calculating, and j represents the indexes of all neurons in the group/level. The variable x 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 website](#).

Consider a trained neural network that classifies data into three categories: 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

The above output shows 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:

$$\text{sum} = \exp(0.9) + \exp(0.2) + \exp(0.4) = 5.17283056695839$$

$$j_0 = \exp(0.9)/\text{sum} = 0.47548495534876745$$

$$j_1 = \exp(0.2)/\text{sum} = 0.2361188410001125$$

$$j_2 = \exp(0.4)/\text{sum} = 0.28839620365112$$

3.1.13 Step Activation Function

The step or threshold activation function is another simple activation function. Neural networks were initially called perceptrons. McCulloch Pitts (1943) introduced the original perceptron and used a step activation function like the following equation:[24]The step activation is 1 if $x \geq 0.5$, and 0 otherwise.

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 above the specified threshold, as seen in Figure 3.11.

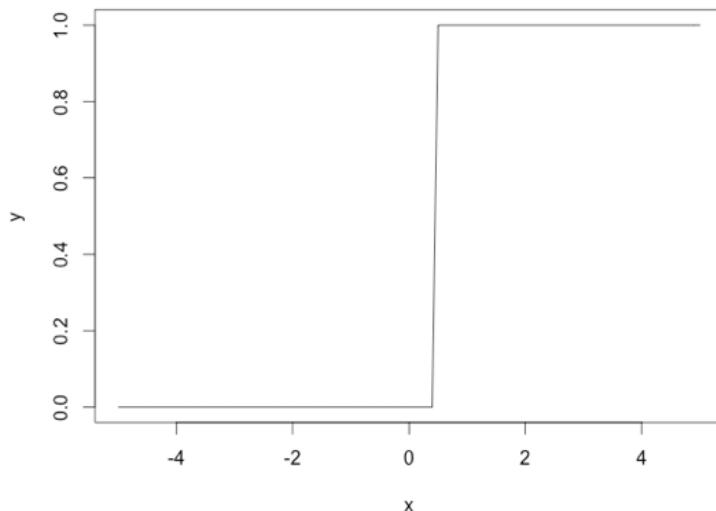


Figure 3.11: Step Activation Function

3.1.14 Sigmoid Activation Function

The sigmoid or logistic activation function is a 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 3.12:

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

3.1.15 Hyperbolic Tangent Activation Function

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

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

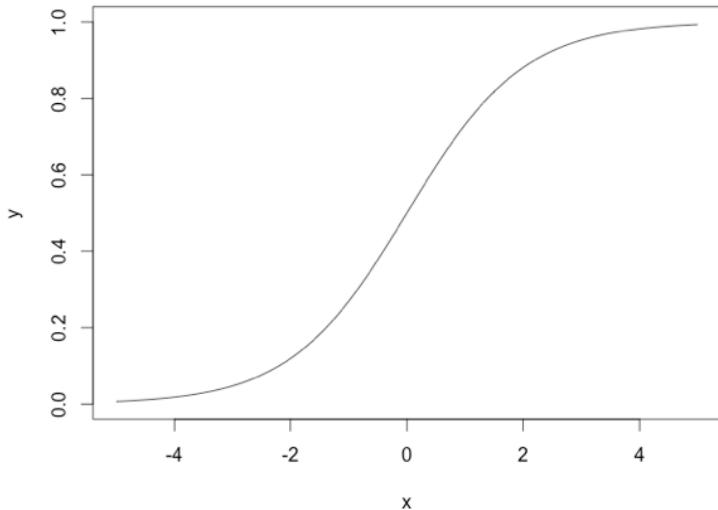


Figure 3.12: Sigmoid Activation Function

The graph of the hyperbolic tangent function has a similar shape to the sigmoid activation function, as seen in Figure 3.13.

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

3.1.16 Why ReLU?

Why is the ReLU activation function so popular? One of the critical improvements to neural networks makes deep learning work.[25] 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 use gradient descent, it is necessary to take the derivative of the activation function. The program must derive partial derivatives of each of the weights for the error function. Figure 3.14 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 3.15.

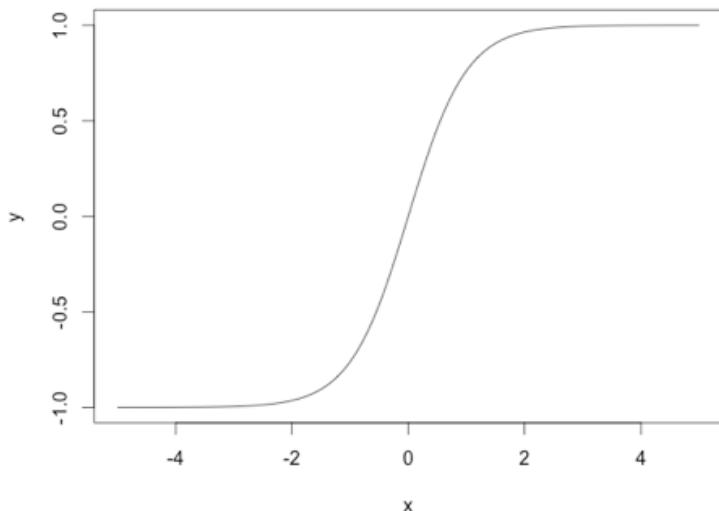


Figure 3.13: Hyperbolic Tangent Activation Function

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:

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

3.1.17 Module 3 Assignment

You can find the first assignment here: [assignment 3](#)

3.2 Part 3.2: Introduction to Tensorflow and Keras

TensorFlow[1] is an open-source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for 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](#)

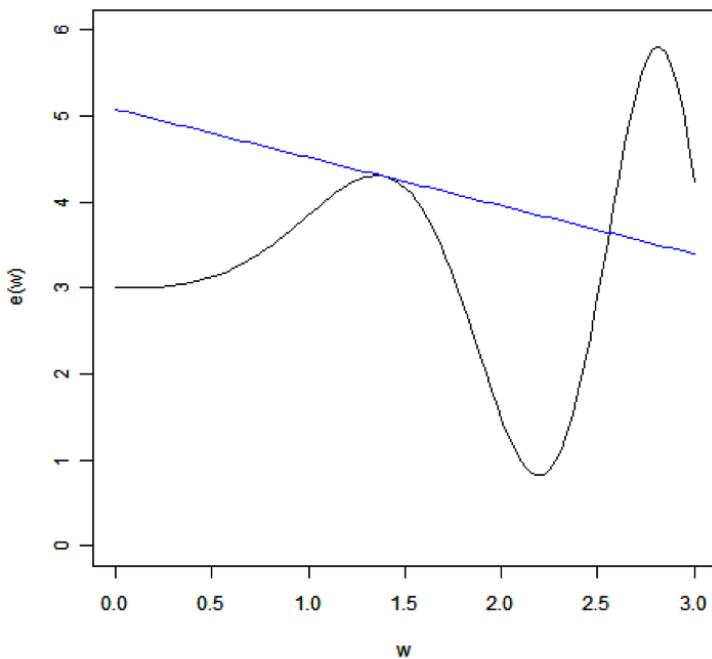


Figure 3.14: Derivative

- TensorFlow GitHub
- TensorFlow Google Groups Support
- TensorFlow Google Groups Developer Discussion
- TensorFlow FAQ

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

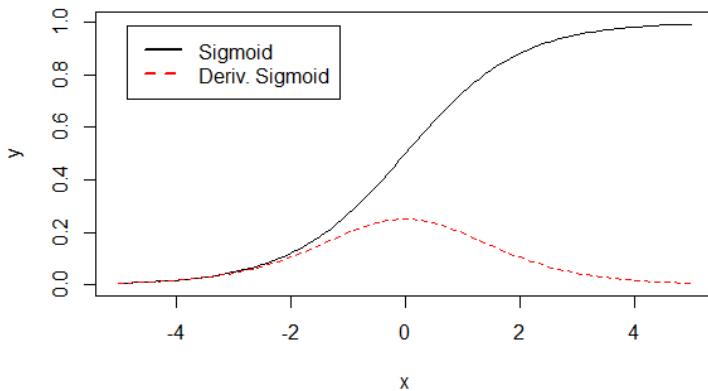


Figure 3.15: Sigmoid Derivative

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:

- **TensorFlow** - Google's deep learning API. The focus of this class, along with Keras.
- **Keras** - Acts as a higher-level to Tensorflow.
- **PyTorch** - PyTorch is an open-source machine learning library based on the Torch library, used for computer vision and natural language applications processing. Facebook's AI Research lab primarily develops PyTorch.

Other deep learning tools:

- **Deeplearning4J** - Java-based. Supports all major platforms. GPU support in Java!
- **H2O** - Java-based.

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[4], 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 directly employs TensorFlow and does not use 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 can render a Mandelbrot set. Note, I based this code on a Mandelbrot example that I originally found with TensorFlow 1.0. 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

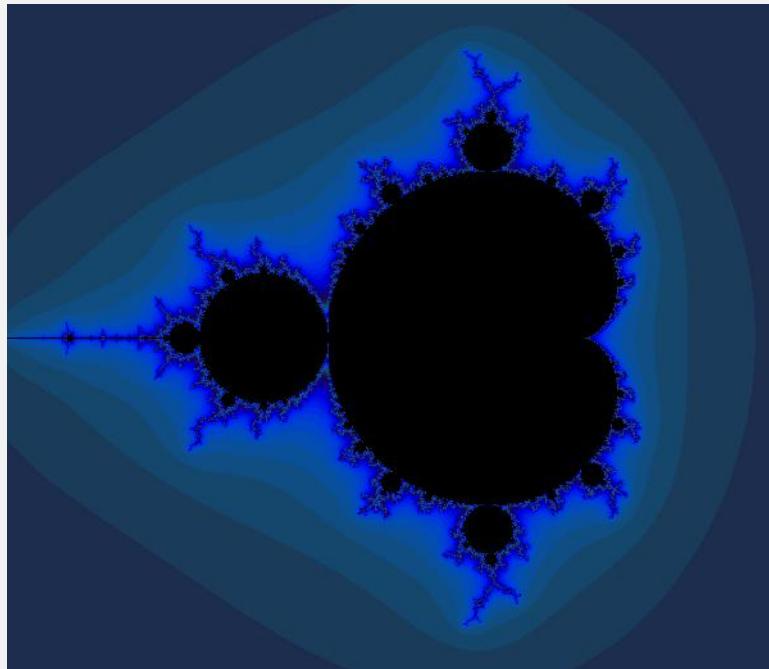
Y, X = np.mgrid[-1.3:1.3:0.005, -2:1:0.005]
Z = X+1j*Y

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



Mandlebrot rendering programs are both simple and infinitely complex at the same time. This view shows the entire Mandlebrot universe simultaneously, as a view 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 researching entirely new structures of deep neural networks, it is unlikely that you need to program TensorFlow directly.

For this class, we will 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:

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

```
...
13/13 - 0s - loss: 139.3435
Epoch 100/100
13/13 - 0s - loss: 135.2217
```

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 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 no right answer. These are hyperparameters. They are settings that can affect neural network performance, yet there are no 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 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 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)
[[22.539425]
 [27.995203]
 [25.851433]
 [25.711117]
 [23.701847]
 [31.893755]
 [35.556503]
 [34.45243 ]
 [36.27014 ]
 [31.358776]]
```

We would like to see how good these predictions are. We know the correct MPG 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): 11.552907365195134
```

The number printed above is the average number of predictions 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]}, '
          + f'predicted MPG: {pred[i]}')
```

Output

1. Car name: chevrolet chevelle malibu , MPG: 18.0 , predicted MPG: [22.539425]
2. Car name: buick skylark 320 , MPG: 15.0 , predicted MPG: [27.995203]
3. Car name: plymouth satellite , MPG: 18.0 , predicted MPG: [25.851433]
4. Car name: amc rebel sst , MPG: 16.0 , predicted MPG: [25.711117]
5. Car name: ford torino , MPG: 17.0 , predicted MPG: [23.701847]
6. Car name: ford galaxie 500 , MPG: 15.0 , predicted MPG: [31.893755]
7. Car name: chevrolet impala , MPG: 14.0 , predicted MPG: [35.556503]
8. Car name: plymouth fury iii , MPG: 14.0 , predicted MPG: [34.45243]
9. Car name: pontiac catalina , MPG: 14.0 , predicted MPG: [36.27014]
10. Car name: amc ambassador dpl , MPG: 15.0 , predicted MPG: [31.358776]

3.2.11 Simple TensorFlow Classification: Iris

Classification is how 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 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 3.16 shows how a computer and a human might respond to question number 1 on an exam.

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 to 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."



Figure 3.16: Classification Neural Network Output

We just saw 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 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

```
...  
5/5 - 0s - loss: 0.0851  
Epoch 100/100  
5/5 - 0s - loss: 0.0880
```

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 trained, 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)  
[[9.9768412e-01 2.3087766e-03 7.1474560e-06]  
 [9.9349666e-01 6.4763017e-03 2.6995105e-05]  
 [9.9618298e-01 3.7991456e-03 1.7790366e-05]  
 [9.9207532e-01 7.8882594e-03 3.6453897e-05]  
 [9.9791318e-01 2.0800228e-03 6.7602941e-06]  
 [9.9684995e-01 3.1442614e-03 5.8112000e-06]  
 [9.9547136e-01 4.5086881e-03 1.9946103e-05]  
 [9.9625921e-01 3.7288493e-03 1.2040506e-05]  
 [9.9011189e-01 9.8296851e-03 5.8434536e-05]]
```

```
[9.9447203e-01 5.5067884e-03 2.1272421e-05]]
```

If you would like to turn off 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 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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1]
```

```
2 1
1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2
2 2]
Expected: [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 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2
2 2]
```

Of course, it is straightforward to turn these indexes back into iris species. We use the species list that we created earlier.

Code

```
print(species[predict_classes[1:10]])
```

Output

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

Accuracy might be a more easily understood error metric. It is essentially a test score. For all of the iris predictions, what percent were correct? The downside is it does not consider how confident the neural network was in each prediction.

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 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

```
[[0.00065001 0.17222181 0.8271282 ]]
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

```
[[0.00065001 0.17222157 0.8271284 ]
 [0.9887937 0.01117751 0.00002886]]
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 you can reload them later. A reloaded neural network will not require retraining. Keras provides three formats for neural network saving.

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

```

Output

```

...
13/13 - 0s - loss: 50.2118 - 25ms/epoch - 2ms/step
Epoch 100/100
13/13 - 0s - loss: 49.8828 - 25ms/epoch - 2ms/step
Before save score (RMSE): 7.044431690300903

```

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 code loads the weights from the previous fit. Now we reload the network and perform another prediction. The RMSE should match the previous one exactly if we saved and reloaded the neural network correctly.

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): 7.044431690300903

```

3.4 Part 3.4: Early Stopping in Keras to Prevent Overfitting

It can be difficult to determine how many epochs to cycle through to train a neural network. Overfitting will occur if you train the neural network for too many epochs, and the neural network will not perform well on new data, despite attaining a good accuracy on the training set. Overfitting occurs when a neural network is trained to the point that it begins to memorize rather than generalize, as demonstrated in Figure 3.17.

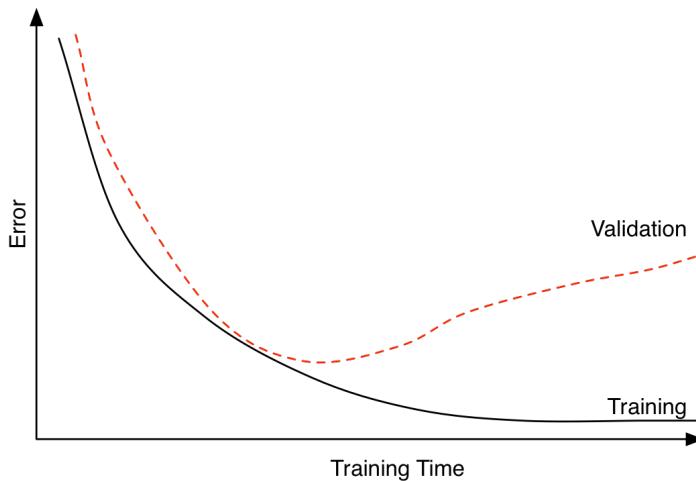


Figure 3.17: Training vs. Validation Error for Overfitting

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

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

You can construct these sets in several different ways. The following programs demonstrate some of these.

The first method is a training and validation set. We use the training data 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% of the data. The predictions for the training data will be overly optimistic, as these were the data that we used to train the neural network. Figure 3.18 demonstrates how we divide the dataset.

3.4.1 Early Stopping with Classification

We will now see an example of classification training with early stopping. We will train the neural network until the error no longer improves on the validation set.

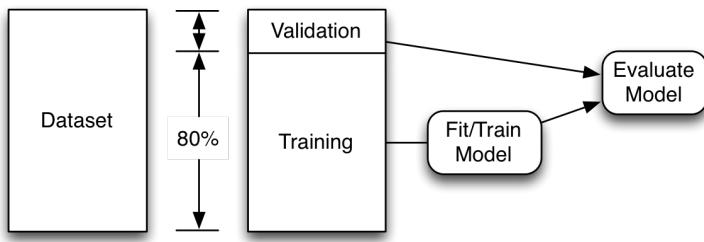


Figure 3.18: Training with a Validation Set

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
...
112/112 - 0s - loss: 0.1017 - val_loss: 0.0926
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

```

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
...
298/298 - 0s - loss: 34.0591 - val_loss: 29.3044
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
```

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 training process 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 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 practical 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. The weights can be so bad that training is impossible. If 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, considerable research has been around it. By default, Keras uses the Xavier weight initialization algorithm, introduced in 2006 by Glorot Bengio[7], 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. 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, 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 can control the range of random numbers that you will receive.

The Xavier weight initialization sets all 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 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 3.19 shows how this algorithm might initialize one layer.

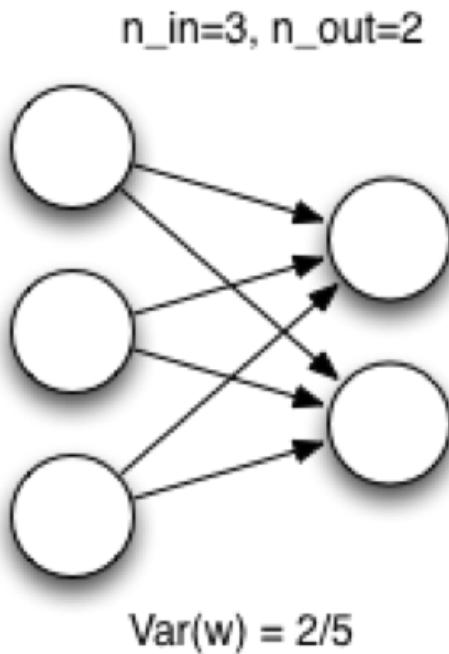


Figure 3.19: Xavier Weight Initialization

We complete this process for each layer in the neural network.

3.5.2 Manual Neural Network Calculation

This section 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 hand-code the neurons to provide an XOR function; however, we will allow Keras for simplicity to train this network for us. The neural network is small, with two inputs, two hidden neurons, and a single output. We will use 100K epochs on the ADAM optimizer. This approach is overkill, but it gets the result, and our focus here is not on tuning.

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]
])

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 fourth 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 we've trained the neural network, we can 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 data is what you would typically see in Microsoft Excel. Neural networks require numeric input. This numeric form is called a feature vector. Each input neurons receive one feature (or column) from this vector. Each row of training data typically becomes one vector. This section will see how to encode the following tabular data into a feature vector. You can see an example of tabular data below.

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

You can make the following observations 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 the column "product". This field specifies what product someone bought.
- There is an ID column. You should exclude his column because it contains no information useful for prediction.
- Many of these fields are numeric and might not require 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 the dummies now represent it.

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 hint at what the income was. For example, it might be beneficial to calculate a median income for each area or job category. This technique is something to keep in mind for the class Kaggle competition.

At this point, the Pandas data frame 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 data 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 that this is how we generate y for a classification problem. Regression would not use dummies and would 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

The program generates the x values the same way for a regression neural network. 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

The output of modern neural networks can be of many different forms. However, classically, neural network output has typically been one of the following:

- **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)

We will look at some visualizations for all three in this section.

It is important to evaluate the false positives and negatives in the results produced by a neural network. We will now look at assessing 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: 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 system will either "issue a credit card" or "decline a credit card." This classification system must decide how to respond to a new potential customer.

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 essential to understand the difference. For the previous example, issuing a credit card would be positive. A false positive occurs when a model decides to issue a credit card to someone who will not make payments as agreed. A false negative happens when a model denies a credit card to someone who would have made payments as agreed.

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 prevalent in analytics. It is essential to know how to read them. Even their name is confusing. Do not worry about their name; the receiver operating characteristic curve (ROC) comes from electrical engineering (EE).

Binary classification is common in medical testing. Often you want to diagnose if someone has a disease. This diagnosis can lead to two types of errors, known as false positives and false negatives:

- **False Positive** - Your test (neural network) indicated that the patient had the disease; however, the patient did not.
- **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.

Figure 4.1 shows you these types of errors.

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 the probability of it being positive. However, at what possibility 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

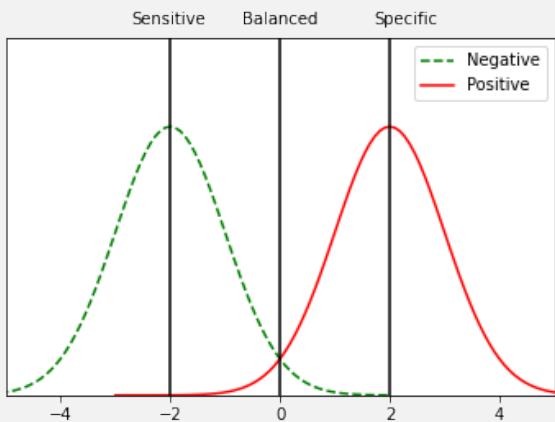
Code

```
%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



We will now train a neural network for the Wisconsin breast cancer dataset. We begin by preprocessing the data. Because we have all numeric data, we compute a z-score for each column.

Code

We can now define two functions. The first function plots a confusion matrix. The second function plots a ROC chart.

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)
```

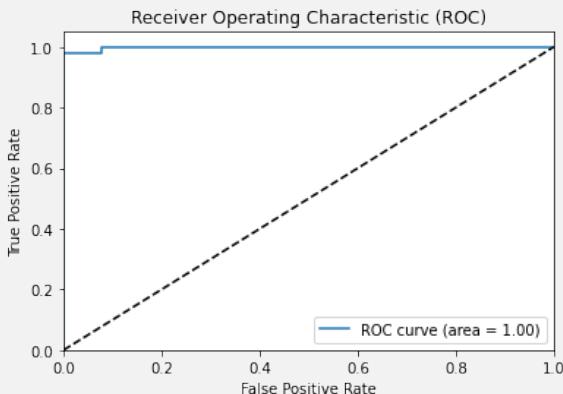
Output

```
...
14/14 - 0s - loss: 0.0458 - accuracy: 0.9836 - val_loss: 0.0486 -
val_accuracy: 0.9860 - 119ms/epoch - 8ms/step
Epoch 13/1000
Restoring model weights from the end of the best epoch: 8.
14/14 - 0s - loss: 0.0417 - accuracy: 0.9883 - val_loss: 0.0477 -
val_accuracy: 0.9860 - 124ms/epoch - 9ms/step
Epoch 13: early stopping
```

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 results, 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. We will assess the following classification neural 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

```

# 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

```

...
47/47 - 0s - loss: 0.6624 - accuracy: 0.7147 - val_loss: 0.7527 -
val_accuracy: 0.6800 - 328ms/epoch - 7ms/step
Epoch 21/1000
Restoring model weights from the end of the best epoch: 16.
47/47 - 1s - loss: 0.6558 - accuracy: 0.7160 - val_loss: 0.7653 -
val_accuracy: 0.6720 - 527ms/epoch - 11ms/step
Epoch 21: early stopping

```

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.

$$\text{accuracy} = \frac{c}{N}$$

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.        , 0.1201, 0.7286, 0.1494, 0.0018, 0.        , 0.        ],
       [0.        , 0.6962, 0.3016, 0.0001, 0.0022, 0.        , 0.        ],
       [0.        , 0.7234, 0.2708, 0.0003, 0.0053, 0.0001, 0.        ],
       [0.        , 0.3836, 0.6039, 0.0086, 0.0039, 0.        , 0.        ],
       [0.        , 0.0609, 0.6303, 0.3079, 0.001 , 0.        , 0.        ]],
      dtype=float32) As percent probability
[ 0.0001 12.0143 72.8578 14.9446 0.1823 0.0009 0.0001]
Log loss score: 0.7423401429280638
```

Log loss is calculated as follows:

$$\text{log loss} = -\frac{1}{N} \sum_{i=1}^N (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

You should use this equation only as an objective function for classifications that have two outcomes. The variable \hat{y} 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 \hat{y} , then your prediction for the second class is 1 minus \hat{y} . 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:

- $-\log(1.0) = 0$
- $-\log(0.95) = 0.02$
- $-\log(0.9) = 0.05$
- $-\log(0.8) = 0.1$
- $-\log(0.5) = 0.3$
- $-\log(0.1) = 1$
- $-\log(0.01) = 2$

- $-\log(1.0e-12) = 12$
- $-\log(0.0) = \text{negative infinity}$

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

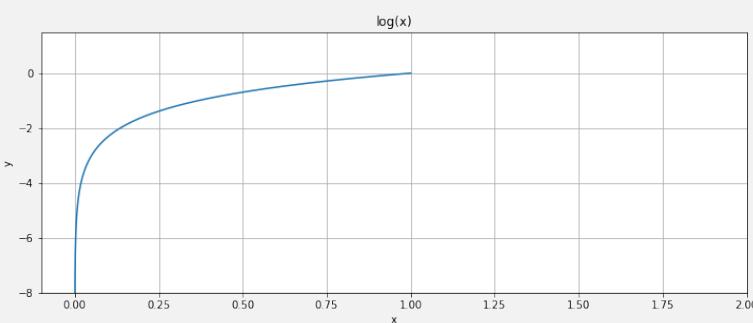
#t = arange(1e-5, 5.0, 0.00001)
#t = arange(1.0, 5.0, 0.00001) # computer scientists
t = arange(0.0, 1.0, 0.00001) # data scientists

fig = figure(1, figsize=(12, 10))

ax1 = fig.add_subplot(211)
ax1.plot(t, np.log(t))
ax1.grid(True)
ax1.set_ylimits((-8, 1.5))
ax1.set_xlim((-0.1, 2))
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('log(x)')

show()
```

Output



4.2.6 Confusion Matrix

A confusion matrix shows which predicted classes are often confused for the other classes. The vertical axis (y) represents the true labels and the horizontal axis (x) represents the predicted labels. When the true label and predicted label are the same, the highest values occur down the diagonal extending from the upper left to the lower right. The other values, outside the diagonal, represent incorrect predictions. For example, in the confusion matrix below, the value in row 2, column 1 shows how often the predicted value A occurred when it should have been B.

Code

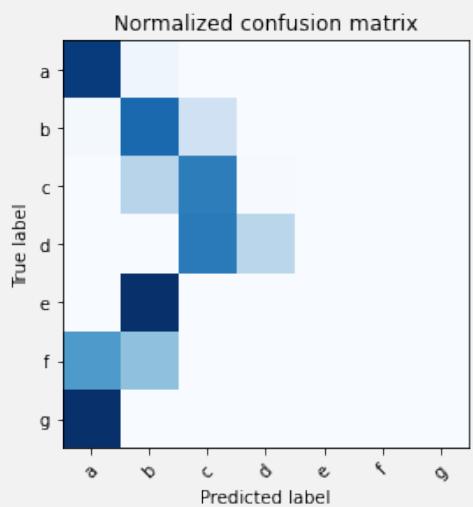
```
import numpy as np
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

# Compute confusion matrix
cm = confusion_matrix(y_compare, pred)
np.set_printoptions(precision=2)

# 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



Normalized confusion matrix

```
[[0.95  0.05  0.    0.    0.    0.    0.    ]
 [0.02  0.78  0.2   0.    0.    0.    0.    ]
 [0.    0.29  0.7   0.01  0.    0.    0.    ]
 [0.    0.    0.71  0.29  0.    0.    0.    ]
 [0.    0.    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

We evaluate regression results differently than classification. Consider the following code that trains a neural network for regression on the data set `jh-simple-dataset.csv`. We begin by preparing the data set.

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)

```

Next, we create a neural network to fit the data we just loaded.

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
...
1500/1500 - 0s - loss: 0.4081 - val_loss: 0.5540
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
```

4.3.1 Mean Square Error

The mean square error (MSE) 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. We seek to achieve low MSE values. The following equation demonstrates how to calculate MSE.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

The following code calculates the MSE on the predictions from the neural network.

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. We desire Low RMSE values. The following equation calculates RMSE.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2}$$

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

We often visualize the results of regression with a lift chart. To generate a lift chart, perform the following activities:

- Sort the data by expected output and plot these values.
- For every point on the x-axis, plot that same data point's predicted value in another color.
- 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.

You can interpret the lift chart as follows:

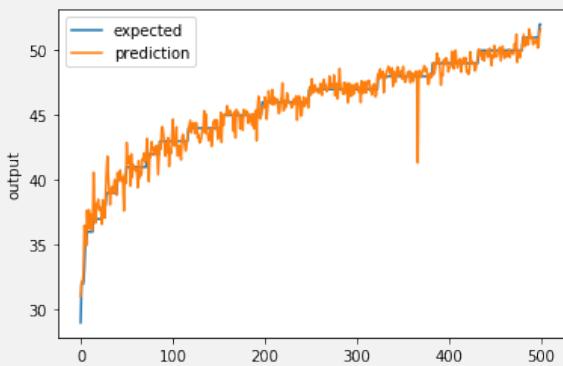
- The expected and predict lines should be close. Notice where one is above the other.
- The below chart is the most accurate for lower ages.

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()

# Plot the chart
chart_regression(pred.flatten(), y_test)
```

Output



4.4 Part 4.4: Training Neural Networks

Backpropagation[29] is one of the most common methods for training a neural network. Rumelhart, Hinton, Williams 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.

Researchers have extended classic backpropagation and modified to give rise to many different training algorithms. This section will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and end the chapter with stochastic gradient descent (SGD).

Backpropagation is the primary means of determining a neural network's weights during training. Backpropagation works by calculating a weight change amount (v_t) for every weight(θ , theta) in the neural network. This value is subtracted from every weight by the following equation:

$$\theta_t = \theta_{t-1} - v_t$$

We repeat this process for every iteration(t). The training algorithm determines how we calculate the weight change. Classic backpropagation calculates a gradient (∇ , nabla) for every weight in the neural network for the neural network's error function (J). We scale the gradient 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 a learning rate will usually converge to a reasonable solution; however, the process will be prolonged.
- 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 gradient descent type, and many texts will use these two terms interchangeably. Gradient descent refers to calculating 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 indicate 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. We can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

The gradient is the partial derivative of each weight in the neural network concerning the error function. Each weight has a gradient that is the slope of the error function. 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 does not contribute to the neural network's error.

- Negative gradient - The algorithm should increase the weight to lower error.
- Positive gradient - The algorithm should decrease the weight to 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 infinite computation resources, we would 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 shortcuts 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:

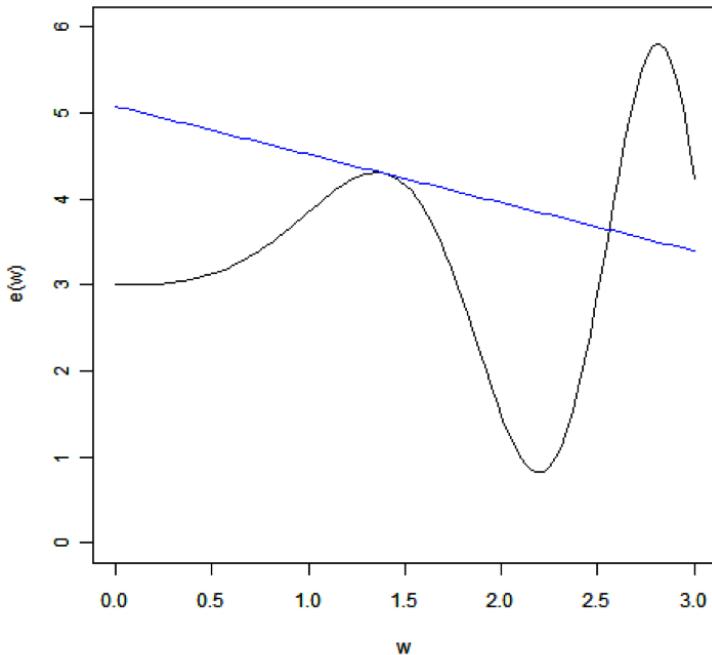


Figure 4.2: Derivative

Looking at this chart, you can easily see that the optimal weight is 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 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 weight.

Derivatives are one of the most fundamental concepts in calculus. For this book, you 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 show how to calculate it.

4.4.1 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 adds the scaled value of the previous weight change amount (v_{t-1}) to the current weight change amount (v_t).

This technique has the effect of adding additional force behind the direction a weight is moving. Figure 4.3 shows how this might allow the weight to escape local minima.

A typical value for momentum is 0.9.

4.4.2 Batch and Online Backpropagation

How often should the weights of a neural network be updated? We can calculate gradients 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 minimal batch size. Mini-batches are very popular, often in the 32-64 element range.

Because the batch size is smaller than the full training set size, it may take several batches to make it completely through the training set.

- **Step/Iteration** - The number of processed batches.
- **Epoch** - The number of times the algorithm processed the complete training set.

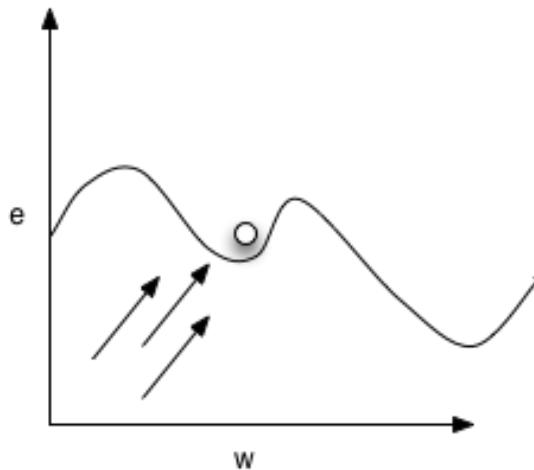


Figure 4.3: Momentum

4.4.3 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 technique 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. Each training step can be relatively fast, even with a huge training set.
- Decreases overfitting by focusing on only a portion of the training set each step.

4.4.4 Other Techniques

One problem with simple backpropagation training algorithms is that they are susceptible to learning rate and momentum. This technique 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 not to destabilize the training.
- A single learning rate/momentun is often not good enough for the entire training process. It is often helpful to automatically decrease the learning rate as the training progresses.
- All weights share a single learning rate/momentun.

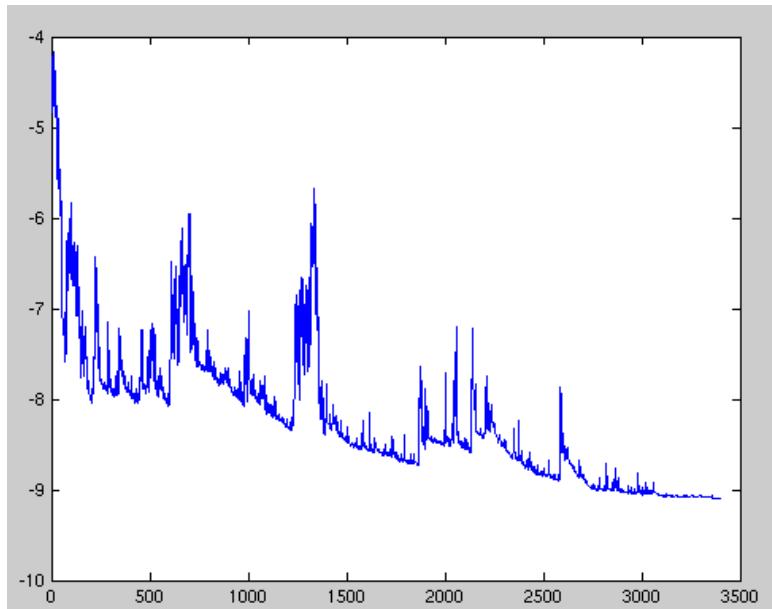


Figure 4.4: SGD Error

Other training techniques:

- **Resilient Propagation** - Use only the magnitude of the gradient and allow each neuron to learn at its rate. There is no need for learning rate/momentum; however, it 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.

4.4.5 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.[19] 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 gradients' first moment (the mean) and the second moment (the uncentered variance). However, they will be strongly biased 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 (α) 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 .

4.4.6 Methods Compared

The following image shows how each of these algorithms train. It is animated, so it is not displayed in the printed book, but can be accessed from here: <https://bit.ly/3kykkbn>.

Image credits: Alec Radford

4.4.7 Specifying the Update Rule in Keras

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' ])

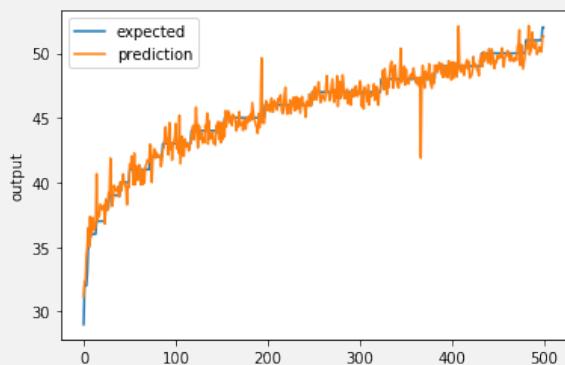
# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df[ 'age' ].values

# 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. RMSE is typically used for regression. We begin by calculating RMSE with libraries.

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
```

We can also calculate without libraries.

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

```

4.5.1 Classification

We will now look at how to calculate a logloss by hand. For this, we look at a binary prediction. The predicted is some number between 0-1 that indicates the probability true (1). The expected is always 0 or 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]))+\nnp.log(1.0-np.abs(expected[2]-predicted[2]))+\nnp.log(1.0-np.abs(expected[3]-predicted[3]))+\nnp.log(1.0-np.abs(expected[4]-predicted[4])))\\*\n*(-1/len(predicted))\n\nprint(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 examining how 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 retaking the practice exam. The programmer has memorized much of the practice exam at some point rather than learning the techniques necessary to figure out the individual questions. The programmer has now overfitted for 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.

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. A computer can overfit as well. Regularization is one of the techniques that can prevent overfitting. Several 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 standard regularization techniques that can reduce the effects of overfitting. These algorithms can either work with an objective function or as part of the backpropagation algorithm. The regularization algorithm is attached to the training algorithm by adding an objective in both cases.

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. You can add this penalty calculation to the calculated gradients for gradient-descent-based algorithms, such as backpropagation. The penalty is negatively combined with the objective score for objective-function-based training, such as simulated annealing.

We will 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)
```

We will use the data just loaded for several examples. The first examples in this part use several forms of linear regression. For linear regression, it is helpful to examine the model's coefficients. The following function is utilized to display these coefficients.

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

Before jumping into L1/L2 regularization, we begin with linear regression. Researchers first introduced the L1/L2 form of regularization for linear regression. We can also make use of L1/L2 for neural networks. To fully understand L1/L2 we will begin with how we can use them 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

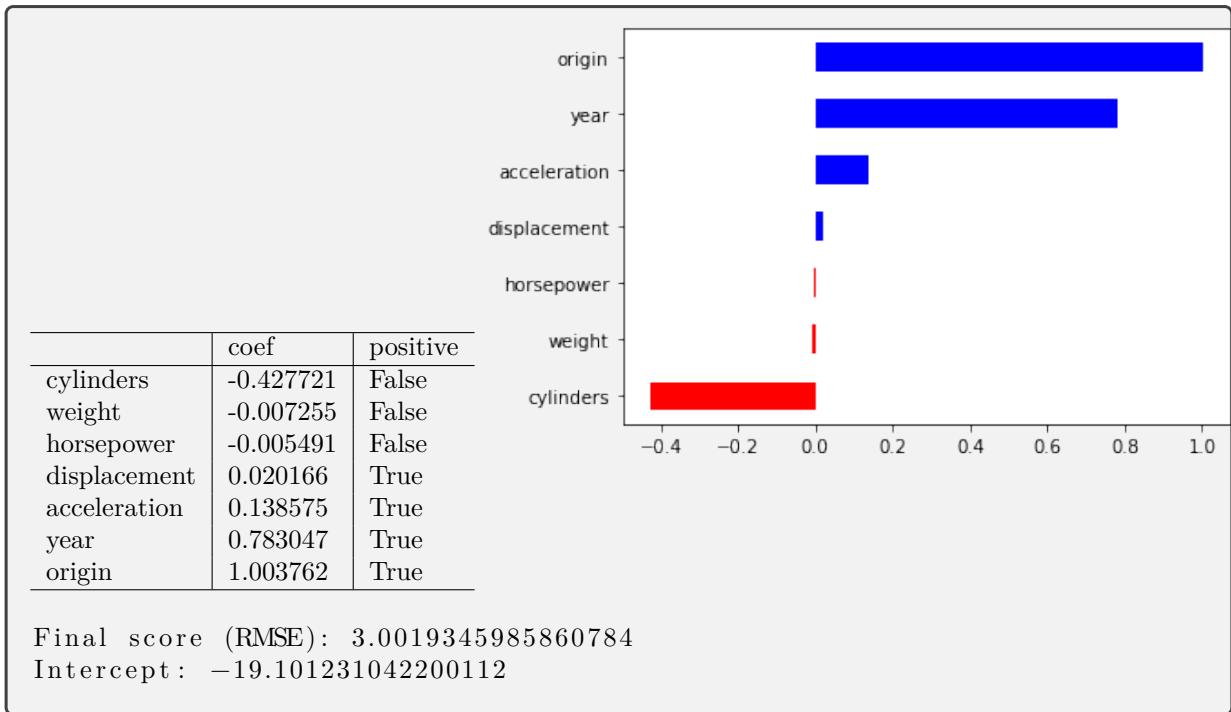
# Create linear regression
regressor = sklearn.linear_model.LinearRegression()

# 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



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) 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 the 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 many unnecessary input features, 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 the weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

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

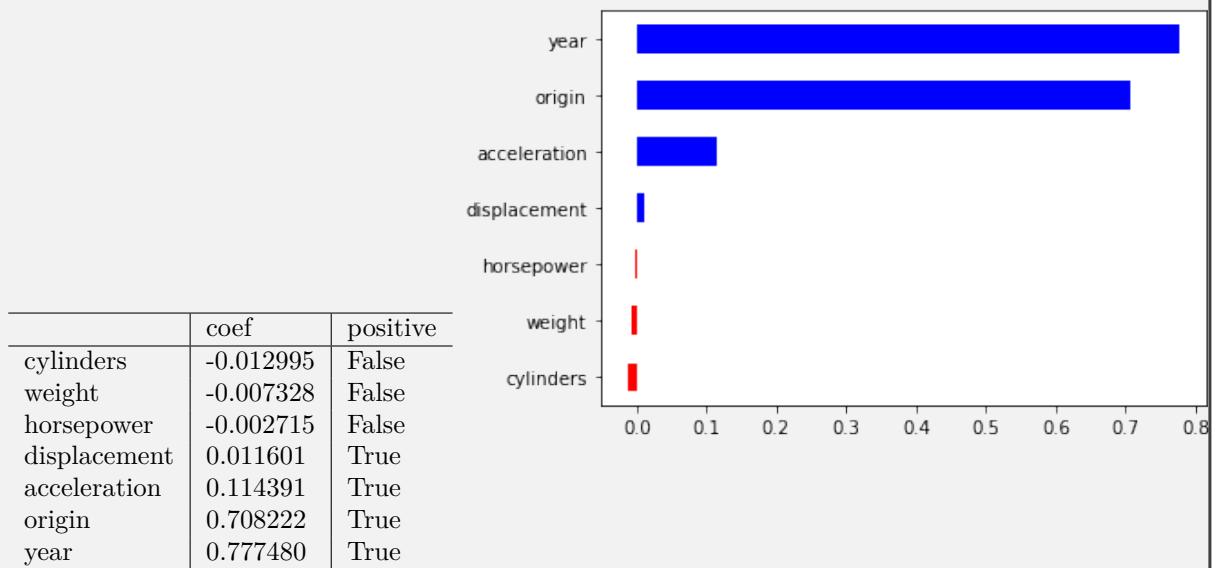
# Create linear regression
regressor = Lasso(random_state=0, alpha=0.1)

# 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



```
Final score (RMSE): 3.0604021904033303
Intercept: -18.506677982383252
```

5.1.4 L2 (Ridge) Regularization

You should use Tikhonov/Ridge/L2 regularization when you are less concerned about creating a sparse 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 algorithm will not sum bias values.

You should use L2 regularization when you are less concerned about creating a sparse 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 with many inputs, and you can prune some of the weaker inputs.

The following code uses L2 with linear regression (Ridge regression):

Code

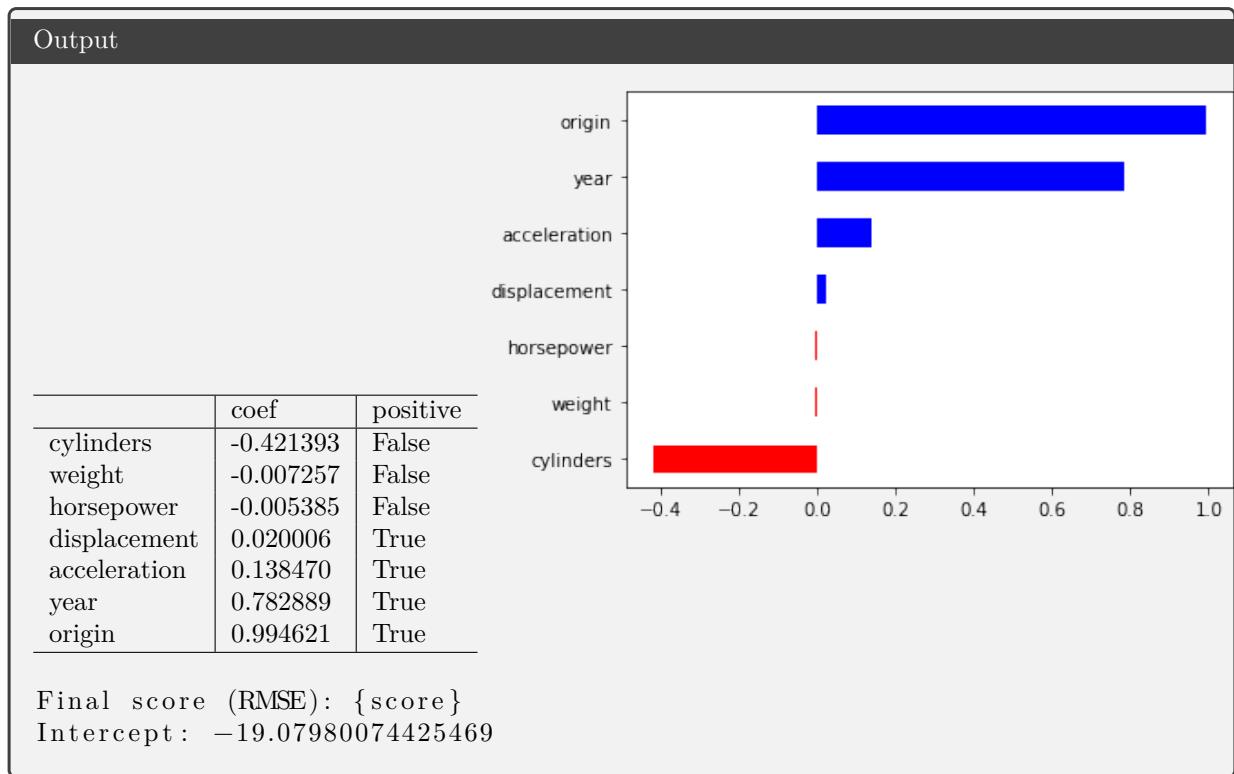
```
import sklearn
from sklearn.linear_model import Ridge

# 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): " + str(score))

report_coef(
    names,
    regressor.coef_,
    regressor.intercept_)
```



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.

$$a * \text{L1} + b * \text{L2}$$

Code

```
import sklearn
from sklearn.linear_model import ElasticNet

# Create linear regression
regressor = ElasticNet(alpha=0.1, l1_ratio=0.1)

# Fit/train LASSO
```

```

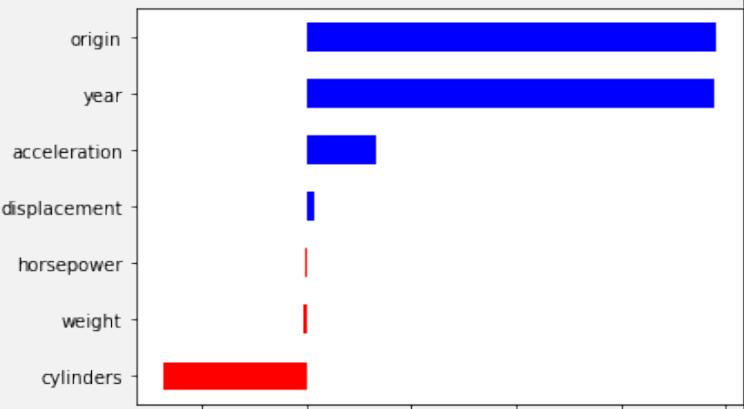
regressor.fit(x_train, y_train)
# Predict
pred = regressor.predict(x_test)

# 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



Final score (RMSE): 3.0450899960775013

Intercept: -18.389355690429767

5.2 Part 5.2: Using K-Fold Cross-validation with Keras

You can use cross-validation for a variety of purposes in predictive modeling:

- 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 several folds and multiple models to provide each data segment a chance to serve as both the validation and training set. Figure 5.1 shows cross-validation.

It is important to note that each fold will have one model (neural network). To generate predictions for new data (not present in the training set), predictions from the fold models can be handled in several ways:

- Choose the model with the highest validation score as the final model.
- Preset new data to the five 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 concerning cross-validation. Regression is the simpler case where you can break up the data set into K folds with little regard for where each item lands. For regression, the data items should fall into the folds as randomly as possible. It is also important to remember that not every fold will necessarily have 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 in the original. The balance of classes that a model was trained on must remain the same (or similar) to the training set. 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 technique is called stratified sampling and is accomplished by using the Scikit-Learn object **StratifiedKFold** in place of **KFold** whenever you use classification. In summary, you should use the following two objects in Scikit-Learn:

- **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 model is set up as 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

```
EPOCHS=500

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=EPOCHS)

    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): 0.6814299426511208
Fold #2
Fold score (RMSE): 0.45486513719487165
Fold #3
Fold score (RMSE): 0.571615041876392
Fold #4
Fold score (RMSE): 0.46416356081116916
Fold #5
Fold score (RMSE): 1.0426518491685475
Final, out of sample score (RMSE): 0.678316077597408

```

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 required.

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 the out-of-sample (predictions on the test set) results.

It is good to perform stratified k-fold cross-validation with classification data. This technique ensures that the percentages of each class remain the same across all folds. Use the **StratifiedKFold** object instead of the **KFold** object used in the 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()
    # Hidden 1
    model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
    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=EPOCHS)

    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): 0.6325
Fold #2
Fold score (accuracy): 0.6725
Fold #3
Fold score (accuracy): 0.6975
Fold #4
Fold score (accuracy): 0.6575
Fold #5
Fold score (accuracy): 0.675
Final score (accuracy): 0.667

```

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 holdout set will be the final evaluation before using your model for its real-world use. Figure ?? HOLDOUT shows this division.

The following program uses 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=EPOCHS)

    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): 0.544195299216696
Fold #2
Fold score (RMSE): 0.48070599342910353
Fold #3
Fold score (RMSE): 0.7034584765928998
Fold #4
Fold score (RMSE): 0.5397141785190473
Fold #5
Fold score (RMSE): 24.126205213080077
Cross-validated score (RMSE): 10.801732731207947
Holdout score (RMSE): 24.097657947297677

```

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[26]. 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 objective.

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. You can add this penalty calculation to the calculated gradients for gradient-descent-based algorithms, such as backpropagation. The penalty is negatively combined with the objective score for objective-function-based training, such as simulated annealing.

Both L1 and L2 work differently in that they penalize the size of the 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
```

We now create a Keras network with L1 regression.

Code

```
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()
    # Hidden 1
    model.add(Dense(50, input_dim=x.shape[1],
                   activation='relu',
                   activity_regularizer=regularizers.l1(1e-4)))
    # Hidden 2
    model.add(Dense(25, activation='relu',
                   activity_regularizer=regularizers.l1(1e-4)))
    # Output
    model.add(Dense(y.shape[1], activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    model.fit(x_train, y_train, validation_data=(x_test, y_test),
              verbose=0, epochs=500)

    oos_y.append(y_test)
    oos_pred.append(model.predict(x_test))
```

```
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.[33] Although dropout works differently than L1 and L2, it accomplishes the same goal---the prevention of overfitting. However, the algorithm does 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 technique decreases co-adaptation between neurons, which results in less overfitting.

Most neural network frameworks implement dropout as a separate layer. Dropout layers function like 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 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 and 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.

The program chooses different sets of neurons from the dropout layer during subsequent training iterations. 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 codependency developing between two neurons. Two neurons that develop 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, thereby forcing generalization.

Dropout also decreases overfitting by forcing a bootstrapping process upon the neural network. Bootstrapping is a prevalent ensemble technique. Ensembling is a technique of machine learning that combines multiple models to produce a better result than those achieved by individual models. The 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 bootstrapping programmer simply trains several neural networks to perform precisely the same task. However, each neural network 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 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 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 "{}:{}{:02d}{}".format(h, m, s)
```

5.5.1 Bootstrapping for Regression

Regression bootstrapping uses the **ShuffleSplit** object to perform the splits. This technique is similar to **KFold** for cross-validation; no balancing occurs. We will attempt to predict the age column for the **jh-simple-dataset**; the following code loads this data.

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=0.630750, mean score=0.630750, stdev=0.000000    epochs=147,
mean epochs=147  time=0:00:12.56
#2: score=1.020895, mean score=0.825823, stdev=0.195072    epochs=101,
mean epochs=124  time=0:00:08.70
#3: score=0.803801, mean score=0.818482, stdev=0.159614    epochs=155,
mean epochs=134  time=0:00:20.85
#4: score=0.540871, mean score=0.749079, stdev=0.183188    epochs=122,
mean epochs=131  time=0:00:10.64
#5: score=0.802589, mean score=0.759781, stdev=0.165240    epochs=116,
mean epochs=128  time=0:00:10.84
#6: score=0.862807, mean score=0.776952, stdev=0.155653    epochs=108,
mean epochs=124  time=0:00:10.65
#7: score=0.550373, mean score=0.744584, stdev=0.164478    epochs=131,
mean epochs=125  time=0:00:10.85
#8: score=0.659148, mean score=0.733904, stdev=0.156428    epochs=118,
...
mean epochs=116  time=0:00:09.33
#49: score=0.911419, mean score=0.747607, stdev=0.185098    epochs=124,
mean epochs=116  time=0:00:10.66
#50: score=0.599252, mean score=0.744639, stdev=0.184411    epochs=132,
mean epochs=116  time=0:00:20.91
```

The bootstrapping process for classification is similar, and I present it in the next section.

5.5.2 Bootstrapping for Classification

Regression bootstrapping uses the **StratifiedShuffleSplit** class to perform the splits. This class is similar to **StratifiedKFold** for cross-validation, as the classes are balanced so that the sampling does not affect proportions. To demonstrate this technique, we will attempt to predict the product column for the **jh-simple-dataset**; the following code loads this data.

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 now run this data through a number of splits specified by the SPLITS variable. We track the average error through each of these splits.

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=0.666342, mean score=0.666342, stdev=0.000000, epochs=66,
mean epochs=66, time=0:00:06.31
#2: score=0.645598, mean score=0.655970, stdev=0.010372, epochs=59,
mean epochs=62, time=0:00:10.63
#3: score=0.676924, mean score=0.662955, stdev=0.013011, epochs=66,
mean epochs=63, time=0:00:10.64
#4: score=0.672602, mean score=0.665366, stdev=0.012017, epochs=84,
mean epochs=68, time=0:00:08.20
#5: score=0.667274, mean score=0.665748, stdev=0.010776, epochs=73,
mean epochs=69, time=0:00:10.65
#6: score=0.706372, mean score=0.672518, stdev=0.018055, epochs=50,
mean epochs=66, time=0:00:04.81
#7: score=0.687937, mean score=0.674721, stdev=0.017565, epochs=71,
mean epochs=67, time=0:00:06.89
#8: score=0.734794, mean score=0.682230, stdev=0.025781, epochs=43,
...
mean epochs=66, time=0:00:04.14
#49: score=0.665493, mean score=0.673305, stdev=0.049060, epochs=60,
mean epochs=66, time=0:00:10.65
#50: score=0.692625, mean score=0.673691, stdev=0.048642, epochs=55,
mean epochs=65, time=0:00:05.22

```

5.5.3 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 has the best settings that I could determine. Later in this

book, 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}, "
    f"stdev={mdev:.6f}, epochs={epochs}, "
    f"mean_epochs={int(m2)}, time={hms_string(time_took)}")

```

Output

```

#1: score=0.642887, mean score=0.642887, stdev=0.000000,
epochs=325, mean epochs=325, time=0:00:42.10
#2: score=0.555518, mean score=0.599202, stdev=0.043684,

```

```
epochs=208,mean epochs=266, time=0:00:41.74
#3: score=0.605537, mean score=0.601314,stdev=0.035793,
epochs=187,mean epochs=240, time=0:00:24.22
#4: score=0.609415, mean score=0.603339,stdev=0.031195,
epochs=250,mean epochs=242, time=0:00:41.72
#5: score=0.619657, mean score=0.606603,stdev=0.028655,
epochs=201,mean epochs=234, time=0:00:26.10
#6: score=0.638641, mean score=0.611943,stdev=0.028755,
epochs=172,mean epochs=223, time=0:00:41.73
#7: score=0.671137, mean score=0.620399,stdev=0.033731,
epochs=203,mean epochs=220, time=0:00:26.58
#8: score=0.635294, mean score=0.622261,stdev=0.031935,
...
epochs=173,mean epochs=196, time=0:00:22.70
#99: score=0.697473, mean score=0.649279,stdev=0.042577,
epochs=172,mean epochs=196, time=0:00:41.79
#100: score=0.678298, mean score=0.649569,stdev=0.042462,
epochs=169,mean epochs=196, time=0:00:21.90
```

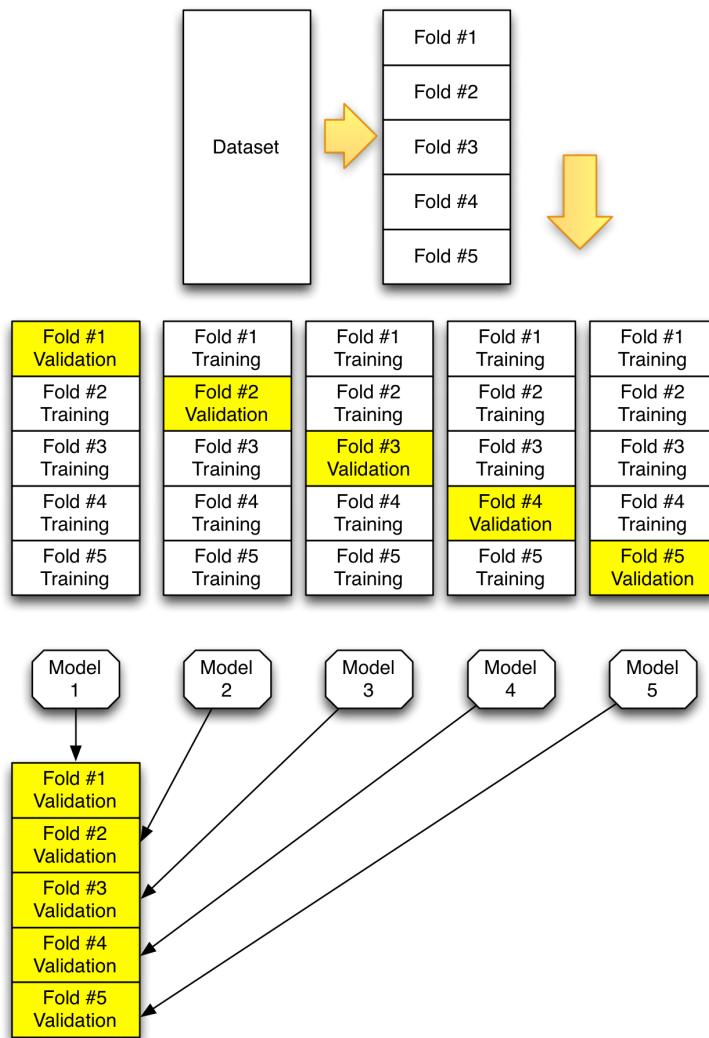


Figure 5.1: K-Fold Crossvalidation

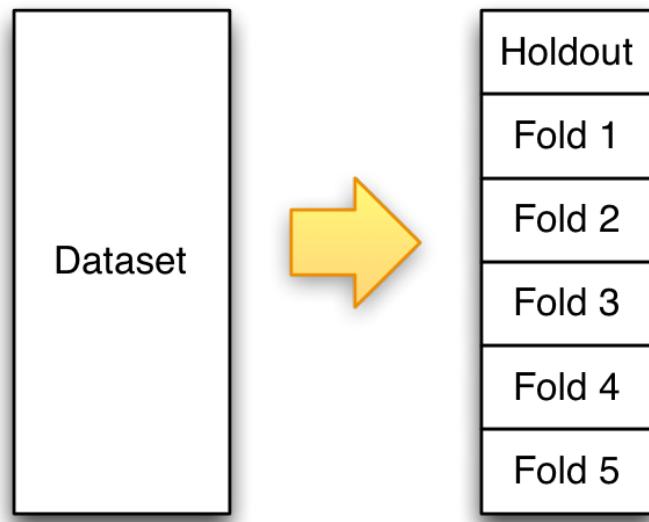


Figure 5.2: Cross-Validation and a Holdout Set

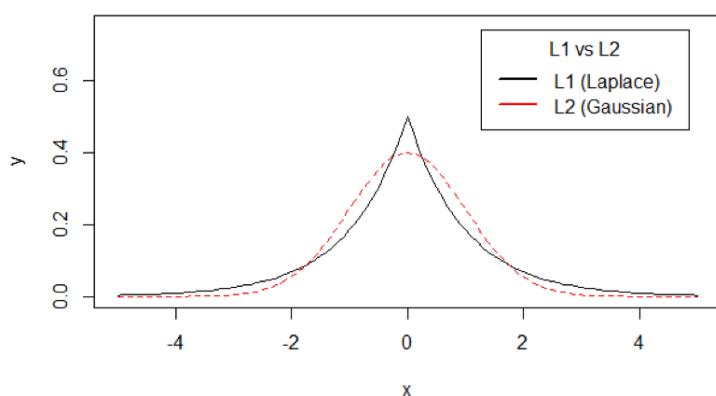


Figure 5.3: L1 vs L2

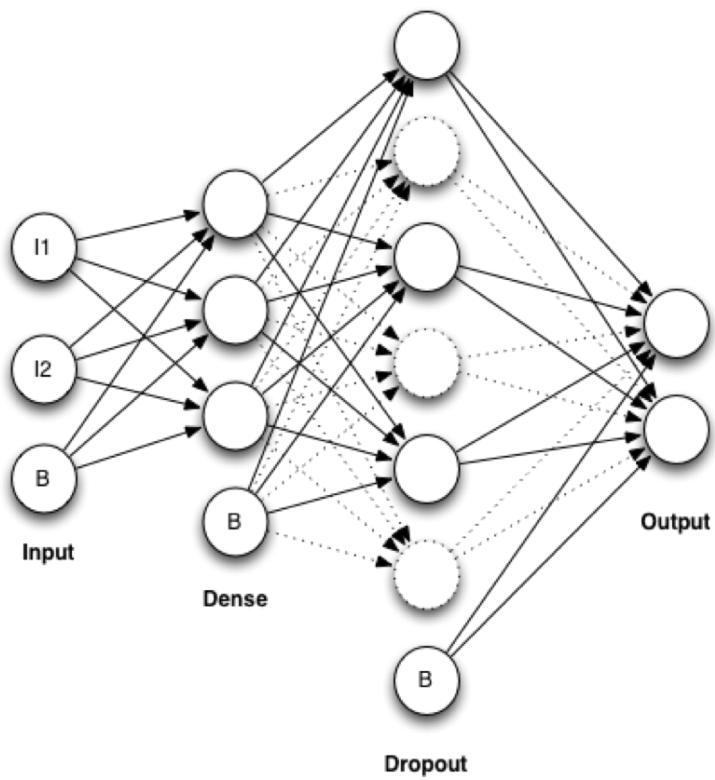


Figure 5.4: Dropout Regularization

Chapter 6

Convolutional Neural Networks (CNN) for Computer Vision

6.1 Part 6.1: Image Processing in Python

Computer vision requires processing images. These images might come from a video stream, a camera, or files on a storage drive. We begin this chapter by looking at how to process images with Python. To use images in Python, we will make use of the Pillow package. 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://data.heatonresearch.com/images/jupyter/brookings.jpeg"

response = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
img = Image.open(BytesIO(response.content))
img.load()

print(np.asarray(img))

img
```

Output



```
[[[199 213 240]
 [200 214 240]
 [200 214 240]
 ...
 [ 86   34   96]
 [ 48     4   57]
 [ 57   21   65]]
 [[199 213 239]
 [200 214 240]
 [200 214 240]
 ...
 [215 215 251]
 [252 242 255]
 [237 218 250]]
 [[200 214 240]
 ...
 [131   98   91]
 ...
 [ 86   82   57]
 [ 89   85   60]]
```

```
[ 89   85   60]]
```

6.1.1 Creating Images from Pixels in Python

You can use Pillow to create an image from a 3D NumPy cube-shaped array. The rows and columns specify the pixels. The third dimension (size 3) defines red, green, and blue color values. The following code demonstrates creating a simple image from a NumPy array.

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]

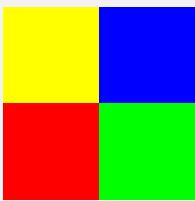
# Red
for row in range(32):
    for col in range(32):
        data[row+32, col] = [255, 0, 0]

# 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://data.heatonresearch.com/images/jupyter/brookings.jpeg"
response = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})

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: 768, Cols: 1024

6.1.3 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

images = [
    "https://data.heatonresearch.com/images/jupyter/brookings.jpeg",
    "https://data.heatonresearch.com/images/jupyter/SeigleHall.jpeg",
    "https://data.heatonresearch.com/images/jupyter/WUSTLKnight.jpeg"]
```

```
[]

def crop_square(image):
    width, height = image.size

    # Crop the image, centered
    new_width = min(width, height)
    new_height = new_width
    left = (width - new_width)/2
    top = (height - new_height)/2
    right = (width + new_width)/2
    bottom = (height + new_height)/2
    return image.crop((left, top, right, bottom))

x = []

for url in images:
    ImageFile.LOAD_TRUNCATED_IMAGES = False
    response = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
    img = Image.open(BytesIO(response.content))
    img.load()
    img = crop_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://data.heatonresearch.com/images/jupyter/brookings.jpeg>



<https://data.heatonresearch.com/images/jupyter/SeigleHall.jpeg>



<https://data.heatonresearch.com/images/jupyter/WUSTLKnight.jpeg>

(3, 49152)

6.1.4 Adding Noise to an Image

Sometimes it is beneficial to add noise to images. We might use noise to augment images to generate more training data or modify images to test the recognition capabilities of neural networks. 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://data.heatonresearch.com/images/jupyter/brookings.jpeg"

response = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
img = Image.open(BytesIO(response.content))
img.load()

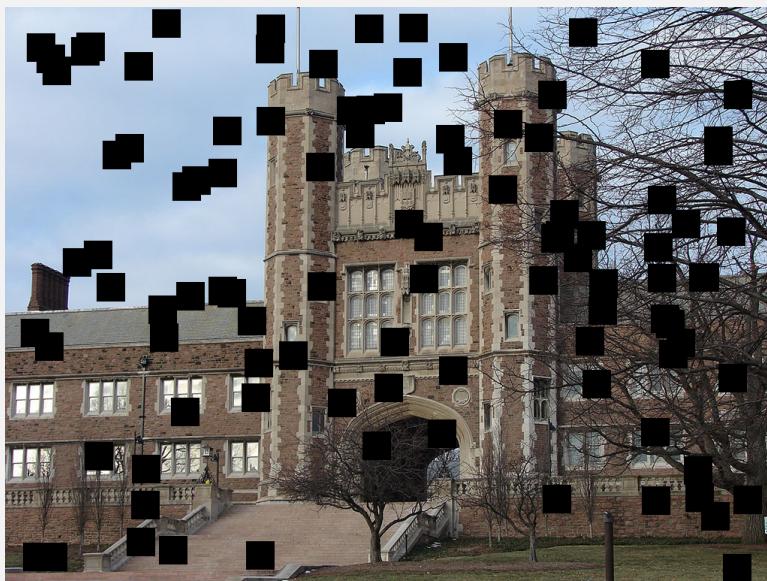
img_array = np.asarray(img)
rows = img_array.shape[0]
cols = img_array.shape[1]

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: 768, Cols: 1024
(768, 1024, 3)

6.1.5 Preprocessing Many Images

To download images, we define several paths. We will download sample images of paperclips from the URL specified by **DOWNLOAD_SOURCE**. Once downloaded, we will unzip and perform the preprocessing on these paper clips. I mean for this code as a starting point for other image preprocessing.

Code

```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases/"
#DOWNLOAD_SOURCE = URL+"download/v1/iris-image.zip"
DOWNLOAD_SOURCE = URL+"download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
    EXTRACT_TARGET = os.path.join(PATH, "clips")
    SOURCE = os.path.join(PATH, "/content/clips/paperclips")
    TARGET = os.path.join(PATH, "/content/clips-processed")
else:
```

```
# I used this locally on my machine, you may need different
PATH = "/Users/jeff/temp"
EXTRACT_TARGET = os.path.join(PATH, "clips")
SOURCE = os.path.join(PATH, "clips/paperclips")
TARGET = os.path.join(PATH, "clips-processed")
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

Code

```
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null
```

The following code contains functions that we use to preprocess the images. The **crop_square** function converts images to a square by cropping extra data. The **scale** function increases or decreases the size of an image. The **standardize** function ensures an image is full color; a mix of color and grayscale images can be problematic.

Code

```
import imageio
import glob
from tqdm import tqdm
from PIL import Image
import os

def scale(img, scale_width, scale_height):
    # Scale the image
    img = img.resize((
        scale_width,
        scale_height),
        Image.ANTIALIAS)

    return img

def standardize(image):
    rgbiimg = Image.new("RGB", image.size)
    rgbiimg.paste(image)
    return rgbiimg
```

```
def fail_below(image, check_width, check_height):
    width, height = image.size
    assert width == check_width
    assert height == check_height
```

Next, we loop through each image. The images are loaded, and you can apply any desired transformations. Ultimately, the script saves the images as JPG.

Code

```
files = glob.glob(os.path.join(SOURCE, "*.jpg"))

for file in tqdm(files):
    try:
        target = ""
        name = os.path.basename(file)
        filename, _ = os.path.splitext(name)
        img = Image.open(file)
        img = standardize(img)
        img = crop_square(img)
        img = scale(img, 128, 128)
        #fail_below(img, 128, 128)

        target = os.path.join(TARGET, filename + ".jpg")
        img.save(target, quality=25)
    except KeyboardInterrupt:
        print("KeyboardInterrupt")
        break
    except AssertionError:
        print("Assertion")
        break
    except:
        print("Unexpected exception while processing image source: " \
              f"{file}, target: {target}", exc_info=True)
```

Now we can zip the preprocessed files and store them somewhere.

6.1.6 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

This module 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 types: 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 the image to the input for a neural network.

6.2.1 Common Computer Vision Data Sets

There are many data sets for computer vision. Two of the most popular classic datasets are the MNIST digits data set and the CIFAR image data sets. We will not use either of these datasets in this course, but it is important to be familiar with them since neural network texts often refer to them.

The MNIST Digits Data Set is very popular in the neural network research community. You can see a sample of it in Figure 6.1.



Figure 6.1: MNIST 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 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. You can see this data in Figure 6.2.



Figure 6.2: MNIST Fashion 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.

6.2.2 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)[5] introduced the original concept of a convolutional neural network, and LeCun, Bottou, Bengio Haffner (1998)[20] 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.

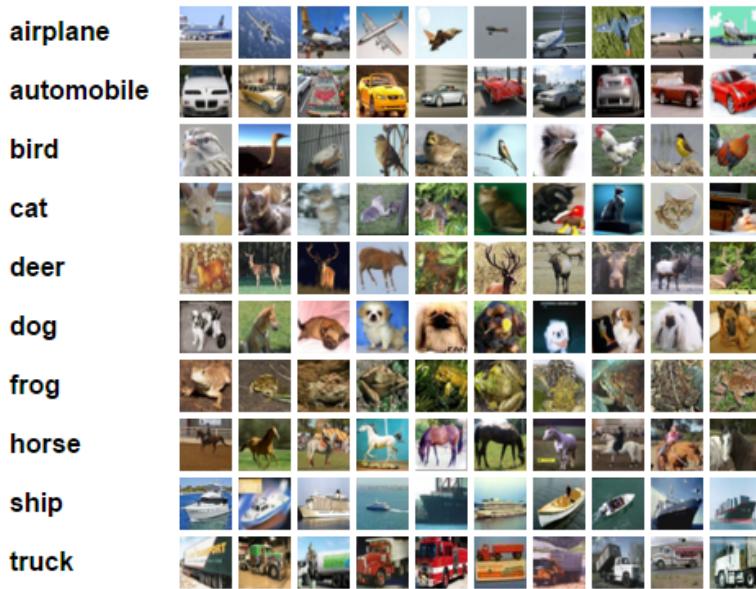


Figure 6.3: CIFAR Data Set

Although computer vision primarily uses CNNs, this technology has some applications 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 to mimic the properties of visual data.

The order of the input array elements is crucial to the training. In contrast, most neural networks that are not CNNs treat their input data as a long vector of values, and the order in which you arrange the incoming features in this vector is irrelevant. You cannot change the order for these types of neural networks after you have trained the network.

The CNN network arranges 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. 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.

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 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 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 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 can process scale, rotation, and noise in the field of computer vision.

You can see this network structure in Figure 6.4.

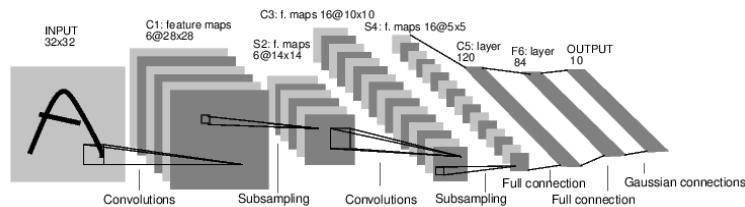


Figure 6.4: A LeNET-5 Network (LeCun, 1998)

So far, we have only seen one layer type (dense layers). By the end of this book we will have seen:

- **Dense Layers** - Fully connected layers.
- **Convolution Layers** - Used to scan across images.
- **Max Pooling Layers** - Used to downsample images.
- **Dropout Layers** - Used to add regularization.
- **LSTM and Transformer Layers** - Used for time series data.

6.2.3 Convolution Layers

The first layer that we will examine is the convolutional layer. We will begin by looking at the hyperparameters 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 of 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 we give to a convolutional layer, the more features it can see.

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 image row. There is also a hyperparameter 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 6.5 illustrates the sweep:

0	0	0	0	0	0	0	0	0	0	0
0	1	3	2	8	4	2	1	3	0	0
0	0	5	4	8	7	3	2	1	0	0
0	8	1	8	4	1	3	6	2	0	0
0	18	4	8	1	23	2	4	17	0	0
0	19	8	24	14	22	10	11	12	0	0
0	20	62	23	9	21	6	7	4	0	0
0	3	13	17	5	13	16	2	8	0	0
0	0	0	0	0	0	0	0	0	0	0

Figure 6.5: Convolutional Neural Network

The above figure shows a convolutional filter with 4 and a padding size of 1. The padding size is responsible for the border of zeros in the area that the filter sweeps. Even though the image is 8x7, the extra padding provides a virtual image size of 9x8 for the filter to sweep across. The stride specifies the number of positions the convolutional filters will stop. The convolutional filters move to the right, advancing by the number of cells specified in the stride. Once you reach the far right, 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 concerning the size of the stride. The stride cannot be 0. The convolutional filter would never move if you set the stride. 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 border, move a certain number of strides, and land on the far right or bottom border. 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.

6.2.4 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 the 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 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 is $f=2$ and $s=2$. The spatial extent (f) specifies that boxes of 2×2 will be scaled down to single pixels. Of these four pixels, the pixel with the maximum value will represent the 2×2 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 6×6 grid becomes a 3×3 :

Of course, the above diagram shows each pixel as a single number. A grayscale image would have this characteristic. We usually take the average of the three numbers for an RGB image to determine which pixel has the maximum value.

6.2.5 Regression Convolutional Neural Networks

We will now look at two examples, one for regression and another for classification. For supervised computer vision, your dataset will need some labels. For classification, this label usually specifies what the image

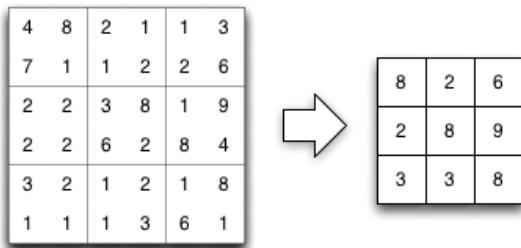


Figure 6.6: Max Pooling Layer

is a picture of. For regression, this "label" is some numeric quantity the image should produce, such as a count. We will look at two different means of providing this label.

The first example will show how to handle regression with convolution neural networks. We will provide an image and expect the neural network to count items in that image. We will use a dataset that I created that contains a random number of paperclips. The following code will download this dataset for you.

Code

```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases/"
DOWNLOAD_SOURCE = URL+"download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('')+1:]

if COLAB:
    PATH = "/content"
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"

EXTRACT_TARGET = os.path.join(PATH, "clips")
SOURCE = os.path.join(EXTRACT_TARGET, "paperclips")
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

Code

```
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
```

```
! mkdir -p {EXTRACT_TARGET}
! unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null
```

The labels are contained in a CSV file named **train.csv** for regression. This file has just two labels, **id** and **clip_count**. The ID specifies the filename; for example, row id 1 corresponds to the file **clips-1.jpg**. The following code loads the labels for the training set and creates a new column, named **filename**, that contains the filename of each image, based on the **id** column.

Code

```
import pandas as pd

df = pd.read_csv(
    os.path.join(SOURCE, "train.csv"),
    na_values=['NA', '?'])

df['filename'] = "clips-" + df["id"].astype(str) + ".jpg"
```

This results in the following dataframe.

Code

```
df
```

Output

	id	clip_count	filename
0	30001	11	clips-30001.jpg
1	30002	2	clips-30002.jpg
2	30003	26	clips-30003.jpg
3	30004	41	clips-30004.jpg
4	30005	49	clips-30005.jpg
...
19995	49996	35	clips-49996.jpg
19996	49997	54	clips-49997.jpg
19997	49998	72	clips-49998.jpg
19998	49999	24	clips-49999.jpg
19999	50000	35	clips-50000.jpg

Separate into a training and validation (for early stopping)

Code

```
TRAIN_PCT = 0.9
TRAIN_CUT = int(len(df) * TRAIN_PCT)

df_train = df[0:TRAIN_CUT]
df_validate = df[TRAIN_CUT:]

print(f"Training size:{len(df_train)}")
print(f"Validate size:{len(df_validate)}")
```

Output

```
Training size: 18000
Validate size: 2000
```

We are now ready to create two `ImageDataGenerator` objects. We currently use a generator, which creates additional training data by manipulating the source material. This technique can produce considerably stronger neural networks. The generator below flips the images both vertically and horizontally. Keras will train the neuron network both on the original images and the flipped images. This augmentation increases the size of the training data considerably. Module 6.4 goes deeper into the transformations you can perform. You can also specify a target size to resize the images automatically.

The function `flow_from_dataframe` loads the labels from a Pandas dataframe connected to our `train.csv` file. When we demonstrate classification, we will use the `flow_from_directory`; which loads the labels from the directory structure rather than a CSV.

Code

```
import tensorflow as tf
import keras.preprocessing
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(
    rescale = 1./255,
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode='nearest')

train_generator = training_datagen.flow_from_dataframe(
    dataframe=df_train,
    directory=SOURCE,
    x_col="filename",
```

```

y_col="clip_count",
target_size=(256, 256),
batch_size=32,
class_mode='other')

validation_datagen = ImageDataGenerator(rescale = 1./255)

val_generator = validation_datagen.flow_from_dataframe(
    dataframe=df_validate,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
    target_size=(256, 256),
    class_mode='other')

```

Output

```

Found 18000 validated image filenames.
Found 2000 validated image filenames.

```

We can now train the neural network. The code to build and train the neural network is not that different than in the previous modules. We will use the Keras Sequential class to provide layers to the neural network. We now have several new layer types that we did not previously see.

- **Conv2D** - The convolution layers.
- **MaxPooling2D** - The max-pooling layers.
- **Flatten** - Flatten the 2D (and higher) tensors to allow a Dense layer to process.
- **Dense** - Dense layers, the same as demonstrated previously. Dense layers often form the final output layers of the neural network.

The training code is very similar to previously. This code is for regression, so a final linear activation is used, along with mean_squared_error for the loss function. The generator provides both the x and y matrixes we previously supplied.

Code

```

from tensorflow.keras.callbacks import EarlyStopping
import time

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150
    # with 3 bytes color.
    # This is the first convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu',

```

```

        input_shape=(256, 256, 3)),
        tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='linear')
])

model.summary()
epoch_steps = 250 # needed for 2.2
validation_steps = len(df_validate)
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)

start_time = time.time()
history = model.fit(train_generator,
    verbose = 1,
    validation_data=val_generator, callbacks=[monitor], epochs=25)

elapsed_time = time.time() - start_time
print("Elapsed time: {} {}".format(hms_string(elapsed_time)))

```

Output

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 254, 254, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 127, 127, 64)	0
)		
conv2d_1 (Conv2D)	(None, 125, 125, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 62, 62, 64)	0
flatten (Flatten)	(None, 246016)	0
dense (Dense)	(None, 512)	125960704

dense_1 (Dense)	(None, 1)	513
-----------------	-----------	-----

Total params: 125,999,937

...

3.2399 - val_loss: 4.0449
 Epoch 25/25
 563/563 [=====] - 53s 94ms/step - loss:
 3.2823 - val_loss: 4.4899
 Elapsed time: 0:22:22.78

This code will run very slowly if you do not use a GPU. The above code takes approximately 13 minutes with a GPU.

6.2.6 Score Regression Image Data

Scoring/predicting from a generator is a bit different than training. We do not want augmented images, and we do not wish to have the dataset shuffled. For scoring, we want a prediction for each input. We construct the generator as follows:

- shuffle=False
- batch_size=1
- class_mode=None

We use a **batch_size** of 1 to guarantee that we do not run out of GPU memory if our prediction set is large. You can increase this value for better performance. The **class_mode** is None because there is no y , or label. After all, we are predicting.

Code

```
df_test = pd.read_csv(
    os.path.join(SOURCE, "test.csv"),
    na_values=['NA', '?'])

df_test['filename']="clips-"+df_test["id"].astype(str)+".jpg"

test_datagen = ImageDataGenerator(rescale = 1./255)

test_generator = validation_datagen.flow_from_dataframe(
    dataframe=df_test,
    directory=SOURCE,
    x_col="filename",
    batch_size=1,
    shuffle=False,
```

```
target_size=(256, 256),
class_mode=None)
```

Output

```
Found 5000 validated image filenames.
```

We need to reset the generator to ensure we are always at the beginning.

Code

```
test_generator.reset()
pred = model.predict(test_generator, steps=len(df_test))
```

We can now generate a CSV file to hold the predictions.

Code

```
df_submit = pd.DataFrame({'id': df_test['id'], 'clip_count': pred.flatten()})
df_submit.to_csv(os.path.join(PATH, "submit.csv"), index=False)
```

6.2.7 Classification Neural Networks

Just like earlier in this module, we will load data. However, this time we will use a dataset of images of three different types of the iris flower. This zip file contains three different directories that specify each image's label. The directories are named the same as the labels:

- iris-setosa
- iris-versicolour
- iris-virginica

Code

```
import os

URL = "https://github.com/jeffheaton/data-mirror/releases"
DOWNLOAD_SOURCE = URL + "/download/v1/iris-image.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
    EXTRACT_TARGET = os.path.join(PATH, "iris")
```

```
SOURCE = EXTRACT_TARGET # In this case its the same, no subfolder
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"
    EXTRACT_TARGET = os.path.join(PATH, "iris")
    SOURCE = EXTRACT_TARGET # In this case its the same, no subfolder
```

Just as before, we unzip the images.

Code

```
! wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
! mkdir -p {SOURCE}
! mkdir -p {TARGET}
! mkdir -p {EXTRACT_TARGET}
! unzip -o -d {EXTRACT_TARGET} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null
```

You can see these folders with the following command.

Code

```
! ls /content/iris
```

Output

```
iris-setosa  iris-versicolour  iris-virginica
```

We set up the generator, similar to before. This time we use flow_from_directory to get the labels from the directory structure.

Code

```
import tensorflow as tf
import keras_preprocessing
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(
    rescale = 1./255,
    horizontal_flip=True,
    vertical_flip=True,
    width_shift_range=[-200,200],
    rotation_range=360,
```

```

fill_mode='nearest')

train_generator = training_datagen.flow_from_directory(
    directory=SOURCE, target_size=(256, 256),
    class_mode='categorical', batch_size=32, shuffle=True)

validation_datagen = ImageDataGenerator(rescale = 1./255)

validation_generator = validation_datagen.flow_from_directory(
    directory=SOURCE, target_size=(256, 256),
    class_mode='categorical', batch_size=32, shuffle=True)

```

Output

```

Found 421 images belonging to 3 classes.
Found 421 images belonging to 3 classes.

```

Training the neural network with classification is similar to regression.

Code

```

from tensorflow.keras.callbacks import EarlyStopping

class_count = len(train_generator.class_indices)

model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image
    # 300x300 with 3 bytes color
    # This is the first convolution
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
        input_shape=(256, 256, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution

```

```

tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# The fifth convolution
tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# Flatten the results to feed into a DNN

tf.keras.layers.Flatten(),
tf.keras.layers.Dropout(0.5),
# 512 neuron hidden layer
tf.keras.layers.Dense(512, activation='relu'),
# Only 1 output neuron. It will contain a value from 0-1
tf.keras.layers.Dense(class_count, activation='softmax')
])

model.summary()

model.compile(loss = 'categorical_crossentropy', optimizer='adam')

model.fit(train_generator, epochs=50, steps_per_epoch=10,
          verbose = 1)

```

Output

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 254, 254, 16)	448
max_pooling2d_2 (MaxPooling2D)	(None, 127, 127, 16)	0
conv2d_3 (Conv2D)	(None, 125, 125, 32)	4640
dropout (Dropout)	(None, 125, 125, 32)	0
max_pooling2d_3 (MaxPooling2D)	(None, 62, 62, 32)	0
conv2d_4 (Conv2D)	(None, 60, 60, 64)	18496
dropout_1 (Dropout)	(None, 60, 60, 64)	0
max_pooling2d_4 (MaxPooling2D)	(None, 30, 30, 64)	0

...

```

...
10/10 [=====] - 5s 458ms/step - loss: 0.7957
Epoch 50/50
10/10 [=====] - 5s 501ms/step - loss: 0.8670

```

The iris image dataset is not easy to predict; it turns out that a tabular dataset of measurements is more manageable. However, we can achieve a 63%.

Code

```

from sklearn.metrics import accuracy_score
import numpy as np

validation_generator.reset()
pred = model.predict(validation_generator)

predict_classes = np.argmax(pred, axis=1)
expected_classes = validation_generator.classes

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

```

Output

```
Accuracy: 0.6389548693586699
```

6.2.8 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.

6.3 Part 6.3: Transfer Learning for Computer Vision

Many advanced prebuilt neural networks are available for computer vision, and Keras provides direct access to many networks. Transfer learning is the technique where you use these prebuilt neural networks. Module 9 takes a deeper look at transfer learning.

There are several different levels of transfer learning.

- Use a prebuilt neural network in its entirety
- Use a prebuilt neural network's structure
- Use a prebuilt neural network's weights

We will begin by using the MobileNet prebuilt neural network in its entirety. MobileNet will be loaded and allowed to classify simple images. We can already classify 1,000 images through this technique without ever having trained the network.

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 (Conv2D)	(None, 112, 112, 32)	864
conv1_bn (BatchNormalizatio n)	(None, 112, 112, 32)	128
conv1_relu (ReLU)	(None, 112, 112, 32)	0
conv_dw_1 (DepthwiseConv2D)	(None, 112, 112, 32)	288
conv_dw_1_bn (BatchNormaliz ation)	(None, 112, 112, 32)	128
conv_dw_1_relu (ReLU)	(None, 112, 112, 32)	0
conv_pw_1 (Conv2D)	(None, 112, 112, 64)	2048
conv_pw_1_bn (BatchNormaliz ation)	(None, 112, 112, 64)	256

...

Total params: 4,253,864
 Trainable params: 4,231,976
 Non-trainable params: 21,888

Several clues to neural network architecture become evident when examining the above structure.

We will now use the MobileNet to classify several image URLs below. You can add additional URLs 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

def classify_image (url ):
    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 (img)
    print (np .argmax (pred , axis=1))

    lst = decode_predictions (pred , top=5)
    for itm in lst [0]:
        print (itm)
```

We can now classify an example image. You can specify the URL of any image you wish to classify.

Code

```
classify_image (ROOT + " soccer_ball . jpg ")
```

Output



```
[805]
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
40960/35363 [=====] - 0s 0us/step
49152/35363 [=====] - 0s 0us/step
('n04254680', 'soccer_ball', 0.9999938)
('n03530642', 'honeycomb', 3.862412e-06)
('n03255030', 'dumbbell', 4.442458e-07)
('n02782093', 'balloon', 3.7038987e-07)
('n04548280', 'wall_clock', 3.143911e-07)
```

Code

```
classify_image(ROOT+"race_truck.jpg")
```

Output



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 imangenet, it is necessary to use transfer learning.

6.3.1 Using the Structure of ResNet

We will train a neural network to count the number of paper clips in images. We will make use of the structure of the ResNet neural network. There are several significant changes that we will make to ResNet to apply to this task. First, ResNet is a classifier; we wish to perform a regression to count. Secondly, we want to change the image resolution that ResNet uses. We will not use the weights from ResNet; changing this resolution invalidates the current weights. Thus, it will be necessary to retrain the network.

Code

```
import os
URL = "https://github.com/jeffheaton/data-mirror/"
DOWNLOAD_SOURCE = URL+"releases/download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
    PATH = "/content"
else:
    # I used this locally on my machine, you may need different
    PATH = "/Users/jeff/temp"
```

```
EXTRACT_TARGET = os.path.join(PATH, "clips")
SOURCE = os.path.join(EXTRACT_TARGET, "paperclips")
```

Output

```
[751]
('n04037443', 'racer', 0.7131951)
('n03100240', 'convertible', 0.100896776)
('n04285008', 'sports_car', 0.0770768)
('n03930630', 'pickup', 0.02635305)
('n02704792', 'amphibian', 0.011636169)
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

Code

```
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null
```

The labels are contained in a CSV file named **train.csv** for the regression. This file has just two labels, **id** and **clip_count**. The ID specifies the filename; for example, row id 1 corresponds to the file **clips-1.jpg**. The following code loads the labels for the training set and creates a new column, named **filename**, that contains the filename of each image, based on the **id** column.

Code

```
df_train = pd.read_csv(os.path.join(SOURCE, "train.csv"))
df_train['filename'] = "clips-" + df_train.id.astype(str) + ".jpg"
```

We want to use early stopping. To do this, we need a validation set. We will break the data into 80 percent test data and 20 validation. Do not confuse this validation data with the test set provided by Kaggle. This validation set is unique to your program and is for early stopping.

Code

```
TRAIN_PCT = 0.9
TRAIN_CUT = int(len(df_train) * TRAIN_PCT)
```

```
df_train_cut = df_train[0:TRAIN_CUT]
df_validate_cut = df_train[TRAIN_CUT:]

print(f"Training size: {len(df_train_cut)}")
print(f"Validate size: {len(df_validate_cut)}")
```

Output

```
Training size: 18000
Validate size: 2000
```

Next, we create the generators that will provide the images to the neural network during training. We normalize the images so that the RGB colors between 0-255 become ratios between 0 and 1. We also use the `flow_from_dataframe` generator to connect the Pandas dataframe to the actual image files. We see here a straightforward implementation; you might also wish to use some of the image transformations provided by the data generator.

The `HEIGHT` and `WIDTH` constants specify the dimensions to which the image will be scaled (or expanded). It is probably not a good idea to expand the images.

Code

```
import tensorflow as tf
import keras.preprocessing
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

WIDTH = 256
HEIGHT = 256

training_datagen = ImageDataGenerator(
    rescale = 1./255,
    horizontal_flip=True,
    #vertical_flip=True,
    fill_mode='nearest')

train_generator = training_datagen.flow_from_dataframe(
    datafram=df_train_cut ,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
    target_size=(HEIGHT, WIDTH),
    # Keeping the training batch size small
```

```

# USUALLY increases performance
batch_size=32,
class_mode='raw')

validation_datagen = ImageDataGenerator(rescale = 1./255)

val_generator = validation_datagen.flow_from_dataframe(
    dataframe=df_validate_cut,
    directory=SOURCE,
    x_col="filename",
    y_col="clip_count",
    target_size=(HEIGHT, WIDTH),
    # Make the validation batch size as large as you
    # have memory for
    batch_size=256,
    class_mode='raw')

```

Output

```

Found 18000 validated image filenames.
Found 2000 validated image filenames.

```

We will now use a ResNet neural network as a basis for our neural network. We will redefine both the input shape and output of the ResNet model, so we will not transfer the weights. Since we redefine the input, the weights are of minimal value. We begin by loading, from Keras, the ResNet50 network. We specify **include_top** as False because we will change the input resolution. We also specify **weights** as false because we must retrain the network after changing the top input layers.

Code

```

from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.layers import Input

input_tensor = Input(shape=(HEIGHT, WIDTH, 3))

base_model = ResNet50(
    include_top=False, weights=None, input_tensor=input_tensor,
    input_shape=None)

```

Now we must add a few layers to the end of the neural network so that it becomes a regression model.

Code

```
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model

x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x)
x=Dense(1024,activation='relu')(x)
model=Model(inputs=base_model.input,outputs=Dense(1)(x))
```

We train like before; the only difference is that we do not define the entire neural network here.

Code

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.metrics import RootMeanSquaredError

# Important, calculate a valid step size for the validation dataset
STEP_SIZE_VALID=val_generator.n//val_generator.batch_size

model.compile(loss = 'mean_squared_error', optimizer='adam',
               metrics=[RootMeanSquaredError(name="rmse")])
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=50, verbose=1, mode='auto',
                        restore_best_weights=True)

history = model.fit(train_generator, epochs=100, steps_per_epoch=250,
                     validation_data = val_generator, callbacks=[monitor],
                     verbose = 1, validation_steps=STEP_SIZE_VALID)
```

Output

```
...
250/250 [=====] - 61s 243ms/step - loss: 1.9211 - rmse: 1.3860 - val_loss: 17.0489 - val_rmse: 4.1290
Epoch 72/100
250/250 [=====] - 61s 243ms/step - loss: 2.3726 - rmse: 1.5403 - val_loss: 167.8536 - val_rmse: 12.9558
```

6.4 Part 6.4: Inside Augmentation

The `ImageDataGenerator` class provides many options for image augmentation. Deciding which augmentations to use can impact the effectiveness of your model. This part will visualize some of these augmentations that you might use to train your neural network. We begin by loading a sample image to augment.

Code

```
import urllib.request
import shutil
from IPython.display import Image

URL = "https://github.com/jeffheaton/t81_558_deep_learning/" +\
    "blob/master/photos/landscape.jpg?raw=true"
LOCAL_IMG_FILE = "/content/landscape.jpg"

with urllib.request.urlopen(URL) as response, \
    open(LOCAL_IMG_FILE, 'wb') as out_file:
    shutil.copyfileobj(response, out_file)

Image(filename=LOCAL_IMG_FILE)
```

Output



Next, we introduce a simple utility function to visualize four images sampled from any generator.

Code

```
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
import matplotlib.pyplot as plt
import numpy as np
import matplotlib

def visualize_generator(img_file, gen):
    # Load the requested image
    img = load_img(img_file)
    data = img_to_array(img)
    samples = expand_dims(data, 0)

    # Generat augmentations from the generator
    it = gen.flow(samples, batch_size=1)
    images = []
    for i in range(4):
        batch = it.next()
        image = batch[0].astype('uint8')
        images.append(image)

    images = np.array(images)

    # Create a grid of 4 images from the generator
    index, height, width, channels = images.shape
    nrows = index//2

    grid = (images.reshape(nrows, 2, height, width, channels)
            .swapaxes(1,2)
            .reshape(height*nrows, width*2, 3))

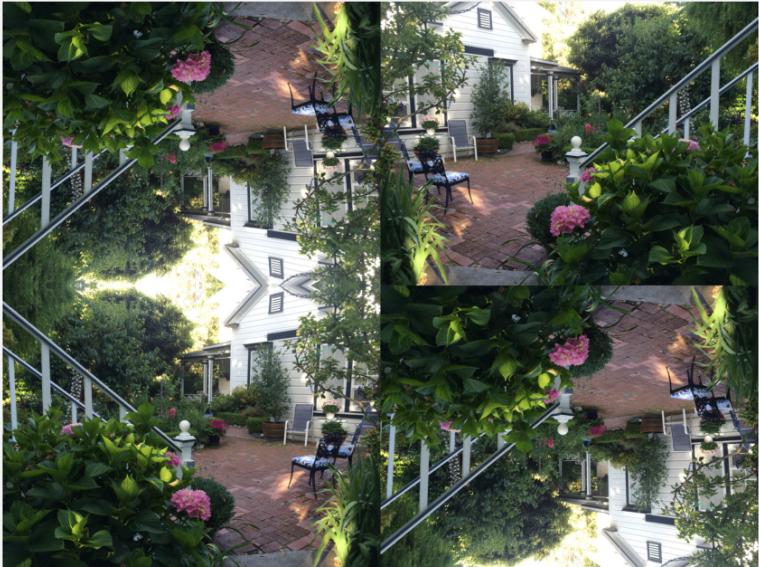
    fig = plt.figure(figsize=(15., 15.))
    plt.axis('off')
    plt.imshow(grid)
```

We begin by flipping the image. Some images may not make sense to flip, such as this landscape. However, if you expect "noise" in your data where some images may be flipped, then this augmentation may be useful, even if it violates physical reality.

Code

```
visualize_generator(
    LOCAL_IMG_FILE,
    ImageDataGenerator(horizontal_flip=True, vertical_flip=True))
```

Output

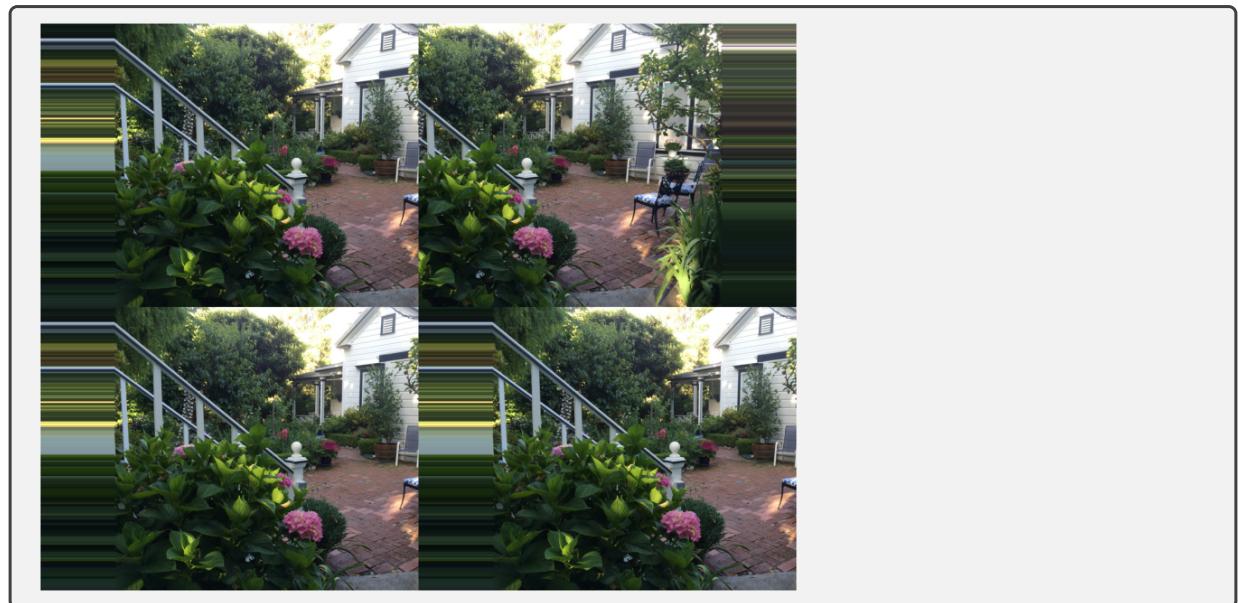


Next, we will try moving the image. Notice how part of the image is missing? There are various ways to fill in the missing data, as controlled by `fill_mode`. In this case, we simply use the nearest pixel to fill. It is also possible to rotate images.

Code

```
visualize_generator(
    LOCAL_IMG_FILE,
    ImageDataGenerator(width_shift_range=[-200,200],
        fill_mode='nearest'))
```

Output



We can also adjust brightness.

Code

```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(brightness_range=[0,1]))  
  
# brightness_range=None, shear_range=0.0
```

Output



Shearing may not be appropriate for all image types, it stretches the image.

Code

```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(shear_range=30))
```

Output



It is also possible to rotate images.

Code

```
visualize_generator(  
    LOCAL_IMG_FILE,  
    ImageDataGenerator(rotation_range=30))
```

Output



6.5 Part 6.5: Recognizing Multiple Images with YOLO5

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 recognize multiple items in a single image. One of the most advanced means of doing this is YOLOv5. You Only Look Once (YOLO) was introduced by Joseph Redmon, who supported YOLO up through V3.[28] 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. It is also possible to run YOLO on live video streams.

Joseph Redmon left computer vision to pursue other interests. The current version, YOLOv5 is supported by the startup company Ultralytics, who released the open-source library that we use in this class.[36]

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).[23] This dataset contains images labeled into 80 different classes. COCO is the source of the file coco.txt used in this module.

6.5.1 Using YOLO in Python

To use YOLO in Python, we will use the open-source library provided by Ultralytics.

- YOLOv5 GitHub

The code provided in this notebook works equally well when run either locally or from Google CoLab. It is easier to run YOLOv5 from CoLab, which is recommended for this course.

We begin by obtaining an image to classify.

Code

```
import urllib.request
import shutil
from IPython.display import Image
!mkdir /content/images/

URL = "https://github.com/jeffheaton/t81_558_deep_learning"
URL += "/raw/master/photos/jeff_cook.jpg"
LOCAL_IMG_FILE = "/content/images/jeff_cook.jpg"

with urllib.request.urlopen(URL) as response, \
    open(LOCAL_IMG_FILE, 'wb') as out_file:
    shutil.copyfileobj(response, out_file)

Image(filename=LOCAL_IMG_FILE)
```

Output



6.5.2 Installing YOLOv5

YOLO is not available directly through either PIP or CONDA. Additionally, YOLO is not installed in Google CoLab by default. Therefore, whether you wish to use YOLO through CoLab or run it locally, you need to go through several steps to install it. This section describes the process of installing YOLO. 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. You must perform these steps only once for your virtual Python environment for a local install. If you are installing locally, install to the same virtual environment you created for this course. The following commands install YOLO directly from its GitHub repository.

Code

```
! git clone https://github.com/ultralytics/yolov5 —tag 6.1  
! mv /content/6.1 /content/yolov5  
%cd /content/yolov5  
%pip install -qr requirements.txt  
  
from yolov5 import utils
```

```
display = utils.notebook_init()
```

Output

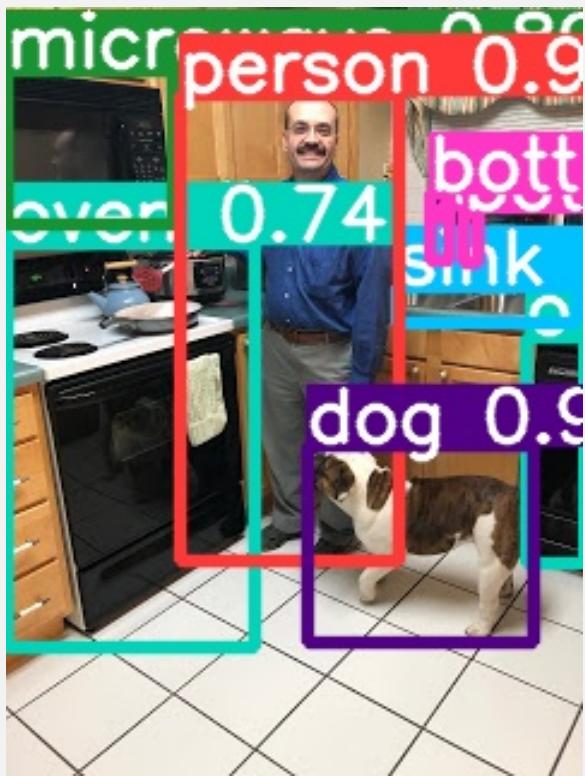
```
Setup complete (12 CPUs, 83.5 GB RAM, 39.9/166.8 GB disk)
```

Next, we will run YOLO from the command line and classify the previously downloaded kitchen picture. You can run this classification on any image you choose.

Code

```
!python detect.py --weights yolov5s.pt --img 640 \
--conf 0.25 --source /content/images/
URL = '/content/yolov5/runs/detect/exp/jeff_cook.jpg'
display.Image(filename=URL, width=300)
```

Output



```
Downloading https://ultralytics.com/assets/Arial.ttf to
/root/.config/Ultralytics/Arial.ttf...
detect: weights=['yolov5s.pt'], source=/content/images/,
data=data/coco128.yaml, imgsz=[640, 640], conf_thres=0.25,
iou_thres=0.45, max_det=1000, device=, view_img=False, save_txt=False,
save_conf=False, save_crop=False, nosave=False, classes=None,
agnostic_nms=False, augment=False, visualize=False, update=False,
project=runs/detect, name=exp, exist_ok=False, line_thickness=3,
hide_labels=False, hide_conf=False, half=False, dnn=False
YOLOv5 v6.1--85-g6f4eb95 torch 1.10.0+cu111 CUDA:0 (A100--SXM4--40GB,
40536MiB)
Downloading https://github.com/ultralytics/yolov5/releases/download/v6
.1/yolov5s.pt to yolov5s.pt...
100% 14.1M/14.1M [00:00 <00:00, 135MB/s]
Fusing layers ...
```

...

```
image 1/1 /content/images/jeff_cook.jpg: 640x480 1 person, 1 dog, 3
bottles, 1 microwave, 2 ovens, 1 sink, Done. (0.016s)
Speed: 0.6ms pre-process, 15.9ms inference, 29.3ms NMS per image at
shape (1, 3, 640, 640)
Results saved to runs/detect/exp
```

6.5.3 Running YOLOv5

In addition to the command line execution, we just saw. The following code adds the downloaded YOLOv5 to Python's environment, allowing **yolov5** to be imported like a regular Python library.

Code

```
import sys
sys.path.append(str("/content/yolov5"))

from yolov5 import utils
display = utils.notebook_init()
```

Output

```
Setup complete (12 CPUs, 83.5 GB RAM, 39.9/166.8 GB disk)
```

Next, we obtain an image to classify. For this example, the program loads the image from a URL. YOLOv5 expects that the image is in the format of a Numpy array. We use PIL to obtain this image. We will convert it to the proper format for PyTorch and YOLOv5 later.

Code

```
from PIL import Image
import requests
from io import BytesIO
import torchvision.transforms.functional as TF

url = "https://raw.githubusercontent.com/jeffheaton/" \
    "t81_558_deep_learning/master/images/cook.jpg"
response = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
img = Image.open(BytesIO(response.content))
```

The following libraries are needed to classify this image.

Code

```

import argparse
import os
import sys
from pathlib import Path

import cv2
import torch
import torch.backends.cudnn as cudnn

from models.common import DetectMultiBackend
from utils.datasets import IMG_FORMATS, VID_FORMATS, LoadImages, LoadStreams
from utils.general import (LOGGER, check_file, check_img_size, check_imshow,
                           check_requirements, colorstr,
                           increment_path, non_max_suppression,
                           print_args, scale_coords, strip_optimizer,
                           xyxy2xywh)
from utils.plots import Annotator, colors, save_one_box
from utils.torch_utils import select_device, time_sync

```

We are now ready to load YOLO with pretrained weights provided by the creators of YOLO. It is also possible to train YOLO to recognize images of your own.

Code

```

device = select_device('')
weights = '/content/yolov5/yolov5s.pt'
imgsz = [img.height, img.width]
original_size = imgsz
model = DetectMultiBackend(weights, device=device, dnn=False)
stride, names, pt, jit, onnx, engine = model.stride, model.names, \
    model.pt, model.jit, model.onnx, model.engine
imgsz = check_img_size(imgsz, s=stride) # check image size
print(f"Original size: {original_size}")
print(f"YOLO input size: {imgsz}")

```

Output

```

Original size: [320, 240]
YOLO input size: [320, 256]

```

The creators of YOLOv5 built upon PyTorch, which has a particular format for images. PyTorch

images are generally a 4D matrix of the following dimensions:

- batch_size, channels, height, width

This code converts the previously loaded PIL image into this format.

Code

```
import numpy as np
source = '/content/images/'

conf_thres=0.25 # confidence threshold
iou_thres=0.45 # NMS IOU threshold
classes = None
agnostic_nms=False, # class-agnostic NMS
max_det=1000

model.warmup(imgsz=(1, 3, *imgsz)) # warmup
dt, seen = [0.0, 0.0, 0.0], 0

# https://stackoverflow.com/questions/50657449/
# convert-image-to-proper-dimension-pytorch
img2 = img.resize([imgsz[1], imgsz[0]], Image.ANTIALIAS)

img_raw = torch.from_numpy(np.asarray(img2)).to(device)
img_raw = img_raw.float() # uint8 to fp16/32
img_raw /= 255 # 0 - 255 to 0.0 - 1.0
img_raw = img_raw.unsqueeze_(0)
img_raw = img_raw.permute(0, 3, 1, 2)
print(img_raw.shape)
```

Output

```
torch.Size([1, 3, 320, 256])
```

With the image converted, we are now ready to present the image to YOLO and obtain predictions.

Code

```
pred = model(img_raw, augment=False, visualize=False)
pred = non_max_suppression(pred, conf_thres, iou_thres, classes,
    agnostic_nms, max_det=max_det)
```

We now convert these raw predictions into the bounding boxes, labels, and confidences for each of the images that YOLO recognized.

Code

```

results = []
for i, det in enumerate(pred): # per image
    gn = torch.tensor(img_raw.shape)[[1, 0, 1, 0]]

    if len(det):
        # Rescale boxes from img_size to im0 size
        det[:, :4] = scale_coords(original_size, det[:, :4], imgsiz).round()

        # Write results
        for *xyxy, conf, cls in reversed(det):
            xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / \
                    gn).view(-1).tolist()
            # Choose between xyxy and xywh as your desired format.
            results.append([names[int(cls)], float(conf), [*xyxy]])

```

We can now see the results from the classification. We will display the first 3.

Code

```

for itm in results[0:3]:
    print(itm)

```

Output

```

['bowl', 0.28484195470809937, [tensor(55., device='cuda:0'),
tensor(120., device='cuda:0'), tensor(93., device='cuda:0'),
tensor(134., device='cuda:0')]]
['oven', 0.31531617045402527, [tensor(245., device='cuda:0'),
tensor(128., device='cuda:0'), tensor(256., device='cuda:0'),
tensor(231., device='cuda:0')]]
['bottle', 0.3567507565021515, [tensor(215., device='cuda:0'),
tensor(80., device='cuda:0'), tensor(223., device='cuda:0'),
tensor(101., device='cuda:0')]]

```

It is important to note that the **yolo** class instantiated here is a callable object, which 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 sub-image detected in the image sent to YOLO.

- **scores** - The confidence for each of the sub-images detected.
- **classes** - The string class names for each item. 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 due to the input image. The following code displays the images detected above the threshold.

To demonstrate the correctness of the results obtained, we draw bounding boxes over the original image.

Code

```
from PIL import Image, ImageDraw

img3 = img.copy()
draw = ImageDraw.Draw(img3)

for item in results:
    b = item[2]
    print(b)
    draw.rectangle(b)

img3
```

Output



```
[ tensor(55., device='cuda:0') , tensor(120., device='cuda:0') ,
tensor(93., device='cuda:0') , tensor(134., device='cuda:0')]
[ tensor(245., device='cuda:0') , tensor(128., device='cuda:0') ,
tensor(256., device='cuda:0') , tensor(231., device='cuda:0')]
[ tensor(215., device='cuda:0') , tensor(80., device='cuda:0') ,
tensor(223., device='cuda:0') , tensor(101., device='cuda:0')]
[ tensor(182., device='cuda:0') , tensor(105., device='cuda:0') ,
tensor(256., device='cuda:0') , tensor(128., device='cuda:0')]
[ tensor(200., device='cuda:0') , tensor(71., device='cuda:0') ,
tensor(210., device='cuda:0') , tensor(101., device='cuda:0')]
[ tensor(0., device='cuda:0') , tensor(96., device='cuda:0') ,
tensor(117., device='cuda:0') , tensor(269., device='cuda:0')]
[ tensor(0., device='cuda:0') , tensor(17., device='cuda:0') ,
tensor(79., device='cuda:0') , tensor(83., device='cuda:0')]
[ tensor(91., device='cuda:0') , tensor(29., device='cuda:0') ,
tensor(185., device='cuda:0') , tensor(233., device='cuda:0')]
[ tensor(142., device='cuda:0') , tensor(183., device='cuda:0') ,
tensor(253., device='cuda:0') , tensor(267., device='cuda:0')]
```

6.5.4 Module 6 Assignment

You can find the first assignment here: [assignment 6](#)

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.[10] Two neural networks compete with each other in a game. The GAN training algorithm starts with a training set and learns to generate new data with the same distributions 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.

This chapter makes use of the PyTorch framework rather than Keras/TensorFlow. While there are versions of StyleGAN2-ADA that work with TensorFlow 1.0, NVIDIA has switched to PyTorch for StyleGAN. Running this notebook in this notebook in Google CoLab is the most straightforward means of completing this chapter. Because of this, I designed this notebook to run in Google CoLab. It will take some modifications if you wish to run it locally.

This original StyleGAN paper used neural networks to automatically generate images for several previously seen datasets: MINST and CIFAR. However, it also included the Toronto Face Dataset (a private dataset used by some researchers). You can see some of these images 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 researchers significantly increased the role of convolutional neural networks for GANs.

We call a GAN a generative model because it generates new data. You can see the overall process in Figure 7.2.

7.1.1 Face Generation with StyleGAN and Python

GANs have appeared frequently in the media, showcasing their ability to generate highly photorealistic faces. One significant step forward for realistic face generation was the NVIDIA StyleGAN series. NVIDIA introduced the original StyleGAN in 2018.[17] StyleGAN was followed by StyleGAN2 in 2019, which improved the quality of StyleGAN by removing certain artifacts.[18] Most recently, in 2020, NVIDIA released

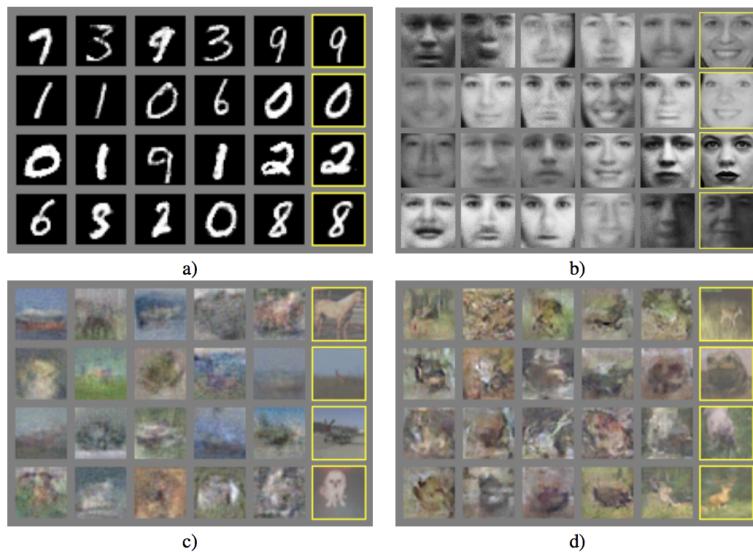


Figure 7.1: GAN Generated Images

StyleGAN2 adaptive discriminator augmentation (ADA), which will be the focus of this module.[16] We will see both how to train StyleGAN2 ADA on any arbitrary set of images; as well as use pretrained weights provided by NVIDIA. The NVIDIA weights allow us to generate high resolution photorealistic looking faces, such seen in Figure 7.3.

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](#).

The primary advancement introduced by the adaptive discriminator augmentation is that the algorithm augments the training images in real-time. Image augmentation is a common technique in many convolution neural network applications. Augmentation has the effect of increasing the size of the training set. Where StyleGAN2 previously required over 30K images for an effective to develop an effective neural network; now much fewer are needed. I used 2K images to train the fish generating GAN for this section. Figure 7.4 demonstrates the ADA process.

The figure shows the increasing probability of augmentation as p increases. For small image sets, the discriminator will generally memorize the image set unless the training algorithm makes use of augmentation. Once this memorization occurs, the discriminator is no longer providing useful information to the training of the generator.

While the above images look much more realistic than images generated earlier in this course, they are not perfect. Look at Figure 7.5. There are usually several 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 what a GAN-generated background is. Also, look at the image character's left eye. It is

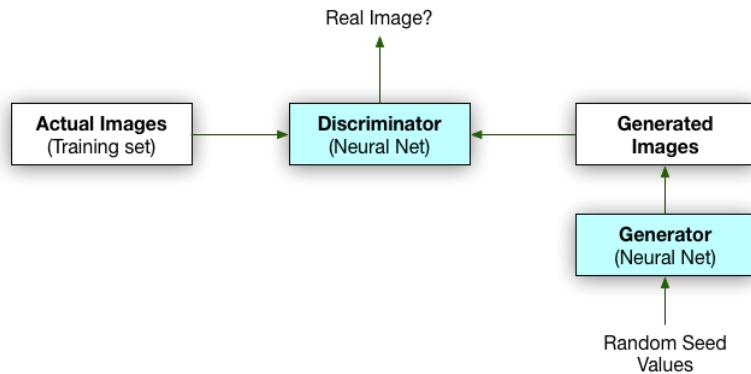


Figure 7.2: GAN Structure



Figure 7.3: StyleGAN2 Generated Faces

slightly unrealistic looking, especially near the eyelashes.

Look at the following GAN face. Can you spot any imperfections?

- Image A demonstrates the 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 and a highly distorted secondary image.
- Image D also contains a highly distorted secondary image that might be a hand.

Several websites allow you to generate GANs of your own without any software.

- 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 face image.

In this chapter, you will learn to create your own StyleGAN pictures using Python.

7.1.2 Generating High Rez GAN Faces with Google CoLab

This notebook demonstrates how to run NVidia StyleGAN2 ADA inside 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

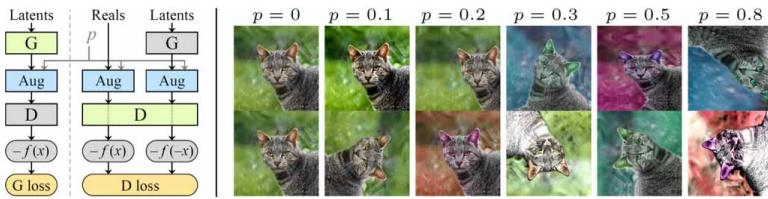


Figure 7.4: StyleGAN2 ADA Training

will run into compute limitations of Google CoLab. Make sure to run this code on a GPU instance. GPU is assumed.

First, we clone StyleGAN3 from GitHub.

Code

```
!git clone https://github.com/NVlabs/stylegan3.git
!pip install ninja
```

Verify that StyleGAN has been cloned.

Code

```
!ls /content/stylegan3
```

Output

avg_spectra.py	Dockerfile	gen_video.py	metrics	train.py
calc_metrics.py	docs	gui_utils	README.md	
visualizer.py				
dataset_tool.py	environment.yml	legacy.py	torch_utils	viz
dnnlib	gen_images.py	LICENSE.txt	training	

7.1.3 Run StyleGan From Command Line

Add the StyleGAN folder to Python so that you can import it. I based this code below on code from NVidia for the original StyleGAN paper. When you use StyleGAN you will generally create a GAN from a seed number. This seed is an integer, such as 6600, that will generate a unique image. The seed generates a latent vector containing 512 floating-point values. The GAN code uses the seed to generate these 512 values. The seed value is easier to represent in code than a 512 value vector; however, while a small change to the latent vector results in a slight change to the image, even a small change to the integer seed value will produce a radically different image.

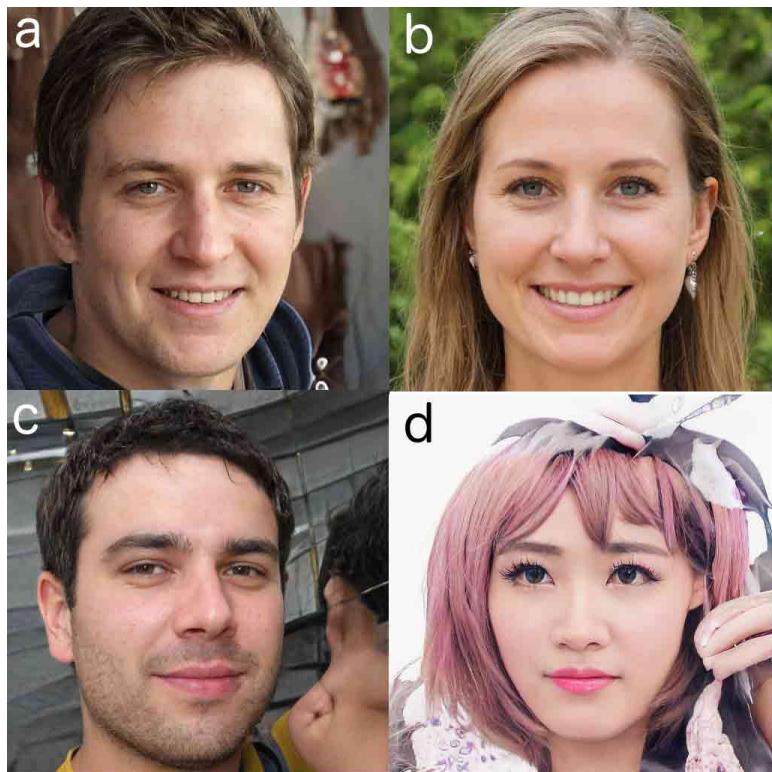


Figure 7.5: StyleGAN2 Face

Code

```
URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/" \
      "stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

!python /content/stylegan3/gen_images.py \
    --network={URL} \
    --outdir=/content/results --seeds=6600-6625
```

We can now display the images created.

Code

```
!ls /content/results
```

Output

```
seed6600.png  seed6606.png  seed6612.png  seed6618.png  seed6624.png  
seed6601.png  seed6607.png  seed6613.png  seed6619.png  seed6625.png  
seed6602.png  seed6608.png  seed6614.png  seed6620.png  
seed6603.png  seed6609.png  seed6615.png  seed6621.png  
seed6604.png  seed6610.png  seed6616.png  seed6622.png  
seed6605.png  seed6611.png  seed6617.png  seed6623.png
```

Next, copy the images to a folder of your choice on GDrive.

Code

```
!cp /content/results/* \  
     /content/drive/My\ Drive/projects/stylegan3
```

7.1.4 Run StyleGAN From Python Code

Add the StyleGAN folder to Python so that you can import it.

Code

```
import sys  
sys.path.insert(0, "/content/stylegan3")  
import pickle  
import os  
import numpy as np  
import PIL.Image  
from IPython.display import Image  
import matplotlib.pyplot as plt  
import IPython.display  
import torch  
import dnnlib  
import legacy  
  
def seed2vec(G, seed):  
    return np.random.RandomState(seed).randn(1, G.z_dim)  
  
def display_image(image):  
    plt.axis('off')  
    plt.imshow(image)  
    plt.show()
```

```

def generate_image(G, z, truncation_psi):
    # Render images for dlatents initialized from random seeds.
    Gs_kwargs = {
        'output_transform': dict(func=tflib.convert_images_to_uint8,
                               nchw_to_nhwc=True),
        'randomize_noise': False
    }
    if truncation_psi is not None:
        Gs_kwargs['truncation_psi'] = truncation_psi

    label = np.zeros([1] + G.input_shapes[1][1:])
    # [minibatch, height, width, channel]
    images = G.run(z, label, **Gs_kwargs)
    return images[0]

def get_label(G, device, class_idx):
    label = torch.zeros([1, G.c_dim], device=device)
    if G.c_dim != 0:
        if class_idx is None:
            ctx.fail("Must specify class label with --class when using a conditional network")
        label[:, class_idx] = 1
    else:
        if class_idx is not None:
            print ("warn: class=lbl ignored when running on an unconditional network")
    return label

def generate_image(device, G, z, truncation_psi=1.0, noise_mode='const',
                   class_idx=None):
    z = torch.from_numpy(z).to(device)
    label = get_label(G, device, class_idx)
    img = G(z, label, truncation_psi=truncation_psi, noise_mode=noise_mode)
    img = (img.permute(0, 2, 3, 1) * 127.5 + 128).clamp(0, 255).to(torch.uint8)
    return PIL.Image.fromarray(img[0].cpu().numpy(), 'RGB')

```

Code

```

#URL = "https://github.com/jeffheaton/pretrained-gan-fish/releases/"\
# "download/1.0.0/fish-gan-2020-12-09.pkl"\
#URL = "https://github.com/jeffheaton/pretrained-merry-gan-mas/releases/"\

```

```
# "download/v1/christmas-gan-2020-12-03.pkl"
URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/" \
    "versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

print(f'Loading networks from "{URL}"... ')
device = torch.device('cuda')
with dnnlib.util.open_url(URL) as f:
    G = legacy.load_network_pkl(f)[ 'G_ema' ].to(device) # type: ignore
```

Output

Loading networks from "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...

We can now generate images from integer seed codes in Python.

Code

```
# Choose your own starting and ending seed.
SEED_FROM = 1000
SEED_TO = 1003

# Generate the images for the seeds.
for i in range(SEED_FROM, SEED_TO):
    print(f"Seed {i}")
    z = seed2vec(G, i)
    img = generate_image(device, G, z)
    display_image(img)
```

Output



Seed 1000

Setting up PyTorch plugin "bias_act_plugin"... Done.

Setting up PyTorch plugin "filtered_lrelu_plugin"... Done.



Seed 1001



Seed 1002

7.1.5 Examining the Latent Vector

Figure 7.6 shows the effects of transforming the latent vector between two images. We accomplish this transformation by slowly moving one 512-value latent vector to another 512 vector. A high-dimension point between two latent vectors will appear similar to both of the two endpoint latent vectors. Images that have similar latent vectors will appear similar to each other.



Figure 7.6: Transforming the Latent Vector

Code

```
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

#URL = "https://github.com/jeffheaton/pretrained-gan-fish/releases/" \
# "download/1.0.0/fish-gan-2020-12-09.pkl"
#URL = "https://github.com/jeffheaton/pretrained-merry-gan-mas/releases/" \
# "download/v1/christmas-gan-2020-12-03.pkl"
#URL = "https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada/pretrained/ffhq.pkl"
URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/" \
"versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

print(f'Loading networks from "{URL}..."')
device = torch.device('cuda')
with dnnlib.util.open_url(URL) as f:
    G = legacy.load_network_pkl(f)[ 'G_ema' ].to(device) # type: ignore

vector_size = G.z_dim
# 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 "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...
(1, 512)

```

The following code will move between the provided seeds. The constant STEPS specify how many frames there should be between each seed.

Code

```

# Choose your seeds to morph through and the number of steps to
# take to get to each.

SEEDS = [6624,6618,6616] # Better for faces
#SEEDS = [1000,1003,1001] # Better for fish
STEPS = 100

# Remove any prior results
!rm /content/results/*

```

```

from tqdm.notebook import tqdm
os.makedirs("./results/", exist_ok=True)

# Generate the images for the video.
idx = 0
for i in range(len(SEEDS)-1):
    v1 = seed2vec(G, SEEDS[i])
    v2 = seed2vec(G, SEEDS[i+1])

    diff = v2 - v1
    step = diff / STEPS
    current = v1.copy()

    for j in tqdm(range(STEPS), desc=f"Seed {SEEDS[i]}"):
        current = current + step
        img = generate_image(device, G, current)
        img.save(f'./results/frame-{idx}.png')
        idx+=1

# Link the images into a video.
!ffmpeg -r 30 -i /content/results/frame-%d.png -vcodec mpeg4 -y movie.mp4

```

You can now download the generated video.

Code

```

from google.colab import files
files.download('movie.mp4')

```

Output

```

<IPython.core.display.Javascript
object><IPython.core.display.Javascript object>

```

7.1.6 Module 7 Assignment

You can find the first assignment here: [assignment 7](#)

7.2 Part 7.2: Train StyleGAN3 with your Images

Training GANs with StyleGAN is resource-intensive. The NVIDIA StyleGAN researchers used computers with eight high-end GPUs for the high-resolution face GANs trained by NVIDIA. The GPU used by NVIDIA is an A100, which has more memory and cores than the P100 or V100 offered by even Colab Pro+. In this part, we will use StyleGAN2 to train rather than StyleGAN3. You can use networks trained with StyleGAN2 from StyleGAN3; however, StyleGAN3 usually is more effective at training than StyleGAN2.

Unfortunately, StyleGAN3 is compute-intensive and will perform slowly on any GPU that is not the latest Ampere technology. Because Colab does not provide such technology, I am keeping the training guide at the StyleGAN2 level. Switching to StyleGAN3 is relatively easy, as will be pointed out later.

Make sure that you are running this notebook with a GPU runtime. You can train GANs with either Google Colab Free or Pro. I recommend at least the Pro version due to better GPU instances, longer runtimes, and timeouts. Additionally, the capability of Google Colab Pro to run in the background is valuable when training GANs, as you can close your browser or reboot your laptop while training continues.

You will store your training data and trained neural networks to GDRIVE. For GANs, I lay out my GDRIVE like this:

- ./data/gan/images - RAW images I wish to train on.
- ./data/gan/datasets - Actual training datasets that I convert from the raw images.
- ./data/gan/experiments - The output from StyleGAN2, my image previews, and saved network snapshots.

You will mount the drive at the following location.

```
/content/drive/MyDrive/data
```

7.2.1 What Sort of GPU do you Have?

The type of GPU assigned to you by Colab will significantly affect your training time. Some sample times that I achieved with Colab are given here. I've found that Colab Pro generally starts you with a V100, however, if you run scripts non-stop for 24hrs straight for a few days in a row, you will generally be throttled back to a P100.

- 1024x1024 - V100 - 566 sec/tick (CoLab Pro)
- 1024x1024 - P100 - 1819 sec/tick (CoLab Pro)
- 1024x1024 - T4 - 2188 sec/tick (CoLab Free)

By comparison, a 1024x1024 GAN trained with StyleGAN3 on a V100 is 3087 sec/tick.

If you use Google CoLab Pro, generally, it will not disconnect before 24 hours, even if you (but not your script) are inactive. Free CoLab WILL disconnect a perfectly good running script if you do not interact for a few hours. The following describes how to circumvent this issue.

- How to prevent Google Colab from disconnecting?

7.2.2 Set Up New Environment

You will likely need to train for >24 hours. Colab will disconnect you. You must be prepared to restart training when this eventually happens. Training is divided into ticks, every so many ticks (50 by default), your neural network is evaluated, and a snapshot is saved. When CoLab shuts down, all training after the last snapshot is lost. It might seem desirable to snapshot after each tick; however, this snapshotting process itself takes nearly an hour. Learning an optimal snapshot size for your resolution and training data is important.

We will mount GDRIVE so that you will save your snapshots there. You must also place your training images in GDRIVE.

You must also install NVIDIA StyleGAN2 ADA PyTorch. We also need to downgrade PyTorch to a version that supports StyleGAN.

Code

```
! pip install torch==1.8.1 torchvision==0.9.1
! git clone https://github.com/NVlabs/stylegan2-ada-pytorch.git
! pip install ninja
```

7.2.3 Find Your Files

The drive is mounted to the following location.

```
/content/drive/MyDrive/data
```

It might be helpful to use an `ls` command to establish the exact path for your images.

Code

```
!ls /content/drive/MyDrive/data/gan/images
```

7.2.4 Convert Your Images

You must convert your images into a data set form that PyTorch can directly utilize. The following command converts your images and writes the resulting data set to another directory.

Code

```
CMD = "python /content/stylegan2-ada-pytorch/dataset_tool.py" \
    "--source /content/drive/MyDrive/data/gan/images/circuit" \
    "--dest /content/drive/MyDrive/data/gan/dataset/circuit"
```

```
!{CMD}
```

You can use the following command to clear out the newly created dataset. If something goes wrong and you need to clean up your images and rerun the above command, you should delete your partially completed dataset directory.

Code

```
#!rm -R /content/drive/MyDrive/data/gan/dataset/circuit/*
```

7.2.5 Clean Up your Images

All images must have the same dimensions and color depth. This code can identify images that have issues.

Code

```
from os import listdir
from os.path import isfile, join
import os
from PIL import Image
from tqdm.notebook import tqdm

IMAGE_PATH = '/content/drive/MyDrive/data/gan/images/fish'
files = [f for f in listdir(IMAGE_PATH) if isfile(join(IMAGE_PATH, f))]

base_size = None
for file in tqdm(files):
    file2 = os.path.join(IMAGE_PATH, file)
    img = Image.open(file2)
    sz = img.size
    if base_size and sz != base_size:
        print(f"Inconsistent size: {file2}")
    elif img.mode != 'RGB':
        print(f"Inconsistent color format: {file2}")
    else:
        base_size = sz
```

7.2.6 Perform Initial Training

This code performs the initial training. Set SNAP low enough to get a snapshot before Colab forces you to quit.

Code

```
import os
```

```
# Modify these to suit your needs
EXPERIMENTS = "/content/drive/MyDrive/data/gan/experiments"
DATA = "/content/drive/MyDrive/data/gan/dataset/circuit"
SNAP = 10

# Build the command and run it
cmd = f"/usr/bin/python3 /content/stylegan2-ada-pytorch/train.py" \
    f"--snap={SNAP} --outdir={EXPERIMENTS} --data={DATA}"
!{cmd}
```

7.2.7 Resume Training

You can now resume training after you are interrupted by something in the previous step.

Code

```
import os

# Modify these to suit your needs
EXPERIMENTS = "/content/drive/MyDrive/data/gan/experiments"
NETWORK = "network-snapshot-000100.pkl"
RESUME = os.path.join(EXPERIMENTS, \
                      "00008-circuit-autol-resumecustom", NETWORK)
DATA = "/content/drive/MyDrive/data/gan/dataset/circuit"
SNAP = 10

# Build the command and run it
cmd = f"/usr/bin/python3 /content/stylegan2-ada-pytorch/train.py" \
    f"--snap={SNAP} --resume={RESUME} --outdir={EXPERIMENTS} --data={DATA}"
!{cmd}
```

7.3 Part 7.3: Exploring the StyleGAN Latent Vector

StyleGAN seeds, such as 3000, are only random number seeds used to generate much longer 512-length latent vectors, which create the GAN image. If you make a small change to the seed, for example, change 3000 to 3001, StyleGAN will create an entirely different picture. However, if you make a small change to a few latent vector values, the image will only change slightly. In this part, we will see how we can fine-tune the latent vector to control, to some degree, the resulting GAN image appearance.

7.3.1 Installing Needed Software

We begin by installing StyleGAN.

Code

```
!git clone https://github.com/NVlabs/stylegan3.git
!pip install ninja
```

We will use the same functions introduced in the previous part to generate GAN seeds and images.

Code

```
import sys
sys.path.insert(0, "/content/stylegan3")
import pickle
import os
import numpy as np
import PIL.Image
from IPython.display import Image
import matplotlib.pyplot as plt
import IPython.display
import torch
import dnnlib
import legacy

def seed2vec(G, seed):
    return np.random.RandomState(seed).randn(1, G.z_dim)

def display_image(image):
    plt.axis('off')
    plt.imshow(image)
    plt.show()

def generate_image(G, z, truncation_psi):
    # Render images for dlatents initialized from random seeds.
    Gs_kwargs = {
        'output_transform': dict(func=tflib.convert_images_to_uint8,
                               nchw_to_nhwc=True),
        'randomize_noise': False
    }
    if truncation_psi is not None:
        Gs_kwargs['truncation_psi'] = truncation_psi

    label = np.zeros([1] + G.input_shapes[1][1:])
```

```

# [minibatch, height, width, channel]
images = G.run(z, label, **G_kwargs)
return images[0]

def get_label(G, device, class_idx):
    label = torch.zeros([1, G.c_dim], device=device)
    if G.c_dim != 0:
        if class_idx is None:
            ctx.fail('Must specify class label with --class '\
                     'when using a conditional network')
        label[:, class_idx] = 1
    else:
        if class_idx is not None:
            print('warn: --class=lbl ignored when running '\
                  'on an unconditional network')
    return label

def generate_image(device, G, z, truncation_psi=1.0,
                   noise_mode='const', class_idx=None):
    z = torch.from_numpy(z).to(device)
    label = get_label(G, device, class_idx)
    img = G(z, label, truncation_psi=truncation_psi,
            noise_mode=noise_mode)
    img = (img.permute(0, 2, 3, 1) * 127.5 + 128) \
        .clamp(0, 255).to(torch.uint8)
    return PIL.Image.fromarray(img[0].cpu().numpy(), 'RGB')

```

Next, we load the NVIDIA FFHQ (faces) GAN. We could use any StyleGAN pretrained GAN network here.

Code

```

# HIDE CODE

URL = "https://api.ngc.nvidia.com/v2/models/nvidia/research/" \
    "stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"

print('Loading networks from %s' % URL)
device = torch.device('cuda')
with dnnlib.util.open_url(URL) as fp:
    G = legacy.load_network_pkl(fp)['G_ema'] \
        .requires_grad_(False).to(device)

```

Output

```
Loading networks from "https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl"...
Downloading https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/stylegan3-r-ffhq-1024x1024.pkl ... done
```

7.3.2 Generate and View GANS from Seeds

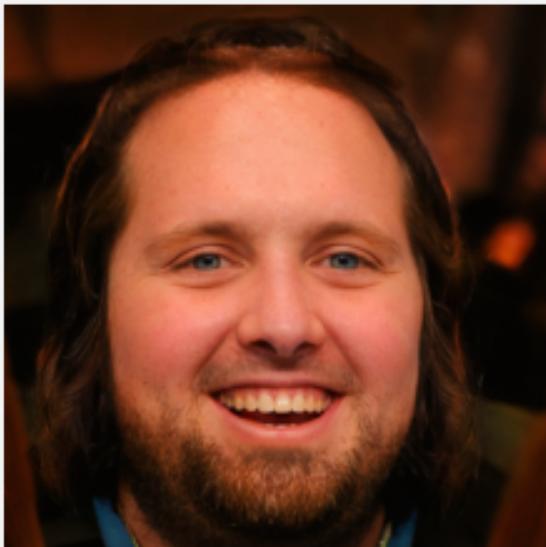
We will begin by generating a few seeds to evaluate potential starting points for our fine-tuning. Try out different seeds ranges until you have a seed that looks close to what you wish to fine-tune.

Code

```
# Choose your own starting and ending seed.
SEED_FROM = 4020
SEED_TO = 4023

# Generate the images for the seeds.
for i in range(SEED_FROM, SEED_TO):
    print(f"Seed {i}")
    z = seed2vec(G, i)
    img = generate_image(device, G, z)
    display_image(img)
```

Output



```
Seed 4020
```

```
Setting up PyTorch plugin "bias_act_plugin"... Done.
```

```
Setting up PyTorch plugin "filtered_lrelu_plugin"... Done.
```

```
...
```

7.3.3 Fine-tune an Image

If you find a seed you like, you can fine-tune it by directly adjusting the latent vector. First, choose the seed to fine-tune.

Code

```
START_SEED = 4022  
  
current = seed2vec(G, START_SEED)
```

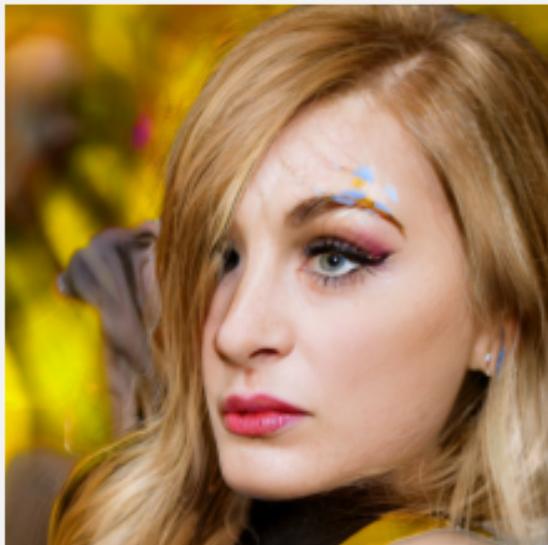
Next, generate and display the current vector. You will return to this point for each iteration of the finetuning.

Code

```
img = generate_image(device, G, current)
```

```
SCALE = 0.5  
display_image(img)
```

Output



Choose an explore size; this is the number of different potential images chosen by moving in 10 different directions. Run this code once and then again anytime you wish to change the ten directions you are exploring. You might change the ten directions if you are no longer seeing improvements.

Code

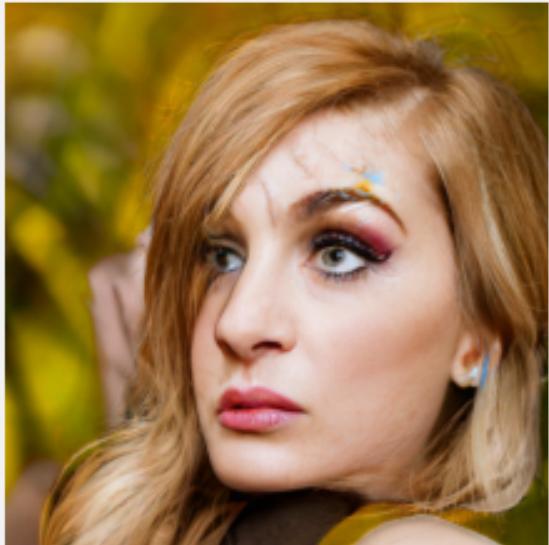
```
EXPLORE_SIZE = 25  
  
explore = []  
for i in range(EXPLORE_SIZE):  
    explore.append( np.random.rand(1, 512) - 0.5 )
```

Each image displayed from running this code shows a potential direction that we can move in the latent vector. Choose one image that you like and change MOVE_DIRECTION to indicate this decision. Once you rerun the code, the code will give you a new set of potential directions. Continue this process until you have a latent vector that you like.

Code

```
# Choose the direction to move. Choose -1 for the initial iteration.  
MOVE_DIRECTION = -1  
SCALE = 0.5  
  
if MOVE_DIRECTION >=0:  
    current = current + explore [MOVE_DIRECTION]  
  
for i , mv in enumerate(explore):  
    print(f"Direction {i}")  
    z = current + mv  
    img = generate_image(device , G, z)  
    display_image(img)
```

Output



Direction 0

...

7.4 Part 7.4: GANS to Enhance Old Photographs Deoldify

For the last two parts of this module, we will examine two applications of GANs. The first application is named deoldify, which uses a PyTorch-based GAN to transform old photographs into more modern-looking images. The complete source code to Deoldify is provided, along with several examples notebooks upon which I based this part.

7.4.1 Install Needed Software

We begin by cloning the deoldify repository.

Code

```
! git clone https://github.com/jantic/DeOldify.git DeOldify  
%cd DeOldify
```

Install any additional Python packages needed.

Code

```
! pip install -r colab_requirements.txt
```

Install the pretrained weights for deoldify.

Code

```
! mkdir './models/'  
CMD = "wget -https://data.deepai.org/deoldify/ColorizeArtistic_gen.pth" \  
      "-O ./models/ColorizeArtistic_gen.pth"  
!{CMD}
```

The authors of deoldify suggest that you might wish to include a watermark to let others know that AI-enhanced this picture. The following code downloads this standard watermark. The authors describe the watermark as follows:

"This places a watermark icon of a palette at the bottom left corner of the image. The authors intend this practice to be a standard way to convey to others viewing the image that AI colorizes it. We want to help promote this as a standard, especially as the technology continues to improve and the distinction between real and fake becomes harder to discern. This palette watermark practice was initiated and led by the MyHeritage in the MyHeritage In Color feature (which uses a newer version of DeOldify than what you're using here)."

Code

```
CMD = "wget -https://media.githubusercontent.com/media/jantic/" \  
      "DeOldify/master/resource_images/watermark.png" \  
      "-O /content/DeOldify/resource_images/watermark.png"
```

```
!{CMD}
```

7.4.2 Initialize Torch Device

First, we must initialize a Torch device. If we have a GPU available, we will detect it here. I assume that you will run this code from Google CoLab, with a GPU. It is possible to run this code from a local GPU; however, some modification will be necessary.

Code

```
import sys

#NOTE: This must be the first call in order to work properly!
from deoldify import device
from deoldify.device_id import DeviceId
#choices: CPU, GPU0...GPU7
device.set(device=DeviceId.GPU0)

import torch

if not torch.cuda.is_available():
    print('GPU not available.')
else:
    print('Using GPU.')
```

Output

Using GPU.

We can now call the model. I will enhance an image from my childhood, probably taken in the late 1970s. The picture shows three miniature schnauzers. My childhood dog (Scooby) is on the left, followed by his mom and sister. Overall, a stunning improvement. However, the red in the fire engine riding toy is lost, and the red color of the picnic table where the three dogs were sitting.

Code

```
import fastai
from deoldify.visualize import *
import warnings
from urllib.parse import urlparse
import os
```

```
warnings.filterwarnings("ignore", category=UserWarning,  
message=".*/Your.*?set is empty.*?")  
  
URL = 'https://raw.githubusercontent.com/jeffheaton/' \  
      't81_558_deep_learning/master/photos/scooby_family.jpg'  
  
!wget {URL}  
  
a = urlparse(URL)  
before_file = os.path.basename(a.path)  
  
RENDER_FACTOR = 35  
WATERMARK = False  
  
colorizer = get_image_colorizer(artistic=True)  
  
after_image = colorizer.get_transformed_image(  
    before_file, render_factor=RENDER_FACTOR,  
    watermarked=WATERMARK)  
#print("Starting image:")
```

You can see the starting image here.

Code

```
from IPython import display  
display.Image(URL)
```

Output



You can see the deoldify version here. Please note that these two images will look similar in a black and white book. To see it in color, visit this link.

Code

```
after_image
```

Output



7.5 Part 7.5: GANs for Tabular Synthetic Data Generation

Typically GANs are used to generate images. However, we can also generate tabular data from a GAN. In this part, we will use the Python tabgan utility to create fake data from tabular data. Specifically, we will use the Auto MPG dataset to train a GAN to generate fake cars. Cite:ashrapov2020tabular

7.5.1 Installing Tabgan

Pytorch is the foundation of the tabgan neural network utility. The following code installs the needed software to run tabgan in Google Colab.

Code

```
CMD = "wget https://raw.githubusercontent.com/Diyago/" \
      "GAN-for-tabular-data/master/requirements.txt"

!{CMD}
!pip install -r requirements.txt
!pip install tabgan
```

Note, after installing; you may see this message:

- You must restart the runtime in order to use newly installed versions.

If so, click the "restart runtime" button just under the message. Then rerun this notebook, and you should not receive further issues.

7.5.2 Loading the Auto MPG Data and Training a Neural Network

We will begin by generating fake data for the Auto MPG dataset we have previously seen. The tabgan library can generate categorical (textual) and continuous (numeric) data. However, it cannot generate unstructured data, such as the name of the automobile. Car names, such as "AMC Rebel SST" cannot be replicated by the GAN, because every row has a different car name; it is a textual but non-categorical value.

The following code is similar to what we have seen before. We load the AutoMPG dataset. The tabgan library requires Pandas dataframe to train. Because of this, we keep both the Pandas and Numpy values.

Code

```
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
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', '?'])

COLS_USED = ['cylinders', 'displacement', 'horsepower', 'weight',
             'acceleration', 'year', 'origin', 'mpg']
COLS_TRAIN = ['cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'year', 'origin']

df = df[COLS_USED]

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

# Split into training and test sets
df_x_train, df_x_test, df_y_train, df_y_test = train_test_split(
    df.drop("mpg", axis=1),
    df["mpg"],
    test_size=0.20,
#shuffle=False,
    random_state=42,
)

# Create dataframe versions for tabular GAN
df_x_test, df_y_test = df_x_test.reset_index(drop=True), \
    df_y_test.reset_index(drop=True)
df_y_train = pd.DataFrame(df_y_train)
df_y_test = pd.DataFrame(df_y_test)

# Pandas to Numpy
x_train = df_x_train.values
x_test = df_x_test.values
y_train = df_y_train.values
y_test = df_y_test.values

# Build the neural network
model = Sequential()
# Hidden 1

```

```

model.add(Dense(50, input_dim=x_train.shape[1], activation='relu'))
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(12, 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)

```

We now evaluate the trained neural network to see the RMSE. We will use this trained neural network to compare the accuracy between the original data and the GAN-generated data. We will later see that you can use such comparisons for anomaly detection. We can use this technique can be used for security systems. If a neural network trained on original data does not perform well on new data, then the new data may be suspect or fake.

Code

```

pred = model.predict(x_test)
score = np.sqrt(metrics.mean_squared_error(pred, y_test))
print("Final score (RMSE): {} .format(score))
```

Output

```
Final score (RMSE): 4.33633936452545
```

7.5.3 Training a GAN for Auto MPG

Next, we will train the GAN to generate fake data from the original MPG data. There are quite a few options that you can fine-tune for the GAN. The example presented here uses most of the default values. These are the usual hyperparameters that must be tuned for any model and require some experimentation for optimal results. To learn more about tabgan refer to its paper or this Medium article, written by the creator of tabgan.

Code

```

from tabgan.sampler import GANGenerator
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

```
gen_x, gen_y = GANGenerator(gen_x_times=1.1, cat_cols=None,
    bot_filter_quantile=0.001, top_filter_quantile=0.999, \
    is_post_process=True,
    adversarial_model_params={
        "metrics": "rmse", "max_depth": 2, "max_bin": 100,
        "learning_rate": 0.02, "random_state": \
        42, "n_estimators": 500,
    }, pregeneration_frac=2, only_generated_data=False, \
gan_params = {"batch_size": 500, "patience": 25, \
"epochs": 500,}).generate_data_pipe(df_x_train, df_y_train, \
df_x_test, deep_copy=True, only_adversarial=False, \
use_adversarial=True)
```

Output

```
Fitting CTGAN transformers for each column: 0%|          | 0/8
[00:00<?, ?it/s] Training CTGAN, epochs :: 0%|      | 0/500
[00:00<?, ?it/s]
```

Note: if you receive an error running the above code, you likely need to restart the runtime. You should have a "restart runtime" button in the output from the second cell. Once you restart the runtime, rerun all of the cells. This step is necessary as tabgan requires specific versions of some packages.

7.5.4 Evaluating the GAN Results

If we display the results, we can see that the GAN-generated data looks similar to the original. Some values, typically whole numbers in the original data, have fractional values in the synthetic data.

Code

```
gen_x
```

Output

	cylinders	displacement	horsepower	weight	acceleration	year	origin
0	5	296.949632	106.872450	2133	18.323035	73	2
1	5	247.744505	97.532052	2233	19.490136	75	2
2	4	259.648421	108.111921	2424	19.898952	79	3
3	5	319.208637	93.764364	2054	19.420225	78	3
4	4	386.237667	129.837418	1951	20.989091	82	2
...
542	8	304.000000	150.000000	3672	11.500000	72	1
543	8	304.000000	150.000000	3433	12.000000	70	1
544	4	98.000000	80.000000	2164	15.000000	72	1
545	4	97.500000	80.000000	2126	17.000000	72	1
546	5	138.526374	68.958515	2497	13.495784	71	1

Finally, we present the synthetic data to the previously trained neural network to see how accurately we can predict the synthetic targets. As we can see, you lose some RMSE accuracy by going to synthetic data.

Code

```
# Predict
pred = model.predict(gen_x.values)
score = np.sqrt(metrics.mean_squared_error(pred, gen_y.values))
print("Final score (RMSE): {}".format(score))
```

Output

Final score (RMSE): 9.083745225633098

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 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 private leaderboard. Your final score/rank is calculated based on this part. You do not see your private leaderboard score until the end.

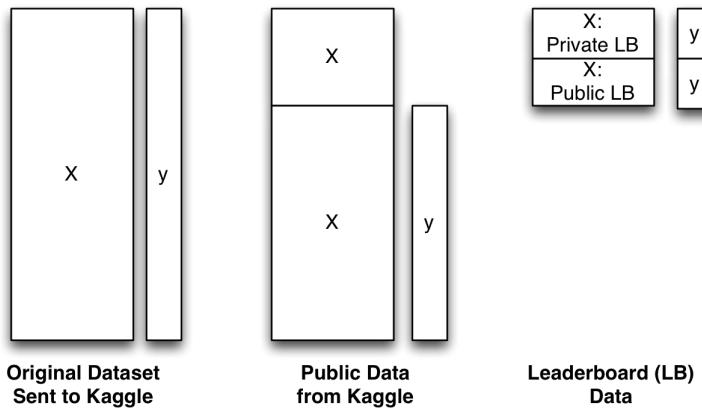


Figure 8.1: How Kaggle Competition Scoring

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 feature (from the feature/import vector) is to predicting a neural network or another model. There are many different ways to evaluate the feature importance of 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[27]. *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 chapter, we will use the input Perturbation feature ranking algorithm. This algorithm will work with any regression or classification network. In the next section, I provide an implementation of the input perturbation algorithm for scikit-learn. 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 input 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 model's accuracy. I first presented the TensorFlow implementation of this algorithm in the following paper.

- Early stabilizing feature importance for TensorFlow deep neural networks[11]

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:
```