

Learning Path

Python: Beginner's

Guide to Artificial

Intelligence

Build applications to intelligently interact with the world around you using Python



Packt

www.packt.com

Denis Rothman, Matthew Lamons, Rahul Kumar,
Abhishek Nagaraja, Amir Ziai, and Ankit Dixit

Python: Beginner's Guide to Artificial Intelligence

Build applications to intelligently interact with the world around you using Python

Denis Rothman
Matthew Lamons
Rahul Kumar
Abhishek Nagaraja
Amir Ziai
Ankit Dixit

Packt

BIRMINGHAM - MUMBAI

Python: Beginner's Guide to Artificial Intelligence

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2018

Production reference: 1211218

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78995-732-7

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customerscare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Denis Rothman graduated from l'Université Paris-Sorbonne and l'Université Paris-Diderot, writing one of the very first word2matrix embedding solutions. He began his career authoring one of the first AI cognitive NLP chatbots applied as a language teacher for Moët et Chandon and other companies. He authored an AI resource optimizer for IBM and apparel producers. He then authored an Advanced Planning and Scheduling (APS) solution used worldwide.

Matthew Lamons's background is in experimental psychology and deep learning. Founder and CEO of Skejul—the AI platform to help people manage their activities. Named by Gartner, Inc. as a "Cool Vendor" in the "Cool Vendors in Unified Communication, 2017" report. He founded The Intelligence Factory to build AI strategy, solutions, insights, and talent for enterprise clients and incubate AI tech startups based on the success of his Applied AI MasterMinds group. Matthew's global community of more than 85 K are leaders in AI, forecasting, robotics, autonomous vehicles, marketing tech, NLP, computer vision, reinforcement, and deep learning. Matthew invites you to join him on his mission to simplify the future and to build AI for good.

Rahul Kumar is an AI scientist, deep learning practitioner, and independent researcher. His expertise in building multilingual NLU systems and large-scale AI infrastructures has brought him to Copenhagen, where he leads a large team of AI engineers as Chief AI Scientist at Jatana. Often invited to speak at AI conferences, he frequently travels between India, Europe, and the US where, among other research initiatives, he collaborates with The Intelligence Factory as NLP data science fellow. Keen to explore the ramifications of emerging technologies for his next book, he's currently involved in various research projects on Quantum Computing (QC), high-performance computing (HPC), and the brain-computer interaction (BCI).

Abhishek Nagaraja was born and raised in India. Graduated Magna Cum Laude from the University of Illinois at Chicago, United States, with a Masters Degree in Mechanical Engineering with a concentration in Mechatronics and Data Science. Abhishek specializes in Keras and TensorFlow for building and evaluation of custom architectures in deep learning recommendation models. His deep learning skills and interest span computational linguistics and NLP to build chatbots to computer vision and reinforcement learning. He has been working as a Data Scientist for Skejul Inc. building an AI-powered activity forecast engine and engaged as a Deep Learning Data Scientist with The Intelligence Factory building solutions for enterprise clients.

Amir Ziai is a senior data scientist at Netflix, where he works on streaming security involving petabyte-scale machine learning platforms and applications. He has worked as a data scientist in AdTech, HealthTech, and FinTech companies. He holds a master's degree in data science from UC Berkeley.

Ankit Dixit is a deep learning expert at AIRA Matrix in Mumbai, India and having an experience of 7 years in the field of computer vision and machine learning. He is currently working on the development of full slide medical image analysis solutions in his organization. His work involves designing and implementation of various customized deep neural networks for image segmentation as well as classification tasks. He has worked with different deep neural network architectures such as VGG, ResNet, Inception, Recurrent Neural Nets (RNN) and FRCNN. He holds a masters degree in computer vision specialization. He has also authored an AI/ML book.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Become an Adaptive Thinker	8
Technical requirements	9
How to be an adaptive thinker	9
Addressing real-life issues before coding a solution	10
Step 1 – MDP in natural language	11
Step 2 – the mathematical representation of the Bellman equation and MDP	14
From MDP to the Bellman equation	14
Step 3 – implementing the solution in Python	18
The lessons of reinforcement learning	20
How to use the outputs	21
Machine learning versus traditional applications	25
Summary	26
Chapter 2: Think Like a Machine	27
Technical requirements	28
Designing datasets – where the dream stops and the hard work begins	29
Designing datasets in natural language meetings	29
Using the McCulloch-Pitts neuron	30
The McCulloch-Pitts neuron	31
The architecture of Python TensorFlow	35
Logistic activation functions and classifiers	37
Overall architecture	37
Logistic classifier	38
Logistic function	38
Softmax	40
Summary	43
Chapter 3: Apply Machine Thinking to a Human Problem	44
Technical requirements	45
Determining what and how to measure	45
Convergence	47
Implicit convergence	48
Numerical – controlled convergence	48
Applying machine thinking to a human problem	50
Evaluating a position in a chess game	50
Applying the evaluation and convergence process to a business problem	54
Using supervised learning to evaluate result quality	56
Summary	60

Chapter 4: Become an Unconventional Innovator	61
Technical requirements	62
The XOR limit of the original perceptron	62
XOR and linearly separable models	62
Linearly separable models	63
The XOR limit of a linear model, such as the original perceptron	64
Building a feedforward neural network from scratch	64
Step 1 – Defining a feedforward neural network	65
Step 2 – how two children solve the XOR problem every day	66
Implementing a vintage XOR solution in Python with an FNN and backpropagation	70
A simplified version of a cost function and gradient descent	72
Linear separability was achieved	75
Applying the FNN XOR solution to a case study to optimize subsets of data	77
Summary	83
Chapter 5: Manage the Power of Machine Learning and Deep Learning	84
Technical requirements	85
Building the architecture of an FNN with TensorFlow	85
Writing code using the data flow graph as an architectural roadmap	86
A data flow graph translated into source code	87
The input data layer	87
The hidden layer	88
The output layer	89
The cost or loss function	90
Gradient descent and backpropagation	90
Running the session	92
Checking linear separability	93
Using TensorBoard to design the architecture of your machine learning and deep learning solutions	94
Designing the architecture of the data flow graph	94
Displaying the data flow graph in TensorBoard	96
The final source code with TensorFlow and TensorBoard	96
Using TensorBoard in a corporate environment	97
Using TensorBoard to explain the concept of classifying customer products to a CEO	98
Will your views on the project survive this meeting?	98
Summary	101
Chapter 6: Focus on Optimizing Your Solutions	102
Technical requirements	103
Dataset optimization and control	103
Designing a dataset and choosing an ML/DL model	104
Approval of the design matrix	105
Agreeing on the format of the design matrix	105
Dimensionality reduction	107
The volume of a training dataset	108

Implementing a k-means clustering solution	108
The vision	109
The data	109
Conditioning management	110
The strategy	111
The k-means clustering program	111
The mathematical definition of k-means clustering	113
Lloyd's algorithm	114
The goal of k-means clustering in this case study	114
The Python program	115
1 – The training dataset	115
2 – Hyperparameters	116
3 – The k-means clustering algorithm	116
4 – Defining the result labels	117
5 – Displaying the results – data points and clusters	117
Test dataset and prediction	118
Analyzing and presenting the results	119
AGV virtual clusters as a solution	120
Summary	122
Chapter 7: When and How to Use Artificial Intelligence	123
Technical requirements	124
Checking whether AI can be avoided	124
Data volume and applying k-means clustering	125
Proving your point	126
NP-hard – the meaning of P	126
NP-hard – The meaning of non-deterministic	127
The meaning of hard	127
Random sampling	127
The law of large numbers – LLN	128
The central limit theorem	129
Using a Monte Carlo estimator	129
Random sampling applications	130
Cloud solutions – AWS	130
Preparing your baseline model	130
Training the full sample training dataset	130
Training a random sample of the training dataset	131
Shuffling as an alternative to random sampling	133
AWS – data management	135
Buckets	135
Uploading files	136
Access to output results	136
SageMaker notebook	137
Creating a job	138
Running a job	140
Reading the results	141
Recommended strategy	142
Summary	142

Chapter 8: Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies	143
Technical requirements	144
Is AI disruptive?	144
What is new and what isn't in AI	145
AI is based on mathematical theories that are not new	145
Neural networks are not new	146
Cloud server power, data volumes, and web sharing of the early 21st century started to make AI disruptive	146
Public awareness contributed to making AI disruptive	147
Inventions versus innovations	147
Revolutionary versus disruptive solutions	148
Where to start?	148
Discover a world of opportunities with Google Translate	149
Getting started	149
The program	150
The header	150
Implementing Google's translation service	151
Google Translate from a linguist's perspective	152
Playing with the tool	153
Linguistic assessment of Google Translate	153
Lexical field theory	153
Jargon	154
Translating is not just translating but interpreting	155
How to check a translation	156
AI as a new frontier	157
Lexical field and polysemy	158
Exploring the frontier – the program	160
k-nearest neighbor algorithm	161
The KNN algorithm	162
The knn_polysemy.py program	164
Implementing the KNN compressed function in Google_Translate_Customized.py	166
Conclusions on the Google Translate customized experiment	174
The disruptive revolutionary loop	175
Summary	175
Chapter 9: Getting Your Neurons to Work	177
Technical requirements	178
Defining a CNN	179
Defining a CNN	179
Initializing the CNN	181
Adding a 2D convolution	182
Kernel	182
Intuitive approach	183
Developers' approach	184
Mathematical approach	185
Shape	186
ReLU	187

Pooling	189
Next convolution and pooling layer	190
Flattening	191
Dense layers	191
Dense activation functions	192
Training a CNN model	193
The goal	193
Compiling the model	194
Loss function	194
Quadratic loss function	195
Binary cross-entropy	195
Adam optimizer	197
Metrics	197
Training dataset	198
Data augmentation	198
Loading the data	199
Testing dataset	199
Data augmentation	200
Loading the data	200
Training with the classifier	200
Saving the model	201
Next steps	202
Summary	202
Chapter 10: Applying Biomimicking to Artificial Intelligence	204
Technical requirements	205
Human biomimicking	206
TensorFlow, an open source machine learning framework	206
Does deep learning represent our brain or our mind?	207
A TensorBoard representation of our mind	209
Input data	209
Layer 1 – managing the inputs to the network	211
Weights, biases, and preactivation	212
Displaying the details of the activation function through the preactivation process	215
The activation function of Layer 1	217
Dropout and Layer 2	218
Layer 2	219
Measuring the precision of prediction of a network through accuracy values	220
Correct prediction	220
accuracy	221
Cross-entropy	223
Training	225
Optimizing speed with Google's Tensor Processing Unit	226
Summary	229
Chapter 11: Conceptual Representation Learning	231
Technical requirements	232
Generate profit with transfer learning	233
The motivation of transfer learning	233

Inductive thinking	233
Inductive abstraction	234
The problem AI needs to solve	234
The Γ gap concept	236
Loading the Keras model after training	236
Loading the model to optimize training	236
Loading the model to use it	239
Using transfer learning to be profitable or see a project stopped	242
Defining the strategy	243
Applying the model	243
Making the model profitable by using it for another problem	244
Where transfer learning ends and domain learning begins	245
Domain learning	245
How to use the programs	245
The trained models used in this section	245
The training model program	246
GAP – loaded or unloaded	246
GAP – jammed or open lanes	249
The gap dataset	251
Generalizing the Γ (gap conceptual dataset)	251
Generative adversarial networks	252
Generating conceptual representations	253
The use of autoencoders	254
The motivation of conceptual representation learning meta-models	255
The curse of dimensionality	255
The blessing of dimensionality	256
Scheduling and blockchains	256
Chatbots	257
Self-driving cars	258
Summary	258
Chapter 12: Optimizing Blockchains with AI	259
Technical requirements	260
Blockchain technology background	260
Mining bitcoins	260
Using cryptocurrency	262
Using blockchains	262
Using blockchains in the A-F network	264
Creating a block	264
Exploring the blocks	265
Using naive Bayes in a blockchain process	266
A naive Bayes example	266
The blockchain anticipation novelty	268
The goal	269
Step 1 the dataset	269
Step 2 frequency	270
Step 3 likelihood	271
Step 4 naive Bayes equation	271
Implementation	272

Gaussian naive Bayes	272
The Python program	273
Summary	275
Chapter 13: Cognitive NLP Chatbots	276
Technical requirements	277
IBM Watson	277
Intents	277
Testing the subsets	279
Entities	280
Dialog flow	282
Scripting and building up the model	283
Adding services to a chatbot	285
A cognitive chatbot service	285
The case study	286
A cognitive dataset	286
Cognitive natural language processing	287
Activating an image + word cognitive chat	289
Solving the problem	291
Implementation	292
Summary	293
Chapter 14: Improve the Emotional Intelligence Deficiencies of Chatbots	294
Technical requirements	295
Building a mind	296
How to read this chapter	297
The profiling scenario	298
Restricted Boltzmann Machines	298
The connections between visible and hidden units	299
Energy-based models	301
Gibbs random sampling	302
Running the epochs and analyzing the results	302
Sentiment analysis	304
Parsing the datasets	304
Conceptual representation learning meta-models	306
Profiling with images	306
RNN for data augmentation	308
RNNs and LSTMs	309
RNN, LSTM, and vanishing gradients	310
Prediction as data augmentation	311
Step1 – providing an input file	311
Step 2 – running an RNN	311
Step 3 – producing data augmentation	312
Word embedding	312
The Word2vec model	313
Principal component analysis	316

Intuitive explanation	317
Mathematical explanation	317
Variance	317
Covariance	319
Eigenvalues and eigenvectors	319
Creating the feature vector	321
Deriving the dataset	321
Summing it up	321
TensorBoard Projector	321
Using Jacobian matrices	322
Summary	323
Chapter 15: Building Deep Learning Environments	324
Building a common DL environment	324
Get focused and into the code!	325
DL environment setup locally	325
Downloading and installing Anaconda	326
Installing DL libraries	327
Setting up a DL environment in the cloud	328
Cloud platforms for deployment	329
Prerequisites	329
Setting up the GCP	329
Automating the setup process	330
Summary	333
Chapter 16: Training NN for Prediction Using Regression	334
Building a regression model for prediction using an MLP deep neural network	335
Exploring the MNIST dataset	336
Intuition and preparation	338
Defining regression	339
Defining the project structure	339
Let's code the implementation!	340
Defining hyperparameters	340
Model definition	341
Building the training loop	344
Overfitting and underfitting	349
Building inference	351
Concluding the project	354
Summary	355
Chapter 17: Generative Language Model for Content Creation	356
LSTM for text generation	357
Data pre-processing	358
Defining the LSTM model for text generation	359
Training the model	360
Inference and results	362

Generating lyrics using deep (multi-layer) LSTM	363
Data pre-processing	364
Defining the model	367
Training the deep TensorFlow-based LSTM model	369
Inference	372
Output	373
Generating music using a multi-layer LSTM	374
Pre-processing data	375
Defining the model and training	380
Generating music	383
Summary	385
Chapter 18: Building Speech Recognition with DeepSpeech2	387
Data preprocessing	388
Corpus exploration	389
Feature engineering	391
Data transformation	395
DS2 model description and intuition	396
Training the model	400
Testing and evaluating the model	405
Summary	406
Chapter 19: Handwritten Digits Classification Using ConvNets	407
Code implementation	408
Importing all of the dependencies	408
Exploring the data	408
Defining the hyperparameters	410
Building and training a simple deep neural network	411
Fitting a model	414
Evaluating a model	415
MLP – Python file	417
Convolution	418
Convolution in Keras	419
Fitting the model	422
Evaluating the model	422
Convolution – Python file	424
Pooling	425
Fitting the model	429
Evaluating the model	430
Convolution with pooling – Python file	432
Dropout	433
Fitting the model	435
Evaluating the model	436
Convolution with pooling – Python file	438
Going deeper	439
Compiling the model	440
Fitting the model	441

Evaluating the model	442
Convolution with pooling and Dropout – Python file	444
Data augmentation	445
Using ImageDataGenerator	446
Fitting ImageDataGenerator	448
Compiling the model	449
Fitting the model	449
Evaluating the model	450
Augmentation – Python file	452
Additional topic – convolution autoencoder	454
Importing the dependencies	455
Generating low-resolution images	456
Scaling	456
Defining the autoencoder	456
Fitting the autoencoder	459
Loss plot and test results	459
Autoencoder – Python file	462
Conclusion	464
Summary	465
Chapter 20: Object Detection Using OpenCV and TensorFlow	466
Object detection intuition	467
Improvements in object detection models	469
Object detection using OpenCV	470
A handcrafted red object detector	471
Installing dependencies	471
Exploring image data	472
Normalizing the image	474
Preparing a mask	475
Post-processing of a mask	476
Applying a mask	478
Object detection using deep learning	479
Quick implementation of object detection	479
Installing all the dependencies	479
Implementation	481
Deployment	484
Object Detection In Real-Time Using YOLOv2	487
Preparing the dataset	487
Using the pre-existing COCO dataset	488
Using the custom dataset	489
Installing all the dependencies	490
Configuring the YOLO model	491
Defining the YOLO v2 model	492
Training the model	493
Evaluating the model	498
Image segmentation	501
Importing all the dependencies	501
Exploring the data	502
Images	502

Annotations	504
Preparing the data	506
Normalizing the image	507
Encoding	508
Model data	509
Defining hyperparameters	510
Define SegNet	511
Compiling the model	514
Fitting the model	514
Testing the model	515
Conclusion	517
Summary	517
Chapter 21: Building Face Recognition Using FaceNet	518
Setup environment	520
Getting the code	520
Building the Docker image	520
Downloading pre-trained models	523
Building the pipeline	525
Preprocessing of images	527
Face detection	527
Aligning faces	528
Feature extraction	532
Execution on Docker	534
Training the classifier	534
Evaluation	535
Summary	537
Chapter 22: Generative Adversarial Networks	538
GANs	539
Implementation of GANs	541
Real data generator	543
Random data generator	545
Linear layer	546
Generator	547
Discriminator	548
GAN	549
Keep calm and train the GAN	552
GAN for 2D data generation	555
MNIST digit dataset	556
DCGAN	557
Discriminator	558
Generator	562
Transposed convolutions	563
Batch normalization	568
GAN-2D	571
Training the GAN model for 2D data generation	573

GAN Zoo	578
BiGAN – bidirectional generative adversarial networks	578
CycleGAN	579
GraspGAN – for deep robotic grasping	580
Progressive growth of GANs for improved quality	582
Summary	583
Chapter 23: From GPUs to Quantum computing - AI Hardware Computers – an ordinary tale	584
A brief history	585
Central Processing Unit	587
CPU for machine learning	590
Motherboard	590
Processor	592
Clock speed	592
Number of cores	593
Architecture	593
RAM	593
HDD and SSD	594
Operating system (OS)	594
Graphics Processing Unit (GPU)	595
GP-GPUs and NVIDIA CUDA	596
cuDNN	597
ASICs, TPUs, and FPGAs	598
ASIC	599
TPU	600
Systolic array	606
Field-programmable gate arrays	607
Quantum computers	608
Can we really build a quantum computer?	608
How far are we from the quantum era?	609
Summary	610
Chapter 24: TensorFlow Serving	612
What is TensorFlow Serving?	612
Understanding the basics of TensorFlow Serving	613
Servables	614
Servable versions	614
Models	615
Sources	615
Loaders	616
Aspired versions	616
Managers	616
Installing and running TensorFlow Serving	618
Virtual machines	619
Containers	619
Installation using Docker Toolbox	620

Table of Contents

Operations for model serving	620
Model creation	621
Saving the model	621
Serving a model	625
What is gRPC?	626
Calling the model server	627
Running the model from the client side	629
Summary	630
Other Books You May Enjoy	631
Index	634

Preface

This Learning Path offers practical knowledge and techniques you need to create and contribute to machine learning, deep learning, and modern data analysis. You will be introduced to various machine learning and deep learning algorithms from scratch, and show you how to apply them to practical industry challenges using realistic and interesting examples. You'll find a new balance of classical ideas and modern insights into machine learning. You will learn to build powerful, robust, and accurate predictive models with the power of TensorFlow, combined with other open-source Python libraries.

Throughout the Learning Path, you'll learn how to develop deep learning applications for machine learning systems using Feedforward Neural Networks, Convolutional Neural Networks, Recurrent Neural Networks, and Autoencoders. Discover how to attain deep learning programming on GPU in a distributed way.

By the end of this Learning Path, you will have understood the fundamentals of AI and worked through a number of case studies that will help you apply your skills to real-world projects.

This Learning Path includes content from the following Packt products:

- Artificial Intelligence By Example by Denis Rothman
- Python Deep Learning Projects by Matthew Lamons, Rahul Kumar, and Abhishek Nagaraja
- Hands-On Artificial Intelligence with TensorFlow by Amir Ziai, Ankit Dixit

Who this book is for

This Learning Path is for anyone who wants to understand the fundamentals of Artificial Intelligence and implement it practically by devising smart solutions. You will learn to extend your machine learning and deep learning knowledge by creating practical AI smart solutions. Prior experience with Python and statistical knowledge is essential to make the most out of this Learning Path.

What this book covers

Chapter 1, *Become an Adaptive Thinker*, covers reinforcement learning through the Bellman equation based on the Markov Decision Process (MDP). A case study describes how to solve a delivery route problem with a human driver and a self-driving vehicle.

Chapter 2, *Think like a Machine*, demonstrates neural networks starting with the McCulloch-Pitts neuron. The case study describes how to use a neural network to build the reward matrix used by the Bellman equation in a warehouse environment.

Chapter 3, *Apply Machine Thinking to a Human Problem*, shows how machine evaluation capacities have exceeded human decision-making. The case study describes a chess position and how to apply the results of an AI program to decision-making priorities.

Chapter 4, *Become an Unconventional Innovator*, is about building a feedforward neural network (FNN) from scratch to solve the XOR linear separability problem. The business case describes how to group orders for a factory.

Chapter 5, *Manage the Power of Machine Learning and Deep Learning*, uses TensorFlow and TensorBoard to build an FNN and present it in meetings.

Chapter 6, *Don't Get Lost in Techniques – Focus on Optimizing Your Solutions*, covers a Kmeans clustering program with Lloyd's algorithm and how to apply to the optimization of automatic guided vehicles in a warehouse.

Chapter 7, *When and How to Use Artificial Intelligence*, shows cloud platform machine learning solutions. We use Amazon Web Services SageMaker to solve a K-means clustering problem. The business case describes how a corporation can analyze phone call durations worldwide.

Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies*, explains the difference between a revolutionary innovation and a disruptive innovation.

Chapter 9, *Getting Your Neurons to Work*, describes convolutional neural networks (CNN) in detail: kernels, shapes, activation functions, pooling, flattening, and dense layers. The case study illustrates the use of a CNN in a food processing company.

Chapter 10, *Applying Biomimicking to Artificial Intelligence*, describes the difference between neuroscience models and deep learning solutions when representing human thinking. A TensorFlow MNIST classifier is explained component by component and displayed in detail in TensorBoard. We cover images, accuracy, cross-entropy, weights, histograms, and graphs.

Chapter 11, *Conceptual Representation Learning*, explains Conceptual Representation Learning (CRL), an innovative way to solve production flows with a CNN transformed into a CRL Meta-model. The case study shows how to use a CRLMM for transfer and domain learning, extending the model to scheduling and self-driving cars.

Chapter 12, *Optimizing Blockchains with AI*, is about mining blockchains and describes how blockchains function. We use Naive Bayes to optimize the blocks of a Supply Chain Management (SCM) blockchain by predicting transactions to anticipate storage levels.

Chapter 13, *Cognitive NLP Chatbots*, shows how to implement IBM Watson's chatbot with intents, entities, and a dialog flow. We add scripts to customize the dialogs, add sentiment analysis to give a human touch to the system, and use conceptual representation learning meta-models (CRLMMs) to enhance the dialogs.

Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*, shows how to turn a chatbot into a machine that has empathy by using a variety of algorithms at the same time to build a complex dialog. We cover Restricted Boltzmann Machines (RBMs), CRLMM, RNN, word to vector (word2Vec) embedding, and principal component analysis (PCA). A Python program illustrates an empathetic dialog between a machine and a user.

Chapter 15, *Building Deep Learning Environments*, in this chapter we will establish a common workspace for our projects with core technologies such as Ubuntu, Anaconda, Python, TensorFlow, Keras, and Google Cloud Platform (GCP).

Chapter 16, *Training NN for Prediction Using Regression*, in this chapter we will build a 2 layer (minimally deep) neural network in TensorFlow and train it on the classic MNIST dataset of handwritten digits for a restaurant patron text notification business use case.

Chapter 17, *Generative Language Model for Content Creation*, in this chapter we implement a generative model to generate content using the long short-term memory (LSTM), variational autoencoders, and Generative Adversarial Networks (GANs). You will effectively implement models for both text and music which can generate song lyrics, scripts, and music for artists and various creative businesses.

Chapter 18, *Building Speech Recognition with DeepSpeech2*, in this chapter we build and train an automatic speech recognition (ASR) system to accept then convert an audio call to text that could then be used as the input into a text-based chatbot. Work with speech and spectrograms to build an end-to-end speech recognition system with a Connectionist Temporal Classification (CTC) loss function, batch normalization and SortaGrad for the RNNs.

Chapter 19, *Handwritten Digit Classification Using ConvNets*, in this chapter we will teach the fundamentals of Convolutional Neural Networks (ConvNets) in an examination of the convolution operation, pooling, and dropout regularization. These are the levers you'll adjust in tuning your models in your career. See the value of deploying a more complex and deeper model in the performance results compared to an earlier Python Deep Learning Project in Chapter 16, *Training NN for Prediction Using Regression*.

Chapter 20, *Object Detection Using OpenCV and TensorFlow*, in this chapter we will learn to master object detection and classification while using significantly more informationally complex data than previous projects, to produce impressive outcomes. Learn to use the deep learning package YOLOv2 and gain experience how this model architecture gets deeper and more complex and produces good results.

Chapter 21, *Building Facial Recognition Using FaceNet*, in this chapter we will be using FaceNet to build a model that looks at a picture and identifies all the possible faces in it, then performs face extraction to understand the quality of the face part of the image. Performing feature extraction on the face identified parts of the image provides the basis for comparison to another data point (a labeled image of the person's face). This Python Deep Learning Project demonstrates the exciting potential for this technology in applications from social media to security.

Chapter 22, *Generative Adversarial Networks*, talks about extending the capabilities of convolutional neural networks (CNNs). This will be done by using them to create synthetic images. We will learn how to use a simple CNN to generate images. We will see various types of GANs and their applications.

Chapter 23, *From GPUs to Quantum computing – AI Hardware*, discusses the different hardware that can be used for AI application development. We will start with CPUs and extend our topic toward GPUs, ASICs, and TPUs. We will see how hardware technology has evolved along with the software technology.

Chapter 24, *TensorFlow Serving*, explains how to deploy our trained models on servers so that the majority of people can use our solutions. We will learn about TensorFlow Serving and we will deploy a very simple model on a local server.

To get the most out of this book

You should be able to install various Python packages. The code used in this book is written in Python 3.6, which will be compatible with any Python 3.X. If you are using Python 2.X then you may face some errors during execution. This book uses TensorFlow GPU version 1.8 to build neural networks. TensorFlow can only run with Python 3.X on Windows machines. This book also contains code written using the Keras API. It uses TensorFlow at the backend. We are using Keras 2.2 to build the machine learning models. I suggest you use Anaconda for Python package management: it will help you to install and uninstall packages quite easily.

For dataset-related operations, we are using pandas 0.23.0, NumPy 1.14.3, and scikit-learn 0.19.1.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Ziipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `tf.log` is applied to the results to squash them for efficient comparisons."

A block of code is set as follows:

```
W1 = tf.Variable(tf.random_uniform([2,2], -1, 1), name = "Weights1")
B1 = tf.Variable(tf.zeros([2]), name = "Bias1")
```

Any command-line input or output is written as follows:

```
docker run -p 8500:8500 \
--mount type=bind,source=$(pwd)/models/basic,target=/models/basic\
-e MODEL_NAME=BasicModel -t TensorFlow/serving &
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Then click on the **GET STARTED** option under **Python**."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Become an Adaptive Thinker

In May 2017, Google revealed AutoML, an automated machine learning system that could create an artificial intelligence solution without the assistance of a human engineer. **IBM Cloud** and **Amazon Web Services (AWS)** offer machine learning solutions that do not require AI developers. GitHub and other cloud platforms already provide thousands of machine learning programs, reducing the need of having an AI expert at hand. These cloud platforms will slowly but surely reduce the need for artificial intelligence developers. Google Cloud's AI provides intuitive machine learning services. Microsoft Azure offers user-friendly machine learning interfaces.

At the same time, **Massive Open Online Courses (MOOC)** are flourishing everywhere. Anybody anywhere can pick up a machine learning solution on GitHub, follow a MOOC without even going to college, and beat any engineer to the job.

Today, artificial intelligence is mostly mathematics translated into source code which makes it difficult to learn for traditional de

velopers. That is the main reason why Google, IBM, Amazon, Microsoft, and others have ready-made cloud solutions that will require fewer engineers in the future.

As you will see, starting with this chapter, you can occupy a central role in this new world as an adaptive thinker. There is no time to waste. In this chapter, we are going to dive quickly and directly into reinforcement learning, one of the pillars of Google Alphabet's DeepMind asset (the other being neural networks). Reinforcement learning often uses the **Markov Decision Process (MDP)**. MDP contains a memoryless and unlabeled action-reward equation with a learning parameter. This equation, the Bellman equation (often coined as the Q function), was used to beat world-class Atari gamers.

The goal here is not to simply take the easy route. We're striving to break complexity into understandable parts and confront them with reality. You are going to find out right from the start how to apply an adaptive thinker's process that will lead you from an idea to a solution in reinforcement learning, and right into the center of gravity of Google's DeepMind projects.

The following topics will be covered in this chapter:

- A three-dimensional method to implement AI, ML, and DL
- Reinforcement learning
- MDP
- Unsupervised learning
- Stochastic learning
- Memoryless learning
- The Bellman equation
- Convergence
- A Python example of reinforcement learning with the Q action-value function
- Applying reinforcement learning to a delivery example

Technical requirements

- Python 3.6x 64-bit from <https://www.python.org/>
- NumPy for Python 3.6x
- Program on Github, Chapter01_MDP.py

Check out the following video to see the code in action:

<https://goo.gl/72tSxQ>

How to be an adaptive thinker

Reinforcement learning, one of the foundations of machine learning, supposes learning through trial and error by interacting with an environment. This sounds familiar, right? That is what we humans do all our lives—in pain! Try things, evaluate, and then continue; or try something else.

In real life, you are the **agent** of your thought process. In a machine learning model, the agent is the function calculating through this trial-and-error process. This thought process in machine learning is the **MDP**. This form of action-value learning is sometimes called **Q**.

To master the outcomes of MDP in theory and practice, a three-dimensional method is a prerequisite.

The three-dimensional approach that will make you an artificial expert, in general terms, means:

- Starting by describing a problem to solve with real-life cases
- Then, building a mathematical model
- Then, write source code and/or using a cloud platform solution

It is a way for you to enter any project with an adaptive attitude from the outset.

Addressing real-life issues before coding a solution

In this chapter, we are going to tackle Markov's Decision Process (Q function) and apply it to reinforcement learning with the Bellman equation. You can find tons of source code and examples on the web. However, most of them are toy experiments that have nothing to do with real life. For example, reinforcement learning can be applied to an e-commerce business delivery person, self-driving vehicle, or a drone. You will find a program that calculates a drone delivery. However, it has many limits that need to be overcome. You as an adaptive thinker are going to ask some questions:

- What if there are 5,000 drones over a major city at the same time?
- Is a drone-jam legal? What about the noise over the city? What about tourism?
- What about the weather? Weather forecasts are difficult to make, so how is this scheduled?

In just a few minutes, you will be at the center of attention, among theoreticians who know more than you on one side and angry managers who want solutions they cannot get on the other side. Your real-life approach will solve these problems.

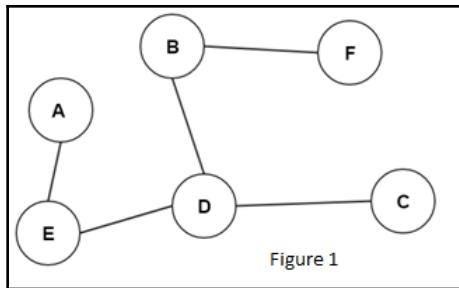
A foolproof method is the practical three-dimensional approach:

- **Be a subject matter expert (SME):** First, you have to be an SME. If a theoretician geek comes up with a hundred Google DeepMind TensorFlow functions to solve a drone trajectory problem, you now know it is going to be a tough ride if real-life parameters are taken into account.
An SME knows the subject and thus can quickly identify the critical factors of a given field. Artificial intelligence often requires finding a solution to a hard problem that even an expert in a given field cannot express mathematically. Machine learning sometimes means finding a solution to a problem that humans do not know how to explain. Deep learning, involving complex networks, solves even more difficult problems.
- **Have enough mathematical knowledge to understand AI concepts:** Once you have the proper natural language analysis, you need to build your abstract representation quickly. The best way is to look around at your everyday life and make a mathematical model of it. Mathematics is not an option in AI, but a prerequisite. The effort is worthwhile. Then, you can start writing solid source code or start implementing a cloud platform ML solution.
- **Know what source code is about as well as its potential and limits:** MDP is an excellent way to go and start working in the three dimensions that will make you adaptive: describing what is around you in detail in words, translating that into mathematical representations, and then implementing the result in your source code.

Step 1 – MDP in natural language

Step 1 of any artificial intelligence problem is to transpose it into something you know in your everyday life (work or personal). Something you are an SME in. If you have a driver's license, then you are an SME of driving. You are certified. If you do not have a driver's license or never drive, you can easily replace moving around in a car by moving around on foot.

Let's say you are an e-commerce business driver delivering a package in an area you do not know. You are the operator of a self-driving vehicle. You have a GPS system with a beautiful color map on it. The areas around you are represented by the letters A to F, as shown in the simplified map in the following diagram. You are presently at F. Your goal is to reach area C. You are happy, listening to the radio. Everything is going smoothly, and it looks like you are going to be there on time. The following graph represents the locations and routes that you can possibly cover.



The guiding system's **state** indicates the complete path to reach **C**. It is telling you that you are going to go from **F** to **B** to **D** and then to **C**. It looks good!

To break things down further, let's say:

- The present **state** is the letter *s*.
- Your next **action** is the letter *a* (*action*). This action *a* is not location **A**.
- The next action *a* (not location **A**) is to go to location **B**. You look at your guiding system; it tells you there is no traffic, and that to go from your present state **F** to your next state **B** will take you only a few minutes. Let's say that the next state **B** is the letter **B**.

At this point, you are still quite happy, and we can sum up your situation with the following sequence of events:

$$s, a, s'$$

The letter *s* is your present state, your present situation. The letter *a* is the action you're deciding, which is to go to the next area; there you will be in another state, *s'*. We can say that thanks to the action *a*, you will go from *s* to *s'*.

Now, imagine that the driver is not you anymore. You are tired for some reason. That is when a self-driving vehicle comes in handy. You set your car to autopilot. Now you are not driving anymore; the system is. Let's call that system the **agent**. At point **F**, you set your car to autopilot and let the self-driving **agent** take over.

The agent now sees what you have asked it to do and checks its mapping **environment**, which represents all the areas in the previous diagram from **A** to **F**.

In the meantime, you are rightly worried. Is the **agent** going to make it or not? You are wondering if its strategy meets yours. You have your **policy** P —your way of thinking—which is to take the shortest paths possible. Will the agent agree? What's going on in its mind? You observe and begin to realize things you never noticed before. Since this is the first time you are using this car and guiding system, the agent is **memoryless**, which is an MDP feature. This means the agent just doesn't know anything about what went on before. It seems to be happy with just calculating from this **state** s at area F. It will use machine power to run as many calculations as necessary to reach its goal.

Another thing you are watching is the total distance from F to C to check whether things are OK. That means that the agent is calculating all the states from F to C.

In this case, state F is state 1, which we can simplify by writing s_1 . B is state 2, which we can simplify by write s_2 . D is s_3 and C is s_4 . The agent is calculating all of these possible states to make a decision.

The agent knows that when it reaches D, C will be better because the reward will be higher to go to C than anywhere else. Since it cannot eat a piece of cake to reward itself, the agent uses numbers. Our agent is a real number cruncher. When it is wrong, it gets a poor reward or nothing in this model. When it's right, it gets a reward represented by the letter R. This action-value (reward) transition, often named the Q function, is the core of many reinforcement learning algorithms.

When our agent goes from one state to another, it performs a *transition* and gets a reward. For example, the *transition* can be from F to B, state 1 to state 2, or s_1 to s_2 .

You are feeling great and are going to be on time. You are beginning to understand how the machine learning agent in your self-driving car is thinking. Suddenly your guiding system breaks down. All you can see on the screen is that static image of the areas of the last calculation. You look up and see that a traffic jam is building up. Area D is still far away, and now you do not know whether it would be good to go from D to C or D to E to get a taxi that can take special lanes. You are going to need your agent!

The agent takes the traffic jam into account, is stubborn, and increases its reward to get to C by the shortest way. Its **policy** is to stick to the initial plan. You do not agree. You have another **policy**.

You stop the car. You both have to agree before continuing. You have your opinion and policy; the agent does not agree. Before continuing, your views need to **converge**.

Convergence is the key to making sure that your calculations are correct. This is the kind of problem that persons, or soon, self-driving vehicles (not to speak about drone air jams), delivering parcels encounter all day long to get the workload done. The number of parcels to delivery per hour is an example of the workload that needs to be taken into account when making a decision.

To represent the problem at this point, the best way is to express this whole process mathematically.

Step 2 – the mathematical representation of the Bellman equation and MDP

Mathematics involves a whole change in your perspective of a problem. You are going from words to functions, the pillars of source coding.

Expressing problems in mathematical notation does not mean getting lost in academic math to the point of never writing a single line of code. Mathematics is viewed in the perspective of getting a job done. Skipping mathematical representation will fast-track a few functions in the early stages of an AI project. However, when the real problems that occur in all AI projects surface, solving them with source code only will prove virtually impossible. The goal here is to pick up enough mathematics to implement a solution in real-life companies.

It is necessary to think of a problem through by finding something familiar around us, such as the delivery itinerary example covered before. It is a good thing to write it down with some abstract letters and symbols as described before, with **a** meaning an action and **s** meaning a state. Once you have understood the problem and expressed the parameters in a way you are used to, you can proceed further.

Now, mathematics well help clarify the situation by shorter descriptions. With the main ideas in mind, it is time to convert them into equations.

From MDP to the Bellman equation

In the previous step 1, the agent went from **F** or state 1 or **s** to **B**, which was state 2 or **s'**.

To do that, there was a strategy—a policy represented by **P**. All of this can be shown in one mathematical expression, the MDP state transition function:

$$Pa(s, s')$$

P is the policy, the strategy made by the agent to go from **F** to **B** through action *a*. When going from **F** to **B**, this state transition is called **state transition function**:

- *a* is the action
- *s* is state 1 (*F*) and *s'* is state 2 (*B*)

This is the basis of MDP. The reward (right or wrong) is represented in the same way:

$$Ra(s, s')$$

That means **R** is the reward for the action of going from state *s* to state *s'*. Going from one state to another will be a random process. This means that potentially, all states can go to another state.

The example we will be working on inputs a reward matrix so that the program can choose its best course of action. Then, the agent will go from state to state, learning the best trajectories for every possible starting location point. The goal of the MDP is to go to **C** (line 3, column 3 in the reward matrix), which has a starting value of 100 in the following Python code.

```
# Markov Decision Process (MDP) - The Bellman equations adapted to
# Reinforcement Learning
# R is The Reward Matrix for each state
R = ql.matrix([
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 1],
    [0, 0, 100, 1, 0, 0],
    [0, 1, 1, 0, 1, 0],
    [1, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 0]
])
```

Each line in the matrix in the example represents a letter from **A** to **F**, and each column represents a letter from **A** to **F**. All possible states are represented. The 1 values represent the nodes (vertices) of the graph. Those are the possible locations. For example, line 1 represents the possible moves for letter **A**, line 2 for letter **B**, and line 6 for letter **F**. On the first line, **A** cannot go to **C** directly, so a 0 value is entered. But, it can go to **E**, so a 1 value is added.

Some models start with -1 for impossible choices, such as **B** going directly to **C** and 0 values to define the locations. This model starts with 0 and 1 values. It sometimes takes weeks to design functions that will create a reward matrix (see Chapter 2, *Think like a Machine*).

There are several properties of this decision process. A few of them are mentioned here:

- **The Markov property:** The process is applied when the past is not taken into account. It is the memoryless property of this decision process, just as you do in a car with a guiding system. You move forward to reach your goal. This is called the Markov property.
- **Unsupervised learning:** From this memoryless Markov property, it is safe to say that the MDP is not supervised learning. Supervised learning would mean that we would have all the labels of the trip. We would know exactly what **A** means and use that property to make a decision. We would be in the future looking at the past. MDP does not take these labels into account. This means that this is unsupervised learning. A decision has to be made in each state without knowing the past states or what they signify. It means that the car, for example, was on its own at each location, which is represented by each of its states.
- **Stochastic process:** In step 1, when state **B** was reached, the agent controlling the mapping system and the driver didn't agree on where to go. A random choice could be made in a trial-and-error way, just like a coin toss. It is going to be a heads-or-tails process. The agent will toss the coin thousands of times and measure the outcomes. That's precisely how MDP works and how the agent will learn.
- **Reinforcement learning:** Repeating a trial and error process with feedback from the agent's environment.
- **Markov chain:** The process of going from state to state with no history in a random, stochastic way is called a **Markov chain**.

To sum it up, we have three tools:

- $P_a(s,s')$: A **policy**, **P**, or strategy to move from one state to another
- $T_a(s,s')$: A **T**, or stochastic (random) **transition**, function to carry out that action
- $R_a(s,s')$: An **R**, or **reward**, for that action, which can be negative, null, or positive

T is the transition function, which makes the **agent** decide to go from one point to another with a policy. In this case, it will be random. That's what machine power is for, and that's how reinforcement learning is often implemented.



Randomness is a property of MDP.

The following code describes the choice the *agent* is going to make.

```
next_action = int(ql.random.choice(PossibleAction, 1))  
return next_action
```

Once the code has been run, a new random action (state) has been chosen.



The Bellman equation is the road to programming reinforcement learning.

Bellman's equation completes the MDP. To calculate the value of a state, let's use Q , for the Q action-reward (or value) function. The pre-source code of Bellman's equation can be expressed as follows for one individual state:

$$Q(s) = R(s) + \gamma * \max(s')$$

The source code then translates the equation into a machine representation as in the following code:

```
# The Bellman equation  
Q[current_state, action] = R[current_state, action] + gamma * MaxValue
```

The source code variables of the Bellman equation are as follows:

- $Q(s)$: This is the value calculated for this state—the total reward. In step 1 when the agent went from F to B, the driver had to be happy. Maybe she/he had a crunch in a candy bar to feel good, which is the human counterpart of the reward matrix. The automatic driver maybe ate (reward matrix) some electricity, renewable energy of course! The reward is a number such as 50 or 100 to show the agent that it's on the right track. It's like when a student gets a good grade in an exam.
- $R(s)$: This is the sum of the values up to there. It's the total reward at that point.

- $\gamma = \text{gamma}$: This is here to remind us that trial and error has a price. We're wasting time, money, and energy. Furthermore, we don't even know whether the next step is right or wrong since we're in a trial-and-error mode. **Gamma** is often set to 0.8. What does that mean? Suppose you're taking an exam. You study and study, but you don't really know the outcome. You might have 80 out of 100 (0.8) chances of clearing it. That's painful, but that's life. This is what makes Bellman's equation and MDP realistic and efficient.
- $\max(s')$: s' is one of the possible states that can be reached with $P_a(s, s')$; max is the highest value on the line of that state (location line in the reward matrix).

Step 3 – implementing the solution in Python

In step 1, a problem was described in natural language to be able to talk to experts and understand what was expected. In step 2, an essential mathematical bridge was built between natural language and source coding. Step 3 is the software implementation phase.

When a problem comes up—and rest assured that one always does—it will be possible to go back over the mathematical bridge with the customer or company team, and even further back to the natural language process if necessary.

This method guarantees success for any project. The code in this chapter is in Python 3.6. It is a reinforcement learning program using the Q function with the following reward matrix:

```
import numpy as ql
R = ql.matrix([
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 1],
    [0, 0, 100, 1, 0, 0],
    [0, 1, 1, 0, 1, 0],
    [1, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 0]
])

Q = ql.matrix(ql.zeros([6, 6]))

gamma = 0.8
```

R is the reward matrix described in the mathematical analysis.

Q inherits the same structure as R , but all values are set to 0 since this is a learning matrix. It will progressively contain the results of the decision process. The gamma variable is a double reminder that the system is learning and that its decisions have only an 80% chance of being correct each time. As the following code shows, the system explores the possible actions during the process.

```
agent_s_state = 1

# The possible "a" actions when the agent is in a given state
def possible_actions(state):
    current_state_row = R[state,]
    possible_act = ql.where(current_state_row >0) [1]
    return possible_act

# Get available actions in the current state
PossibleAction = possible_actions(agent_s_state)
```

The agent starts in state 1, for example. You can start wherever you want because it's a random process. Note that only values > 0 are taken into account. They represent the possible moves (decisions).

The current state goes through an analysis process to find possible actions (next possible states). You will note that there is no algorithm in the traditional sense with many rules. It's a pure random calculation, as the following `random.choice` function shows.

```
def ActionChoice(available_actions_range):
    next_action = int(ql.random.choice(PossibleAction, 1))
    return next_action

# Sample next action to be performed
action = ActionChoice(PossibleAction)
```

Now comes the core of the system containing Bellman's equation, translated into the following source code:

```
def reward(current_state, action, gamma):
    Max_State = ql.where(Q[action,] == ql.max(Q[action,])) [1]

    if Max_State.shape[0] > 1:
        Max_State = int(ql.random.choice(Max_State, size = 1))
    else:
        Max_State = int(Max_State)
    MaxValue = Q[action, Max_State]
```

```
# Q function
Q[current_state, action] = R[current_state, action] + gamma * MaxValue

# Rewarding Q matrix
reward(agent_s_state,action,gamma)
```

You can see that the agent looks for the maximum value of the next possible state chosen at random.

The best way to understand this is to run the program in your Python environment and `print()` the intermediate values. I suggest that you open a spreadsheet and note the values. It will give you a clear view of the process.

The last part is simply about running the learning process 50,000 times, just to be sure that the system learns everything there is to find. During each iteration, the agent will detect its present state, choose a course of action, and update the `Q` function matrix:

```
for i in range(50000):
    current_state = ql.random.randint(0, int(Q.shape[0]))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state,action,gamma)
    # Displaying Q before the norm of Q phase
    print("Q :")
    print(Q)

    # Norm of Q
    print("Normed Q :")
    print(Q/ql.max(Q)*100)
```

After the process is repeated and until the learning process is over, the program will print the result in `Q` and the normed result. The normed result is the process of dividing all values by the sum of the values found. The result comes out as a normed percentage.

View the Python program at <https://github.com/PacktPublishing/Artificial-Intelligence-By-Example/blob/master/Chapter01/MDP.py>.

The lessons of reinforcement learning

Unsupervised reinforcement machine learning, such as MDP and Bellman's equation, will topple traditional decision-making software in the next few years. Memoryless reinforcement learning requires few to no business rules and thus doesn't require human knowledge to run.

Being an adaptive AI thinker involves three requisites—the effort to be an SME, working on mathematical models, and understanding source code's potential and limits:

- **Lesson 1:** Machine learning through reinforcement learning can beat human intelligence in many cases. No use fighting! The technology and solutions are already here.
- **Lesson 2:** Machine learning has no emotions, but you do. And so do the people around you. Human emotions and teamwork are an essential asset. Become an SME for your team. Learn how to understand what they're trying to say intuitively and make a mathematical representation of it for them. This job will never go away, even if you're setting up solutions such as Google's AutoML that don't require much development.

Reinforcement learning shows that no human can solve a problem the way a machine does; 50,000 iterations with random searching is not an option. The days of neuroscience imitating humans are over. Cheap, powerful computers have all the leisure it takes to compute millions of possibilities and choose the best trajectories.

Humans need to be more intuitive, make a few decisions, and see what happens because humans cannot try 50,000 ways of doing something. Reinforcement learning marks a new era for human thinking by surpassing human reasoning power.

On the other hand, reinforcement learning requires mathematical models to function. Humans excel in mathematical abstraction, providing powerful intellectual fuel to those powerful machines.

The boundaries between humans and machines have changed. Humans' ability to build mathematical models and every-growing cloud platforms will serve online machine learning services.

Finding out how to use the outputs of the reinforcement learning program we just studied shows how a human will always remain at the center of artificial intelligence.

How to use the outputs

The reinforcement program we studied contains no trace of a specific field, as in traditional software. The program contains Bellman's equation with stochastic (random) choices based on the reward matrix. The goal is to find a route to C (line 3, column 3), which has an attractive reward (100):

```
# Markov Decision Process (MDP) - Bellman's equations adapted to
# Reinforcement Learning with the Q action-value(reward) matrix
```

```
# R is The Reward Matrix for each state
R = ql.matrix([
    [0,0,0,0,1,0],
    [0,0,0,1,0,1],
    [0,0,100,1,0,0],
    [0,1,1,0,1,0],
    [1,0,0,1,0,0],
    [0,1,0,0,0,0]
])
```

That reward matrix goes through Bellman's equation and produces a result in Python:

```
Q :
[[ 0. 0. 0. 0. 258.44 0. ]
 [ 0. 0. 0. 321.8 0. 207.752]
 [ 0. 0. 500. 321.8 0. 0. ]
 [ 0. 258.44 401. 0. 258.44 0. ]
 [ 207.752 0. 0. 321.8 0. 0. ]
 [ 0. 258.44 0. 0. 0. 0. ]]

Normed Q :
[[ 0. 0. 0. 0. 51.688 0. ]
 [ 0. 0. 0. 64.36 0. 41.5504]
 [ 0. 0. 100. 64.36 0. 0. ]
 [ 0. 51.688 80.2 0. 51.688 0. ]
 [ 41.5504 0. 0. 64.36 0. 0. ]
 [ 0. 51.688 0. 0. 0. 0. ]]
```

The result contains the values of each state produced by the reinforced learning process, and also a normed Q (highest value divided by other values).

As Python geeks, we are overjoyed. We made something rather difficult to work, namely reinforcement learning. As mathematical amateurs, we are elated. We know what MDP and Bellman's equation mean.

However, as natural language thinkers, we have made little progress. No customer or user can read that data and make sense of it. Furthermore, we cannot explain how we implemented an intelligent version of his/her job in the machine. We didn't.

We hardly dare say that reinforcement learning can beat anybody in the company making random choices 50,000 times until the right answer came up.

Furthermore, we got the program to work but hardly know what to do with the result ourselves. The consultant on the project cannot help because of the matrix format of the solution.

Being an adaptive thinker means knowing how to be good in all the dimensions of a subject. To solve this new problem, let's go back to step 1 with the result.

By formatting the result in Python, a graphics tool, or a spreadsheet, the result that is displayed as follows:

	A	B	C	D	E	F
A	-	-	-	-	258.44	-
B	-	-	-	321.8	-	207.752
C	-	-	500	321.8	-	-
D	-	258.44	401.	-	258.44	-
E	207.752	-	-	321.8	-	-
F	-	258.44	-	-	-	-

Now, we can start reading the solution:

- Choose a starting state. Take **F** for example.
- The **F** line represents the state. Since the maximum value is 258.44 in the **B** column, we go to state **B**, the second line.
- The maximum value in state **B** in the second line leads us to the **D** state in the fourth column.
- The highest maximum of the **D** state (fourth line) leads us to the **C** state.

Note that if you start at the **C** state and decide not to stay at **C**, the **D** state becomes the maximum value, which will lead you back to **C**. However, the MDP will never do this naturally. You will have to force the system to do it.

You have now obtained a sequence: **F->B->D->C**. By choosing other points of departure, you can obtain other sequences by simply sorting the table.

The most useful way of putting it remains the normalized version in percentages. This reflects the stochastic (random) property of the solution, which produces probabilities and not certainties, as shown in the following matrix:

	A	B	C	D	E	F
A	-	-	-	-	51.68%	-
B	-	-	-	64.36%	-	41.55%
C	-	-	100%	64.36%	-	-
D	-	51.68%	80.2%	-	51.68%	-
E	41.55%	-	-	64.36%	-	-
F	-	51.68%	-	-	-	-

Now comes the very tricky part. We started the chapter with a trip on a road. But I made no mention of it in the result analysis.

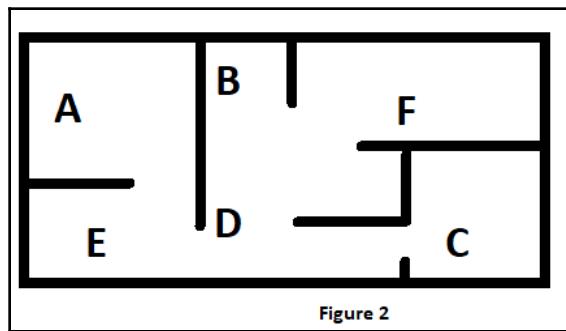
An important property of reinforcement learning comes from the fact that we are working with a mathematical model that can be applied to anything. No human rules are needed. This means we can use this program for many other subjects without writing thousands of lines of code.

Case 1: Optimizing a delivery for a driver, human or not

This model was described in this chapter.

Case 2: Optimizing warehouse flows

The same reward matrix can apply to going from point F to C in a warehouse, as shown in the following diagram:



In this warehouse, the F->B->D->C sequence makes visual sense. If somebody goes from point F to C, then this physical path makes sense without going through walls.

It can be used for a video game, a factory, or any form of layout.

Case 3: Automated planning and scheduling (APS)

By converting the system into a scheduling vector, the whole scenery changes. We have left the more comfortable world of physical processing of letters, faces, and trips. Though fantastic, those applications are social media's tip of the iceberg. The real challenge of artificial intelligence begins in the abstract universe of human thinking.

Every single company, person, or system requires automatic planning and scheduling. The six A to F steps in the example of this chapter could well be six tasks to perform in a given unknown order represented by the following vector x :

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

The reward matrix then reflects the weights of constraints of the tasks of vector x to perform. For example, in a factory, you cannot assemble the parts of a product before manufacturing them.

In this case, the sequence obtained represents the schedule of the manufacturing process.

Case 4 and more: Your imagination

By using physical layouts or abstract decision-making vectors, matrices, and tensors, you can build a world of solutions in a mathematical reinforcement learning model. Naturally, the following chapters will enhance your toolbox with many other concepts.

Machine learning versus traditional applications

Reinforcement learning based on stochastic (random) processes will evolve beyond traditional approaches. In the past, we would sit down and listen to future users to understand their way of thinking.

We would then go back to our keyboard and try to imitate the human way of thinking. Those days are over. We need proper datasets and ML/DL equations to move forward. Applied mathematics has taken reinforcement learning to the next level. Traditional software will soon be in the museum of computer science.

An artificial adaptive thinker sees the world through applied mathematics translated into machine representations.



Use the Python source code example provided in this chapter in different ways. Run it; try to change some parameters to see what happens. Play around with the number of iterations as well. Lower the number from 50,000 down to where you find its best. Change the reward matrix a little to see what happens. Design your own reward matrix trajectory. It can be an itinerary or a decision-making process.

Summary

Presently, artificial intelligence is predominantly a branch of applied mathematics, not of neurosciences. You must master the basics of linear algebra and probabilities. That's a difficult task for a developer used to intuitive creativity. With that knowledge, you will see that humans cannot rival with machines that have CPU and mathematical functions. You will also understand that machines, contrary to the hype around you, don't have emotions although we can represent them to a scary point (See Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*) in chatbots.

That being said, a multi-dimensional approach is a requisite in an AI/ML/DL project—first talk and write about the project, then make a mathematical representation, and finally go for software production (setting up an existing platform and/or writing code). In real-life, AI solutions do not just grow spontaneously in companies like trees. You need to talk to the teams and work with them. That part is the real fulfilling aspect of a project—imagining it first and then implementing it with a group of real-life people.

MDP, a stochastic random action-reward (value) system enhanced by Bellman's equation, will provide effective solutions to many AI problems. These mathematical tools fit perfectly in corporate environments.

Reinforcement learning using the Q action-value function is memoryless (no past) and unsupervised (the data is not labeled or classified). This provides endless avenues to solve real-life problems without spending hours trying to invent rules to make a system work.

Now that you are at the heart of Google's DeepMind approach, it is time to go to Chapter 2, *Think Like a Machine*, and discover how to create the reward matrix in the first place through explanations and source code.

2

Think Like a Machine

The first chapter described a reinforcement learning algorithm through the Q action-value function used by DQN. The agent was a driver. You are at the heart of DeepMind's approach to AI.

DeepMind is no doubt one of the world leaders in applied artificial intelligence. Scientific, mathematical, and applications research drives its strategy.

DeepMind was founded in 2010, was acquired by Google in 2014, and is now part of Alphabet, a collection of companies that includes Google.

One of the focuses of DeepMind is on reinforcement learning. They came up with an innovative version of reinforcement learning called **DQN** and referring to deep neural networks using the Q function (Bellman's equation). A seminal article published in February 2015 in *Nature* (see the link at the end of the chapter) shows how DQN outperformed other artificial intelligence research by becoming a human game tester itself. DQN then went on to beat human game testers.

In this chapter, the agent will be an **automated guided vehicle (AGV)**. An AGV takes over the transport tasks in a warehouse. This case study opens promising perspectives for jobs and businesses using DQN. Thousands upon thousands of warehouses require complex reinforcement learning and customized transport optimization.

This chapter focuses on creating the **reward matrix**, which was the entry point of the Python example in the first chapter. To do so, it describes how to add a primitive McCulloch-Pitts neuron in TensorFlow to create an intelligent adaptive network and add an N (network) to a Q model. It's a small N that will become a feedforward neural network in Chapter 4, *Become an Unconventional Innovator*. The goal is not to copy DQN but to use the conceptual power of the model to build a variety of solutions.

The challenge in this chapter will be to think literally like a machine. The effort is not to imitate human thinking but to beat humans with machines. This chapter will take you very far from human reasoning into the depth of machine thinking.

The following topics will be covered in this chapter:

- AGV
- The McCulloch-Pitts neuron
- Creating a reward matrix
- Logistic classifiers
- The logistic sigmoid
- The softmax function
- The one-hot function
- How to apply machine learning tools to real-life problems such as warehouse management

Technical requirements

- Python 3.6x 64-bit from <https://www.python.org/>
- NumPy for Python 3.6x
- TensorFlow from <https://deepmind.com/> with TensorBoard

The following files:

- <https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter02/MCP.py>
- <https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter02/SOFTMAX.py>

Check out the following video to see the code in action:

<https://goo.gl/jMWLg8>

Designing datasets – where the dream stops and the hard work begins

As in the previous chapter, bear in mind that a real-life project goes through a three-dimensional method in some form or the other. First, it's important to just think and talk about the problem to solve without jumping onto a laptop. Once that is done, bear in mind that the foundation of machine learning and deep learning relies on mathematics. Finally, once the problem has been discussed and mathematically represented, it is time to develop the solution.



First, think of a problem in natural language. Then, make a mathematical description of a problem. Only then, start the software implementation.

Designing datasets in natural language meetings

The reinforcement learning program described in the first chapter can solve a variety of problems involving unlabeled classification in an unsupervised decision-making process. The Q function can be applied indifferently to drone, truck, or car deliveries. It can also be applied to decision-making in games or real life.

However, in a real-life case study problem (such as defining the reward matrix in a warehouse for the AGV, for example), the difficulty will be to design a matrix that everybody agrees with.

This means many meetings with the IT department to obtain data, the SME and reinforcement learning experts. An AGV requires information coming from different sources: daily forecasts and real-time warehouse flows.

At one point, the project will be at a standstill. It is simply too complicated to get the right data for the reinforcement program. This is a real-life case study that I modified a little for confidentiality reasons.

The warehouse manages thousands of locations and hundreds of thousands of inputs and outputs. The Q function does not satisfy the requirement in itself. A small neural network is required.

In the end, through tough negotiations with both the IT department and the users, a dataset format is designed that fits the needs of the reinforcement learning program and has enough properties to satisfy the AGV.

Using the McCulloch-Pitts neuron

The mathematical aspect relies on finding a model for inputs for huge volumes in a corporate warehouse.

In one mode, the inputs can be described as follows:

- Thousands of forecast product arrivals with a low priority weight: $w1 = 10$
- Thousands of confirmed arrivals with a high priority weight: $w2 = 70$
- Thousands of unplanned arrivals decided by the sales department: $w3 = 75$
- Thousands of forecasts with a high priority weight: $w4 = 60$
- Thousands of confirmed arrivals that have a low turnover and so have a low weight: $w5 = 20$

These weights represent vector w :

$$x = \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

All of these products have to be stored in optimal locations, and the distance between nearly 100 docks and thousands of locations in the warehouse for the AGV has to be minimized.

Let's focus on our neuron. Only these weights will be used, though a system such as this one will add up to more than 50 weights and parameters per neuron.

In the first chapter, the reward matrix was size 6x6. Six locations were described (A to F), and now six locations (l1 to l6) will be represented in a warehouse.

A 6x6 reward matrix represents the target of the McCulloch-Pitts layer implemented for the six locations.

Also, this matrix was the input in the first chapter. In real life, and in real companies, you will have to find a way to build datasets from scratch. The reward matrix becomes the output of this part of the process. The following source code shows the input of the reinforcement learning program used in the first chapter. The goal of this chapter describes how to produce the following reward matrix.

```
# R is The Reward Matrix for each location in a warehouse (or any other problem)

R = ql.matrix([
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 1],
    [0, 0, 100, 1, 0, 0],
    [0, 1, 1, 0, 1, 0],
    [1, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 0]
])
```

For this warehouse problem, the McCulloch-Pitts neuron sums up the weights of the priority vector described previously to fill in the reward matrix.

Each location will require its neuron, with its weights.

INPUTS – > WEIGHTS – BIAS – > VALUES

- Inputs are the flows in a warehouse or any form of data
- Weights will be defined in this model
- Bias is for stabilizing the weights
- Values will be the output

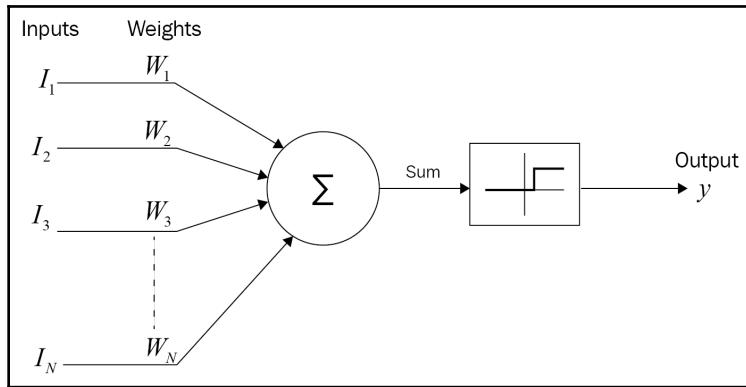


There are as many ways as you can imagine to create reward matrices. This chapter describes one way of doing it that works.

The McCulloch-Pitts neuron

The McCulloch-Pitts neuron dates back to 1943. It contains inputs, weights, and an activation function. This is precisely where you need to think like a machine and forget about human neuroscience brain approaches for this type of problem. Starting from Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations Small to Large Companies*, human cognition will be built on top of these models, but the foundations need to remain mathematical.

The following diagram shows the McCulloch-Pitts, neuron model.



This model contains a number of input x weights that are summed to either reach a threshold which will lead, once transformed, to $y = 0$, or 1 output. In this model, y will be calculated in a more complex way.

A Python-TensorFlow program, `MCP.py` will be used to illustrate the neuron.

When designing neurons, the computing performance needs to be taken into account. The following source code configures the threads. You can fine-tune your model according to your needs.

```
config = tf.ConfigProto(
    inter_op_parallelism_threads=4,
    intra_op_parallelism_threads=4
)
```

In the following source code, the placeholders that will contain the input values (`x`), the weights (`w`), and the bias (`b`) are initialized. A placeholder is not just a variable that you can declare and use later when necessary. It represents the structure of your graph:

```
x = tf.placeholder(tf.float32, shape=(1, 6), name='x')
w = tf.placeholder(tf.float32, shape=(6, 1), name='w')
b = tf.placeholder(tf.float32, shape=(1), name='b')
```

In the original McCulloch-Pitts artificial neuron, the inputs (`x`) were multiplied by the following weights:

$$w_1 x_1 + \dots + w_n x_n = \sum_{j=1}^n w_j x_j$$

The mathematical function becomes a one-line code with a logistic activation function (sigmoid), which will be explained in the second part of the chapter. Bias (b) has been added, which makes this neuron format useful even today shown as follows.

```
y = tf.matmul(x, w) + b
s = tf.nn.sigmoid(y)
```

Before starting a session, the McCulloch-Pitts neuron (1943) needs an operator to directly set its weights. That is the main difference between the McCulloch-Pitts neuron and the perceptron (1957), which is the model of modern deep learning neurons. The perceptron optimizes its weights through optimizing processes. Chapter 4, *Become an Unconventional Innovator*, describes the modern perceptron.

The weights are now provided, and so are the quantities for each x stored at l_1 , one of the locations of the warehouse:

$$x = \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \\ w5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

The weight values will be divided by 100, to represent percentages in terms of 0 to 1 values of warehouse flows in a given location. The following code deals with the choice of **one** location, l_1 **only**, its values, and parameters.

```
with tf.Session(config=config) as tfs:
    tfs.run(tf.global_variables_initializer())
    w_t = [[.1, .7, .75, .6, .2]]
    x_1 = [[10, 2, 1., 6., 2.]]
    b_1 = [1]
    w_1 = np.transpose(w_t)
    value = tfs.run(s,
                    feed_dict={
                        x: x_1,
                        w: w_1,
                        b: b_1
                    })
    print ('value for threshold calculation',value)
```

The session starts; the weights (w_t) and the quantities (x_1) of the warehouse flow are entered. Bias is set to 1 in this model. w_1 is transposed to fit x_1 . The placeholders are solicited with `feed_dict`, and the value of the neuron is calculated using the sigmoid function.

The program returns the following value.

```
print ('value for threshold calculation',value)
      value for threshold calculation [[ 0.99971133]]
```

*This value represents the activity of location l_1 at a given date and a given time. The higher the value, the higher the probable saturation rate of this area. That means there is little space left for an AGV that would like to store products. That is why the reinforcement learning program for a warehouse is looking for the **least loaded** area for a given product in this model.*

Each location has a probable **availability**:

$$A = \text{Availability} = 1 - \text{load}$$

The probability of a load of a given storage point lies between 0 and 1.

High values of availability will be close to 1, and low probabilities will be close to 0 as shown in the following example:

```
>>>print ('Availability of lx',1-value)
      Availability of lx [[ 0.00028867]]
```

For example, the load of l_1 has a probable load of 0.99 and its probable *availability* is 0.002. The goal of the AGV is to search and find the closest and most available location to optimize its trajectories. l_1 is obviously not a good candidate at that day and time. **Load** is a keyword in production activities.

When all of the six locations' availabilities has been calculated by the McCulloch-Pitts neuron—each with its respective x quantity inputs, weights, and bias—a location vector of the results of this system will be produced. This means that the program needs to be implemented to run all six locations and not just one location:

$$\mathbf{A}(L) = \{a(l_1), a(l_2), a(l_3), a(l_4), a(l_5), a(l_6)\}$$

The availability ($1 - \text{output value of the neuron}$) constitutes a six-line vector. The following vector will be obtained by running the previous sample code on **all** six locations.

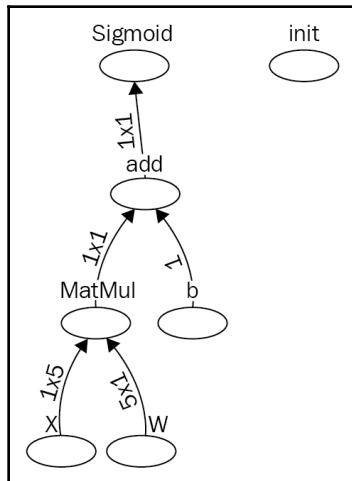
$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix}$$

lv is the vector containing the value of each location for a given AGV to choose from. The values in the vector represent availability. 0.0002 means little availability. 0.9 means high availability. Once the choice is made, the reinforcement learning program presented in the first chapter will optimize the AGV's trajectory to get to this specific warehouse location.

The lv is the result of the weighing function of six potential locations for the AGV. It is also a vector of transformed inputs.

The architecture of Python TensorFlow

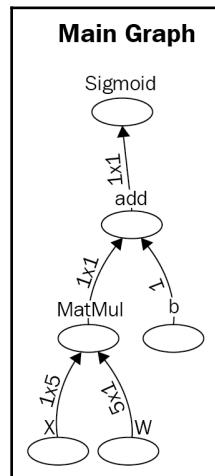
Implementation of the McCulloch-Pitts neuron can best be viewed with TensorBoard, as shown in the following graph:



This is obtained by adding the following TensorBoard code at the end of your session. This data flow graph will help optimize a program when things go wrong.

```
# _____Tensorboard_____  
  
#with tf.Session() as sess:  
  
Writer = tf.summary.FileWriter("directory on your machine", tfs.graph)  
Writer.close()  
  
  
def launchTensorBoard():  
    import os  
    #os.system('tensorboard --logdir=' + 'your directory')  
    os.system('tensorboard --logdir=' + 'your directory')  
    return  
  
import threading  
t = threading.Thread(target=launchTensorBoard, args=[])  
t.start()  
  
tfs.close()  
#Open your browser and go to http://localhost:6006  
#Try the various options. It is a very useful tool.  
#close the system window when you're finished.
```

When you open the URL indicated in the code on your machine, you will see the following TensorBoard data flow graph:



Logistic activation functions and classifiers

Now that the value of each location of $L=\{l_1, l_2, l_3, l_4, l_5, l_6\}$ contains its availability in a vector, the locations can be sorted from the most available to least available location. From there, the reward matrix for the MDP process described in the first chapter can be built.

Overall architecture

At this point, the overall architecture contains two main components:

- **Chapter 1: Become an Adaptive Thinker:** A reinforcement learning program based on the value-action Q function using a reward matrix that is yet to be calculated. The reward matrix was given in the first chapter, but in real life, you'll often have to build it from scratch. This could take weeks to obtain.
- **Chapter 2:** A set of six neurons that represent the flow of products at a given time at six locations. The output is the availability probability from 0 to 1. The highest value is the highest availability. The lowest value is the lowest availability.

At this point, there is some real-life information we can draw from these two main functions:

- An AGV is moving in a warehouse and is waiting to receive its next location to use an MDP, in order to calculate an optimal trajectory of its mission, as shown in the first chapter.
- An AGV is using a reward matrix that was given in the first chapter, but it needs to be designed in a real-life project through meetings, reports, and acceptance of the process.
- A system of six neurons, one per location, weighing the real quantities and probable quantities to give an availability vector lv has been calculated. It is almost ready to provide the necessary reward matrix for the AGV.

To calculate the input values of the reward matrix in this reinforcement learning warehouse model, a bridge function between lv and the reward matrix R is missing.

That bridge function is a logistic classifier based on the outputs of the y neurons.

At this point, the system:

- Took corporate data
- Used y neurons calculated with weights
- Applied an activation function

The activation function in this model requires a logistic classifier, a commonly used one.

Logistic classifier

The logistic classifier will be applied to lv (the six location values) to find the best location for the AGV. It is based on the output of the six neurons ($input \times weight + bias$).

What are logistic functions? The goal of a logistic classifier is to produce a probability distribution from 0 to 1 for each value of the output vector. As you have seen so far, AI applications use applied mathematics with probable values, not raw outputs. In the warehouse model, the AGV needs to choose the best, most probable location, l_i . Even in a well-organized corporate warehouse, many uncertainties (late arrivals, product defects, or a number of unplanned problems) reduce the probability of a choice. A probability represents a value between 0 (low probability) and 1 (high probability). Logistic functions provide the tools to convert all numbers into probabilities between 0 and 1 to *normalize* data.

Logistic function

The logistic sigmoid provides one of the best ways to normalize the weight of a given output. This will be used as the activation function of the neuron. The threshold is usually a value above which the neuron has a $y=1$ value; or else it has $y=0$. In this case, the minimum value will be 0 because the activation function will be more complex.

The logistic function is represented as follows.

$$\frac{1}{1 + e^{-x}}$$

- e represents Euler's number, or 2.71828, the natural logarithm.
- x is the value to be calculated. In this case, x is the result of the logistic sigmoid function.

The code has been rearranged in the following example to show the reasoning process:

```
#For given variables:
x_1 = [[10, 2, 1., 6., 2.]]      # the x inputs
w_t = [[.1, .7, .75, .60, .20]]  # the corresponding weights
b_1 = [1]                          # the bias
# A given total weight y is calculated
y = tf.matmul(x, w) + b
```

```
# then the logistic sigmoid is applied to y which represents the "x" in the
# formal definition of the Logistic Sigmoid
s = tf.nn.sigmoid(y)
```

Thanks to the logistic sigmoid function, the value for the first location in the model comes out as 0.99 (level of saturation of the location).

To calculate the availability of the location once the 0.99 has been taken into account, we subtract the load from the total availability, which is 1, as follows:

As seen previously, once all locations are calculated in this manner, a final availability vector, lv , is obtained.

$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow [?]$$

When analyzing lv , a problem has stopped the process. Individually, each line appears to be fine. By applying the logistic sigmoid to each output weight and subtracting it from 1, each location displays a probable availability between 0 and 1. However, the sum of the lines in lv exceeds 1. That is not possible. Probability cannot exceed 1. The program needs to fix that. In the source code, lv will be named y .

Each line produces a [0,1] solution, which fits the prerequisite of being a valid probability.

In this case, the vector lv contains more than one value and becomes a multiple distribution. The sum of lv cannot exceed 1 and needs to be normalized.

The *softmax* function provides an excellent method to stabilize lv . *Softmax* is widely used in machine learning and deep learning.

Bear in mind that these *mathematical tools are not rules*. You can adapt them to your problem as much as you wish as long as your solution works.

Softmax

The softmax function appears in many artificial intelligence models to normalize data. This is a fundamental function to understand and master. In the case of the warehouse model, an AGV needs to make a probable choice between six locations in the lv vector. However, the total of the lv values exceeds 1. lv requires normalization of the softmax function S . In this sense, the softmax function can be considered as a generalization of the logistic sigmoid function. In the code, lv vector will be named y .

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^n e^{y_j}}$$

The following code used is SOFTMAX.py; y represents the lv vector in the following source code.

```
# y is the vector of the scores of the lv vector in the warehouse example:  
y = [0.0002, 0.2, 0.9, 0.0001, 0.4, 0.6]
```

e^{y_i} is the $\exp(i)$ result of each value in y (lv in the warehouse example), as follows:

```
y_exp = [math.exp(i) for i in y]
```

$\sum_{j=1}^n e^{y_j}$ is the sum of e^{y_i} iterations, as shown in the following code:

```
sum_exp_yi = sum(y_exp)
```

Now, each value of the vector can be normalized in this type of multinomial distribution stabilization by simply applying a division, as follows:

```
softmax = [round(i / sum_exp_yi, 3) for i in y_exp]  
  
#Vector to be stabilized [2.0, 1.0, 0.1, 5.0, 6.0, 7.0]  
#Stabilized vector [0.004, 0.002, 0.001, 0.089, 0.243, 0.661]
```

$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow softmax(lv) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix}$$

$\text{softmax}(lv)$ provides a normalized vector with a sum equal to 1 and is shown in this compressed version of the code. The vector obtained is often described as containing **logits**.

The following code details the process:

```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)

y1 = [0.0002, 0.2, 0.9, 0.0001, 0.4, 0.6]
print("Stabilized vector", softmax(y1))
print("sum of vector", sum(softmax(y1)))
# Stabilized vector [ 0.11119203  0.13578309  0.27343357  0.11118091
0.16584584  0.20256457]
# sum of vector 1.0
```

The softmax function can be used as the output of a classifier (pixels for example) or to make a decision. In this warehouse case, it transforms lv into a decision-making process.

The last part of the softmax function requires $\text{softmax}(lv)$ to be rounded to 0 or 1. The higher the value in $\text{softmax}(lv)$, the more probable it will be. In clear-cut transformations, the highest value will be close to 1 and the others will be closer to 0. In a decision-making process, the highest value needs to be found, as follows:

```
print("highest value in transformed y vector", max(softmax(y1)))
#highest value in normalized y vector 0.273433565194
```

Once line 3 (value 0.273) has been chosen as the most probable location, it is set to 1 and the other, lower values are set to 0. This is called a **one-hot** function. This **one-hot** function is extremely helpful to encode the data provided. The vector obtained can now be applied to the reward matrix. The value 1 probability will become 100 in the R reward matrix, as follows.

$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow \text{softmax}(lv) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix} \rightarrow \text{one-hot} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow R \rightarrow \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The softmax function is now complete. Location l_3 or C is the best solution for the AGV. The probability value is multiplied by 100 in the R function and the reward matrix described can now receive the input.

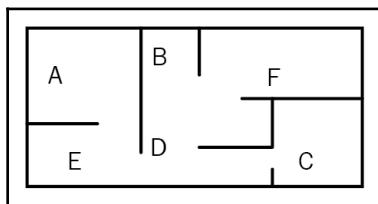


Before continuing, take some time to play around with the values in the source code and run it to become familiar with softmax.

We now have the data for the reward matrix. The best way to understand the mathematical aspect of the project is to go to a paperboard and draw the result using the actual warehouse layout from locations A to F.

$$\text{Locations} = \{l_1\text{-A}, l_2\text{-B}, l_3\text{-C}, l_4\text{-D}, l_5\text{-E}, l_6\text{-F}\}$$

Value of locations in the reward matrix = {0,0,100,0,0,0} where C (the third value) is now the target for the self-driving vehicle, in this case, an AGV in a warehouse.



We obtain the following reward matrix R described in the first chapter.

State/values	A	B	C	D	E	F
A	-	-	-	-	1	-
B	-	-	-	1	-	1
C	-	-	100	1	-	-
D	-	1	1	-	1	-
E	1	-	-	1	-	-
F	-	1	-	-	-	-

This reward matrix is exactly the one used in the Python reinforcement learning program using the Q function in the first chapter. The output of this chapter is the input of the R matrix in the first chapter. The 0 values are there for the agent to avoid those values. This program is designed to stay close to probability standards with positive values, as shown in the following R matrix.

```
R = ql.matrix([
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 1],
    [0, 0, 100, 1, 0, 0],
```

```
[0, 1, 1, 0, 1, 0],  
[1, 0, 0, 1, 0, 0],  
[0, 1, 0, 0, 0, 0] ])
```

At this point, the building blocks are in place to begin evaluating the results of the reinforcement learning program.

Summary

Using a McCulloch-Pitts neuron with a logistic activation function in a one-layer network to build a reward matrix for reinforcement learning shows how to build real-life applications with AI technology.

Processing real-life data often requires a generalization of a logistic sigmoid function through a softmax function, and a one-hot function applied to logits to encode the data.

This shows that machine learning functions are tools that must be understood to be able to use all or parts of them to solve a problem. With this practical approach to artificial intelligence, a whole world of projects awaits you.

You can already use these first two chapters to present powerful trajectory models such as Amazon warehouses and deliveries to your team or customers. Furthermore, Amazon, Google, Facebook, Netflix, and many others are growing their data centers as we speak. Each data center has locations with data flows that need to be calibrated. You can use the ideas given in this chapter to represent the problems and real-time calculations required to calibrate product and data flows.

This neuronal approach is the parent of the multi-layer perceptron that will be introduced in Chapter 5, *Manage The Power of Machine Learning and Deep Learning*. There, a shift from machine learning to deep learning will be made.

However, before that, machine learning or deep learning requires evaluation functions. No result can be validated without evaluation, as explained in Chapter 3, *Apply Machine Thinking to a Human Problem*. In the next chapter, the evaluation process will be illustrated with chess and a real-life situation.

3

Apply Machine Thinking to a Human Problem

In the first chapter, the MDP reinforcement program produced a result as an output matrix. In Chapter 2, *Think Like a Machine*, the McCulloch-Pitts system of neurons produced an *input reward matrix*. However, the intermediate or final results of these two functions need to be constantly measured. Good measurement solves a substantial part of a given problem since decisions rely on them. Reliable decisions are made with reliable evaluations. The goal of this chapter is to introduce measurement methods.

The key function of human intelligence, decision-making, relies on the ability to evaluate a situation. No decision can be made without measuring the pros and cons and factoring the parameters.

Mankind takes great pride in its ability to evaluate. However, in many cases, a machine can do better. Chess represents the pride of mankind in thinking strategy. A chessboard is often present in many movies to symbolize human intelligence.

Today, not a single chess player can beat the best chess engines. One of the extraordinary core capacities of a chess engine is the evaluation function; it takes many parameters into account more precisely than humans.

This chapter focuses on the main concepts of evaluation and measurement; they set the path to deep learning gradient descent-driven models, which will be explained in the following chapter.

The following topics will be covered in this chapter:

- Evaluation of the episodes of a learning session
- Numerical convergence measurements
- An introduction to the idea of cross-entropy convergence
- Decision tree supervised learning as an evaluation method

- Decision tree supervised learning as a predictive model
- How to apply evaluation tools to a real-life problem you build on your own

Technical requirements

- Python version 3.6 is recommended
- NumPy compatible with Python 3.6
- TensorFlow with TensorBoard
- Graphviz 2.28 for use in Python

Programs are available on GitHub, Chapter03:

- Q_learning_convergence.py
- Decision_Tree_Priority_classifier.py

Check out the following video to see the code in action:

<https://goo.gl/Yrgb3i>

Determining what and how to measure

In Chapter 2, *Think Like a Machine*, the system of McCulloch-Pitts neurons generated a vector with a one-hot function in the following process.

$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow \text{softmax}(lv) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix} \rightarrow \text{one-hot} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow R \rightarrow \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

R, the reward vector, represents the input of the reinforcement learning program and needs to be measured.

This chapter deals with an approach designed to build a reward matrix based on the company data. It relies on the data, weights, and biases provided. When deep learning forward feedback neural networks based on perception are introduced (Chapter 4, *Become an Unconventional Innovator*), a system cannot be content with a training set. Systems have a natural tendency to learn training sets through backpropagation. In this case, one set of company data is not enough.

In real-life company projects, a system will not be validated until tens of thousands of results have been produced. In some cases, a corporation will approve the system only after hundreds of datasets with millions of data samples have been tested to be sure that all scenarios are accurate. Each dataset represents a scenario consultants can work on with parameter scripts. The consultant introduces parameter scenarios that are tested by the system and measured. In systems with up to 200 parameters per neuron, a consultant will remain necessary for many years to come in an industrial environment. As of Chapter 4, *Become an Unconventional Innovator*, the system will be on its own without the help of a consultant. Even then, consultants often are needed to manage the hyperparameters. In real-life systems, with high financial stakes, quality control will always remain essential.

Measurement should thus apply to generalization more than simply applying to a single or few datasets. Otherwise, you will have a natural tendency to control the parameters and overfit your model in a too-good-to-be-true scenario.

Beyond the reward matrix, the reinforcement program in the first chapter had a learning parameter $\lambda = 0.8$, as shown in the following code source.

```
# Gamma : It's a form of penalty or uncertainty for learning  
# If the value is 1 , the rewards would be too high.  
# This way the system knows it is learning.  
gamma = 0.8
```

The λ learning parameter in itself needs to be closely monitored because it introduces uncertainty into the system. This means that the learning process will always remain a probability, never a certainty. One might wonder why this parameter is not just taken out. Paradoxically, that will lead to even more global uncertainty. The more the λ learning parameter tends to 1, the more you risk overfitting your results. **Overfitting** means that you are pushing the system to think it's learning well when it isn't. It's exactly like a teacher who gives high grades to everyone in the class all the time. The teacher would be overfitting the grade-student evaluation process, and nobody would know whether the students have learned something.

The results of the reinforcement program need to be measured as they go through episodes. The range of the learning process itself must be measured. In the following code, the range is set to 50,000 to make sure the learning process reaches its goal.

```
for i in range(50000):
    current_state = ql.random.randint(0, int(Q.shape[0]))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state, action, gamma)
```

All of these measurements will have a deep effect on the results obtained.

Convergence

Building the system was fun. Finding the factors that make the system go wrong is another story.

The model presented so far can be summed up as follows:

$$lv = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix} \rightarrow softmax(lv) \rightarrow \begin{bmatrix} 0.111 \\ 0.135 \\ 0.273 \\ 0.111 \\ 0.165 \\ 0.202 \end{bmatrix} \rightarrow one-hot \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow R \rightarrow \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow gamma \rightarrow Q \rightarrow Results$$

From *lv* to *R*, the process creates the reward matrix (Chapter 2, *Think Like a Machine*) required for the reinforcement learning program (Chapter 1, *Become an Adaptive Thinker*), which runs from reading *R* (reward matrix) to the results. Gamma is the learning parameter, *Q* is the *Q* learning function, and the results are the states of *Q* described in the first chapter.

The parameters to be measured are as follows:

- The company's input data. The training sets found on the Web such as MNIST are designed to be efficient. These ready-made datasets often contain some noise (unreliable data) to make them realistic. The same process must be achieved with raw company data. *The only problem is that you cannot download a corporate dataset from somewhere. You have to build the datasets.*
- The weights and biases that will be applied.
- The activation function (a logistic function or other).

- The choices to make after the one-hot process.
- The learning parameter.
- Episode management through convergence.

The best way to start relies on measuring the quality of convergence of the system, the last step of the whole process.

If the system provides good convergence, it will avoid the headache of having to go back and check everything.

Implicit convergence

In the last part of `Reinforcement_Learning_Q_function.py` in the first chapter, a range of 50,000 is implemented.

The idea is to set the number of episodes at such a level that convergence is certain. In the following code, the range (50000) is a constant.

```
for i in range(50000):
    current_state = ql.random.randint(0, int(Q.shape[0]))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state, action, gamma)
```

Convergence, in this case, will be defined as the point at which no matter how long you run the system, the Q result matrix will not change anymore.

By setting the range to 50000, you can test and verify this. As long as the reward matrices remain homogeneous, this will work. If the reward matrices strongly vary from one scenario to another, this model will produce unstable results.



Try to run the program with different ranges. Lower the ranges until you see that the results are not optimal.

Numerical – controlled convergence

This approach can prove time-saving by using the target result, provided it exists beforehand. Training the reinforcement program in this manner validates the process.

In the following source code, an intuitive *cross-entropy* function is introduced (see Chapter 9, *Getting Your Neurons to Work*, for more on cross-entropy).

Cross-entropy refers to energy. The main concepts are as follows:

- Energy represents the difference between one distribution and another
- It is what makes the system continue to train
- When a lot of training needs to be done, there is a **high level of energy**
- When the training reaches the end of its cycles, **the level of energy is low**
- In the following code, **cross-entropy value (CEV)** measures the **difference between a target matrix and the episode matrix**
- Cross-entropy is often measured in more complex forms when necessary (see Chapter 9, *Getting Your Neurons to Work*, and Chapter 10, *Applying Biomimicking to Artificial Intelligence*)

In the following code, a basic function provides sufficient results.

```
for i in range(50000):
    current_state = ql.random.randint(0, int(Q.shape[0]))
    PossibleAction = possible_actions(current_state)
    action = ActionChoice(PossibleAction)
    reward(current_state,action,gamma)
    if Q.sum()>0:
        #print("convergent episode:",i,"Q.Sum",Q.sum(),"numerical convergent
value e-1:",Q.sum()-sum)
        #print("convergent episode:",i,"numerical convergent value:",ceg-
Q.sum())
        CEV=-(math.log(Q.sum())-math.log(ceg))
        print("convergent episode:",i,"numerical convergent value:",CEV)
        sum=Q.sum()
        if(Q.sum()-3992==0):
            print("Final convergent episode:",i,"numerical convergent
value:",ceg-Q.sum())
            break; #break on average (the process is random) before 50000
```

The previous program stops before 50,000 epochs. This is because, in the model described in this chapter (see the previous code excerpt), the system stops when it reaches an acceptable CEV convergence value.

```
convergent episode: 1573 numerical convergent value: -0.0
convergent episode: 1574 numerical convergent value: -0.0
convergent episode: 1575 numerical convergent value: -0.0
convergent episode: 1576 numerical convergent value: -0.0
convergent episode: 1577 numerical convergent value: -0.0
Final convergent episode: 1577 numerical convergent value: 0.0
```

The program stopped at episode 1577. Since the decision process is random, the same number will not be obtained twice in a row. Furthermore, the constant 3992 was known in advance. This is possible in closed environments where a pre-set goal has been set. This is not the case often but was used to illustrate the concept of convergence. The following chapters will explore better ways to reach convergence, such as gradient descent.

The Python program is available at:

https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter03/Q_learning_convergence.py

Applying machine thinking to a human problem

"An efficient manager has a high evaluation quotient. A machine has a better one, in chess and a number of increasing fields. The problem now is to keep up with what the machines are learning!"

-Denis Rothman

Evaluation is one of the major keys to efficient decision making in all fields: from chess, production management, rocket launching, and self-driving cars to data center calibration, software development, and airport schedules. Chess engines are not high-level deep-learning-based software. They rely heavily on evaluations and calculations. They evaluate much better than humans, and there is a lot to learn from them. The question now is to know whether any human can beat a chess engine or not. The answer is no.

To evaluate a position in chess, you need to examine all the pieces, their quantitative value, their qualitative value, cooperation between pieces, who owns each of the 64 squares, the king's safety, bishop pairs, knight positioning, and many other factors.

Evaluating a position in a chess game

Evaluating a position in a chess game shows why machines will surpass humans in quite some decision-making fields within the next few years.

The following position is after move 23 in the Kramnik-Bluebaum 2017 game. It cannot be correctly evaluated by humans. It contains too many parameters to analyze and too many possibilities.



It is white's turn to play, and a close analysis shows that both players are lost at this point. In a tournament like this, they must each continue to keep a poker face. They often look at the position with a confident face to hide their dismay. Some even shorten their thinking time to make their opponent think they know where they are going.

These unsolvable positions for humans are painless to solve with chess engines, even cheap, high-quality chess engines on a smartphone. This can be generalized to all human activity that has become increasingly complex, unpredictable and chaotic. Decision-makers will increasingly rely on artificial intelligence to help them make the right choices.

No human can play chess and evaluate the way a chess engine does by simply calculating the positions of the pieces, their squares of liberty, and many other parameters. A chess engine generates an evaluation matrix with millions of calculations. The following table is the result of an evaluation of only one position among many others (real and potential).

Position evaluated	0,3					
White	34					
	Initial position	position	Value		Quality Value	TotalValue
Pawn	a2	a2	1	a2-b2 small pawn island	0,05	1,05
Pawn	b2	b2	1	a2-b2 small pawn island	0,05	1,05

Pawn	c2	x	0	captured	0	0
Pawn	d2	d4	1	occupies center, defends Be5	0,25	1,25
Pawn	e2	e2	1	defends Qf3	0,25	1,25
Pawn	f2	x	0	captured	0	0
Pawn	g2	g5	1	unattacked, attacking 2 squares	0,3	1,3
Pawn	h2	h3	1	unattacked, defending g4	0,1	1,1
Rook	a1	c1	5	occupying c-file, attacking b7 with Nd5-Be5	1	6
Knight	b1	d5	3	attacking Nb6, 8 squares	0,5	3,5
BishopDS	c1	e5	3	central position, 10 squares, attacking c7	0,5	3,5
Queen	d1	f3	9	battery with Bg2, defending Ne5, X-Ray b7	2	11
King	e1	h1	0	X-rayed by Bb6 on a7-g1 diagonal	-0,5	-0,5
BishopWS	f1	g2	3	supporting Qf3 in defense and attack	0,5	3,5
Knight	g1	x	0	captured	0	0
Rook	h1	x	0	captured	0	0
			29		5	34
						White:34

The value of the position of white is 34.

White	34					
Black	33,7					
	Initial position	position	Value		Quality Value	TotalValue
Pawn	a7	a7	1	a7-b7 small pawn island	0,05	1,05
Pawn	b7	b7	1	a7-b7 small pawn island	0,05	1,05
Pawn	c7	x	0	captured	0	0
Pawn	d7	x	0	captured	0	0
Pawn	e7	f5	1	doubled, 2 squares	0	1
Pawn	f7	f7	1		0	1
Pawn	g7	g6	1	defending f5 but abandoning Kg8	0	1
Pawn	h7	h5	1	well advanced with f5,g6	0,1	1,1
Rook	a8	d8	5	semi-open d-file attacking Nd5	2	7
Knight	b8	x	0	captured	0	0
BishopDS	c8	b6	3	attacking d4, 3 squares	0,5	3,5
Queen	d8	e6	9	attacking d4,e5 , a bit cramped	1,5	10,5
King	e8	g8	0	f6,h6, g7,h8 attacked	-1	-1

BishopWS	f8	x	0	captured, White lost bishop pair	0,5	0,5
Knight	g8	e8	3	defending c7,f6,g7	1	4
Rook	h8	f8	5	out of play	-2	3
			31		2,7	Black:33,7

The value of black is 33.7.

So white is winning by $34 - 33.7 = 0.3$.

The evaluation system can easily be represented with two McCulloch-Pitts neurons, one for black and one for white. Each neuron would have 30 weights = $\{w_1, w_2, \dots, w_{30}\}$, as shown in the previous table. The sum of both neurons requires an activation function that converts the evaluation into 1/100th of a pawn, which is the standard measurement unit in chess. Each weight will be the output of squares and piece calculations. Then the MDP can be applied to Bellman's equation with a random generator of possible positions.



Present-day chess engines contain barely more intelligence than this type of pure calculation approach. They don't need more to beat humans.

No human, not even world champions, can calculate this position with this accuracy. The number of parameters to take into account overwhelms them each time they reach a position like this. They then play more or less randomly with some kind of idea in mind. It resembles a lottery sometimes. Chess expert annotators discover this when they run human-played games with powerful chess engines to check the game. The players themselves now tend to reveal their incapacity when questioned.

Now bear in mind that the position analyzed represents only one possibility. A chess engine will test millions of possibilities. Humans can test only a few.

Measuring a result like this has nothing to do with natural human thinking. Only machines can think like that. Not only do chess engines solve the problem, but also they are impossible to beat.



At one point, there are problems humans face that only machines can solve.

Applying the evaluation and convergence process to a business problem

What was once considered in chess as the ultimate proof of human intelligence has been battered by brute-force calculations with great CPU/RAM capacity. Almost any human problem requiring logic and reasoning can most probably be solved by a machine using relatively elementary processes expressed in mathematical terms.

Let's take the result matrix of the reinforcement learning example of the first chapter. It can also be viewed as a scheduling tool. Automated planning and scheduling have become a crucial artificial intelligence field. In this case, evaluating and measuring the result goes beyond convergence aspects.

In a scheduling process, the input of the reward matrix can represent the priorities of the packaging operation of some products in a warehouse. It would determine in which order customer products must be picked to be packaged and delivered. These priorities extend to the use of a machine that will automatically package the products in a FIFO mode (first in, first out). The systems provide good solutions, but, in real life, many unforeseen events change the order of flows in a warehouse and practically all schedules.

In this case, the result matrix can be transformed into a vector of a scheduled packaging sequence. The packaging department will follow the priorities produced by the system.

The reward matrix (see `Q_learning_convergence.py`) in this chapter is `R` (see the following code).

```
R = ql.matrix([ [-1,-1,-1,-1,0,-1],  
[-1,-1,-1,0,-1,0],  
[-1,-1,100,0,-1,-1],  
[-1,0,100,-1,0,-1],  
[0,-1,-1,0,-1,-1],  
[-1,0,-1,-1,-1,-1] ])
```

Its visual representation is the same as in [Chapter 1, Become an Adaptive Thinker](#). But the values are a bit different for this application:

- **Negative values (-1):** The agent cannot go there
- **0 values:** The agent can go there
- **100 values:** The agent should favor these locations

The result is produced in a Q function early in the first section of the chapter, in a matrix format, displayed as follows:

```

Q :
[[ 0. 0. 0. 0. 258.44 0. ]
 [ 0. 0. 0. 321.8 0. 207.752]
 [ 0. 0. 500. 321.8 0. 0. ]
 [ 0. 258.44 401. 0. 258.44 0. ]
 [ 207.752 0. 0. 321.8 0. 0. ]
 [ 0. 258.44 0. 0. 0. 0. ]]

Normed Q :
[[ 0. 0. 0. 0. 51.688 0. ]
 [ 0. 0. 0. 64.36 0. 41.5504]
 [ 0. 0. 100. 64.36 0. 0. ]
 [ 0. 51.688 80.2 0. 51.688 0. ]
 [ 41.5504 0. 0. 64.36 0. 0. ]
 [ 0. 51.688 0. 0. 0. 0. ]]

```

From that result, the following packaging priority order matrix can be deduced.

Priorities	O1	O2	O3	O4	O5	O6
O1	-	-	-	-	258.44	-
O2	-	-	-	321.8	-	207.75
O3	-	-	500	321.8	-	-
O4	-	258.44	401	-	258.44	-
O5	207.75	-	-	321.8	-	-
O6	-	258.44	-	-	-	-

The **non-prioritized vector (npv)** of packaging orders is np .

$$npv = \begin{bmatrix} O1 \\ O2 \\ O3 \\ O4 \\ O5 \\ O6 \end{bmatrix}$$

The npv contains the priority value of each cell in the matrix, which is not a location but an order priority. Combining this vector with the result matrix, the results become priorities of the packaging machine. They now need to be analyzed, and a final order must be decided to send to the packaging department.

Using supervised learning to evaluate result quality

Having now obtained the *npv*, a more business-like measurement must be implemented.

A warehouse manager, for example, will tell you the following:

- Your reinforcement learning program looks satisfactory (*Chapter 1, Become an Adaptive Thinker*)
- The reward matrix generated by the McCulloch-Pitts neurons works very well (*Chapter 2, Think Like a Machine*)
- The convergence values of the system look nice
- The results on this dataset look satisfactory

But then, the manager will always come up with a killer question, *How can you prove that this will work with other datasets in the future?*

The only way to be sure that this whole system works is to run thousands of datasets with hundreds of thousands of product flows.

The idea now is to use supervised learning to create relationships between the input and output data. It's not a random process like MDP. They are not trajectories anymore. They are priorities. One method is to used decision trees. In *Chapter 4, Become an Unconventional Innovator*, the problem will be solved with a feedforward backpropagation network.

In this model, the properties of the customer orders are analyzed so that we can classify them. This can be translated into decision trees depending on real-time data, to create a distribution representation to predict future outcomes.

1. The first step is to represent the properties of the orders O1 to O6.

```
features = [ 'Priority/location', 'Volume', 'Flow_optimizer' ]
```

In this case, we will limit the model to three properties:

- Priority/location, which is the most important property in a warehouse flow in this model
- Volumes to transport
- Optimizing priority—the financial and customer satisfaction property

2. The second step is to provide some priority parameters to the learning dataset:

```
Y = ['Low', 'Low', 'High', 'High', 'Low', 'Low']
```

3. Step 3 is providing the dataset input matrix, which is the output matrix of the reinforcement learning program. The values have been approximated but are enough to run the model. This simulates some of the intermediate decisions and transformations that occur during the decision process (ratios applied, uncertainty factors added, and other parameters). The input matrix is X:

```
X = [[256, 1, 0],  
     [320, 1, 0],  
     [500, 1, 1],  
     [400, 1, 1],  
     [320, 1, 0],  
     [256, 1, 0]]
```

The features in step 1 apply to each column.

The values in step 2 apply to every line.

4. Step 4 is running a standard decision tree classifier. This classifier will distribute the representations (distributed representations) into two categories:

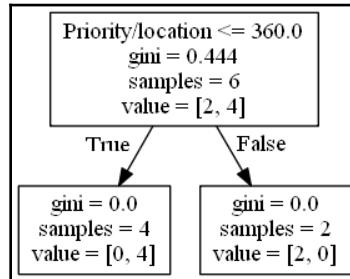
- The properties of high-priority orders
- The properties of low-priority orders

There are many types of algorithms. In this case, a standard `sklearn` function is called to do the job, as shown in the following source code.

```
classify = tree.DecisionTreeClassifier()  
classify = classify.fit(X, Y)
```

Applied to thousands of orders on a given day, it will help adapt to real-time unplanned events that destabilize all scheduling systems: late trucks, bad quality products, robot breakdowns, and absent personnel. This means that the system must be able to constantly adapt to new situations and provide priorities to replan in real time.

The program will produce the following graph, which separates the orders into priority groups.



The goal now is to separate the best orders to replan among hundreds of thousands of simulating orders. In this case, the learning dataset has the six values you have been studying in the first two chapters from various angles.

- Priority/location ≤ 360.0 is the division point between the most probable optimized orders (high) and less interesting ones (low).
- Gini impurity. This would be the measure of incorrect labeling if the choice were random. In this case, the dataset is stable.
- The false arrow points out the two values that are not ≤ 360 , meaning they are good choices, the optimal separation line of the representation. The ones that are not classified as *False* are considered as *don't eliminate* orders.
The *True* elements mean: *eliminate orders as long as possible*.
- The value result reads as *[number of false elements, number of true elements]* of the dataset.

If you play around with the values in steps 1, 2, and 3, you'll obtain different separation points and values.

You can use this part of the source code to generate images of this decision tree-supervised learning program:

```

# 5. Producing visualization if necessary
info =
tree.export_graphviz(classify, feature_names=features, out_file=None, filled=False,
                     rounded=False)
graph = pydotplus.graph_from_dot_data(info)

edges = collections.defaultdict(list)
for edge in graph.get_edge_list():
    edges[edge.get_source()].append(int(edge.get_destination()))
  
```

```
for edge in edges:  
    edges[edge].sort()  
    for i in range(2):  
        dest = graph.get_node(str(edges[edge][i]))[0]  
  
graph.write_png('warehouse_example_decision_tree.png')  
print("Open the image to verify that the priority level prediction of the  
results fits the reality of the reward matrix inputs")
```

The preceding information represents a small part of what it takes to manage a real-life artificial intelligence program on premise.

A warehouse manager will want to run this supervised learning decision tree program on top of the system described in Chapter 2, *Think Like a Machine*, and Chapter 3, *Apply Machine Thinking to a Human Problem*. This is done to generalize these distributed representations directly to the warehouse data to improve the initial corporate data inputs. With better-proven priorities, the system will constantly improve, week by week.

This way of scheduling shows that human thinking was not used nor necessary.

Contrary to the hype surrounding artificial intelligence, most problems can be solved with no human intelligence involved and relatively little machine learning technology.

Human intelligence simply proves that intelligence can solve a problem.



Fortunately for the community of artificial intelligence experts, there are very difficult problems to solve that require more artificial intelligence thinking.

Such a problem will be presented and solved in Chapter 4, *Become an Unconventional Innovator*.

The Python program is available at https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter03/Decision_Tree_Priority_classifier.py

Summary

This chapter led artificial intelligence exploration one more step away from neuroscience to reproduce human thinking. Solving a problem like a machine means using a chain of mathematical functions and properties.

The further you get in machine learning and deep learning, the more you will find mathematical functions that solve the core problems. Contrary to the astounding amount of hype, mathematics relying on CPUs is replacing humans, not some form of alien intelligence.

The power of machine learning with *beyond-human* mathematical reasoning is that generalization to other fields is easier. A mathematical model, contrary to the complexity of humans entangled in emotions, makes it easier to deploy the same model in many fields. The models of the first three chapters can be used for self-driving vehicles, drones, robots in a warehouse, scheduling priorities, and much more. Try to imagine as many fields you can apply these to as possible.

Evaluation and measurement are at the core of machine learning and deep learning. The key factor is constantly monitoring convergence between the results the system produces and the goal it must attain. This opens the door to the constant adaptation of the weights of the network to reach its objectives.

Machine evaluation for convergence through a chess example that has nothing to do with human thinking proves the limits of human intelligence. The decision tree example can beat most humans in classification situations where large amounts of data are involved.

Human intelligence is not being reproduced in many cases and has often been surpassed. In those cases, human intelligence just proves that intelligence can solve a problem, nothing more.

The next chapter goes a step further from human reasoning with self-weighting neural networks and introduces deep learning.

4

Become an Unconventional Innovator

In corporate projects, there always comes the point when a problem that seems impossible to solve hits you. At that point, you try everything you learned, but it doesn't work for what's asked of you. Your team or customer begins to look elsewhere. It's time to react.

In this chapter, an impossible-to-solve business case regarding material optimization will be implemented successfully with an example of a **feedforward neural network (FNN)** with backpropagation.

Feedforward networks are the building blocks of deep learning. The battle around the XOR function perfectly illustrates how deep learning regained popularity in corporate environments. The XOR FNN illustrates one of the critical functions of neural networks: **classification**. Once information becomes classified into subsets, it opens the doors to **prediction** and many other functions of neural networks, such as representation learning.

An XOR FNN network will be built from scratch to demystify deep learning from the start. A vintage, start-from-scratch method will be applied, blowing the deep learning hype off the table.

The following topics will be covered in this chapter:

- How to hand build an FNN
- Solving XOR with an FNN
- Classification
- Backpropagation
- A cost function

- Cost function optimization
- Error loss
- Convergence

Technical requirements

- Python 3.6x 64-bit from <https://www.python.org/>
- NumPy for Python 3.6x

Programs from GitHub Chapter04:

- FNN_XOR_vintage_tribute.py
- FFN_XOR_generalization.py

Check out the following video to see the code in action:

<https://goo.gl/ASyLWz>

The XOR limit of the original perceptron

Once the feedforward network for solving the XOR problem is built, it will be applied to a material optimization business case. The material-optimizing solution will choose the best combinations of dimensions among billions to minimize the use of a material with the generalization of the XOR function. First, a solution to the XOR limitation of a perceptron must be fully clarified.

XOR and linearly separable models

In the academic world, like the private world, competition exists. Such a situation took place in 1969. Minsky and Papert published *Perceptrons*. They proved mathematically that a perceptron could *not* solve an XOR function. Fortunately, today the perceptron and its neocognitron version form the core model for neural networking.

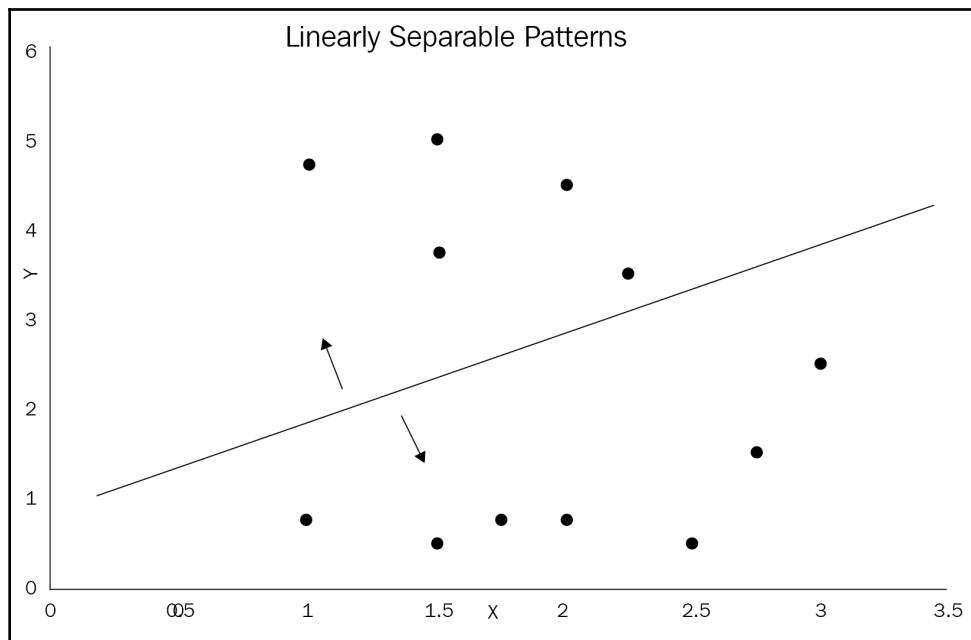
One might be tempted to think, *So what?* However, the entire field of neural networks relies on solving problems such as this to classify patterns. Without pattern classification, images, sounds, and words mean nothing to a machine.

Linearly separable models

The McCulloch-Pitts 1943 neuron (see Chapter 2, *Think Like a Machine*) lead to Rosenblatt's 1957-1962 perceptron and the 1960 Widrow-Hoff adaptive linear element (Adaline).

These models are linear models based on $f(x,w)$, requiring a line to separate results. A perceptron cannot achieve this goal and thus cannot classify many objects it faces.

A standard linear function can separate values. **Linear separability** can be represented in the following graph:



Imagine that the line separating the preceding dots and the part under it represent a picture that needs to be represented by a machine learning or deep learning application. The dots above the line represent *clouds* in the sky; the dots below the line represent *trees* on a hill. The line represents the slope of that hill.

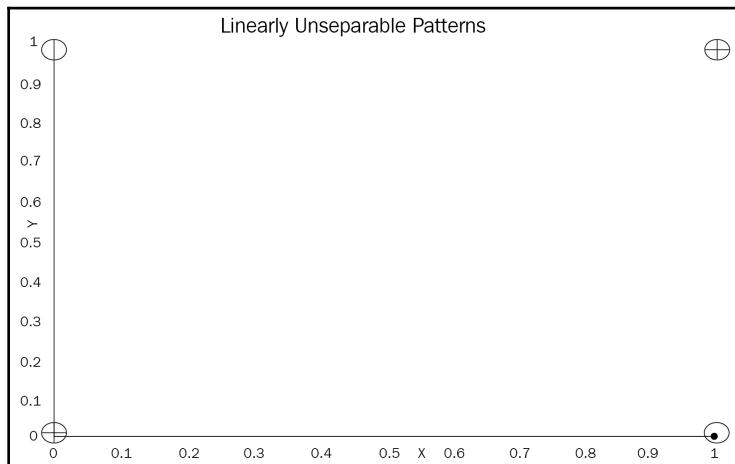
To be linearly separable, a function must be able to separate the *clouds* from the *trees* to classify them. The prerequisite to classification is **separability** of some sort, linear or nonlinear.

The XOR limit of a linear model, such as the original perceptron

A linear model cannot solve the XOR problem expressed as follows in a table:

Value of x_1	Value of x_2	Output
1	1	0
0	0	0
1	0	1
0	1	1

The following graph shows the linear inseparability of the XOR function represented by one perceptron:



The values of the table represent the Cartesian coordinates in this graph. The circle with a cross at $(1,1)$ and $(0,0)$ cannot be separated from the circles at $(1,0)$ and $(0,1)$. That's a huge problem. It means that Frank Rosenblatt's $f(x,w)$ perceptron cannot separate, and thus not classify, these dots into *clouds* and *trees*; an object used to identify that requires linear separability.

Having invented the most powerful concept of the 20th century—a **neuron that can learn**—Frank Rosenblatt had to bear with this limitation through the 1960s.

Let's vindicate this injustice with a vintage solution.

Building a feedforward neural network from scratch

Let's get into a time machine. In nanoseconds, it takes us back to 1969. We have today's knowledge but nothing to prove it. Minsky and Papert have just published their book, *Perceptrons*. They've proven that a perceptron cannot implement the exclusive OR function XOR.

We are puzzled. We know that deep learning will be a great success in the 21st century. We want to try to change the course of history. Thanks to our time machine, we land in a small apartment. It's comfortable, with a vinyl record playing the music we like! There is a mahogany desk with a pad, a pencil, sharpener, and eraser waiting for us. We sit. A warm cup of coffee appears in a big mug. We're ready to solve the XOR problem from scratch. We have to find a way to classify those dots with a neural network.

Step 1 – Defining a feedforward neural network

We look at our piece of paper. We don't have a computer. We're going to have to write code; then we'll hopefully find a computer in a university or a corporation that has a 1960 state-of-the-art language to program in.

We have to be unconventional to solve this problem. First, we must ignore Minsky and Papert's publication and also forget complicated words and theory of the 21st century. In fact, we don't remember much anyway. Time travel made our future fuzzy!

A perceptron is usually represented by a graph. But that doesn't mean much right now. After all, I can't compute circles and lines. In fact, beyond seeing circles and lines, we type characters in computer languages, not circles. So, I decide to simply to write a layer in high-school format. A hidden layer will simply be:

$$h_1 = x * w$$

Ok, now I have one layer. In fact, I just realized that a layer is merely a function. This function can be expressed as:

$$f(x, w)$$

In which x is the input value and w is some kind of value to multiply x by. I also realized that hidden just means that it's the computation, just as $x=2$ and $x+2$ is the hidden layer that leads to 4.

At this point, I've defined a neural network in three lines:

- Input x .
- Some kind of function that changes its value, like $2 \times 2 = 4$, which transformed 2. That is a layer. And if the result is superior to 2, for example, then great! The output is 1, meaning yes or true. Since we don't see the computation, this is the *hidden* layer.
- An output.

Now that I know that basically any neural network is built with values transformed by an operation to become an output of something, I need the logic to solve the XOR problem.

Step 2 – how two children solve the XOR problem every day

Let's see how two children solve the XOR problem using a plain everyday example. I strongly recommend this method. I have taken very complex problems, broken them down into small parts to children's level, and often solved them in a few minutes. Then, you get the sarcastic answer from others such as *Is that all you did?* But, the sarcasm vanishes when the solution works over and over again in high-level corporate projects.

First, let's convert the XOR problem into a candy problem in a store. Two children go to the store and want to buy candy. However, they only have enough money to buy one pack of candy. They have to agree on a choice between two packs of different candy. Let's say pack one is chocolate and the other is chewing gum. Then, during the discussion between these two children, 1 means yes, 0 means no. Their budget limits the options of these two children:

- Going to the store and not buying any of chocolate **or** chewing gum = no, no (0,0). That's not an option for these children! So the answer is false.
- Going to the store and buying both chocolate **and** chewing gum = yes, yes (1,1). That would be fantastic, but that's not possible. It's too expensive. So, the answer is unfortunately false.
- Going to the store and either buying chocolate **or** chewing gum = (1,0 or 0,1) = yes or no/no or yes. That's possible. So, the answer is true.

Sipping my coffee in 1969, I imagine the two children. The eldest one is reasonable. The younger one doesn't know really how to count yet and wants to buy both packs of candy.

I decide to write that down on my piece of paper:

- x_1 (eldest child's decision yes or no, 1 or 0) * w_1 (what the elder child thinks).
The elder child is thinking this, or:

$$x_1 * w_1 \text{ or } h_1 = x_1 * w_1$$

The elder child weighs a decision like we all do every day, such as purchasing a car ($x=0$ or 1) multiplied by the cost ($w1$).

- x_2 (the younger child's decision yes or no, 1 or 0) * w_3 (what the younger child thinks). The younger child is also thinking this, or:

$$x_2 * w_3 \text{ or } h_2 = x_2 * w_3$$



Theory: x_1 and x_2 are the inputs. h_1 and h_2 are neurons (the result of a calculation). Since h_1 and h_2 contain calculations that are not visible during the process, they are *hidden* neurons. h_1 and h_2 thus form a **hidden layer**.

Now I imagine the two children talking to each other.

Hold it a minute! This means that now each child is communicating with the other:

- x_1 (the elder child) says w_2 to the younger child. Thus $w2 = \text{this is what I think and am telling you}$:

$$x_1 * w_2$$

- x_2 (the younger child) says please add my views to your decision, which is represented by: w_4

$$x_2 * w_4$$

I now have the first two equations expressed in high-school-level code. It's *what one thinks + what one says to the other asking the other to take that into account*:

```
h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2
```

h_1 sums up what is going on in one child's mind: personal opinion + other child's opinion.

h_2 sums up what is going on in the other child's mind and conversation: personal opinion + other child's opinion.



Theory. The calculation now contains two input values and one hidden layer. Since in the next step we are going to apply calculations to h_1 and h_2 , we are in a feedforward neural network. We are moving from the input to another layer, which will lead us to another layer, and so on. This process of going from one layer to another is the basis of deep learning. The more layers you have, the deeper the network is. The reason h_1 and h_2 form a hidden layer is that their output is just the input of another layer.

I don't have time to deal with complicated numbers in an activation function such as logistic sigmoid, so I decide to simply decide whether the output values are less than 1 or not:

if $h_1+h_2 \geq 1$ then $y_1=1$

if $h_1+h_2 < 1$ then $y_2=0$



Theory: y_1 and y_2 form a second hidden layer. These variables can be scalars, vectors, or matrices. They are neurons.

Now, a problem comes up. Who is right? The elder child or the younger child?

The only way seems to be to play around, with the weights W representing all the weights.

I decided that at this point, I liked both children. Why would I have to hurt one of them? So from now on, $w_3=w_2, w_4=w_1$. After all, I don't have a computer and my time travel window is consuming a lot of energy. I'm going to be pulled back soon.

Now, somebody has to be an influencer. Let's leave this hard task to the elder child. The elder child, being more reasonable, will continuously deliver the bad news. You have to subtract something from your choice, represented by a minus (-) sign.

Each time they reach the point h , the eldest child applies a critical negative view on purchasing packs of candy. It's $-w$ of everything comes up to be sure not to go over the budget. The opinion of the elder child is biased, so let's call the variable a bias, b_1 . Since the younger child's opinion is biased as well, let's call this view a bias too b_2 . Since the eldest child's view is always negative, $-b_1$ will be applied to all of the eldest child's thoughts.

When we apply this decision process to their view, we obtain:

$$\begin{aligned} h_1 &= y_1 * -b_1 \\ h_2 &= y_2 * b_2 \end{aligned}$$

Then, we just have to use the same result. If the result is ≥ 1 then the threshold has been reached. The threshold is calculated as shown in the following function.

$$y = h_1 + h_2$$

Since I don't have a computer, I decide to start finding the weights in a practical manner, starting by setting the weights and biases to 0.5, as follows:

$$\begin{aligned} w_1 &= 0.5; w_2 = 0.5; b_1 = 0.5 \\ w_3 &= w_2; w_4 = w_1; b_2 = b_1 \end{aligned}$$

It's not a full program yet, but its theory is done.

Only the communication going on between the two children is making the difference; I focus on only modifying w_2 and b_1 after a first try. An hour later, on paper, it works!

I can't believe that this is all there is to it. I copy the mathematical process on a clean sheet of paper:

```
Solution to the XOR implementation with
a feedforward neural network (FNN)

I.Setting the first weights to start the process
w1=0.5;w2=0.5;b1=0.5
w3=w2;w4=w1;b2=b1

#II hidden layer #1 and its output
h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2

#III.threshold I, hidden layer 2
if(h1>=1):h1=1;
if(h1<1):h1=0;
```

```
if(h2>=1):h2=1
if(h2<1):h2=0
h1= h1 * -b1
h2= h2 * b2

IV.Threshold II and Final OUTPUT y
y=h1+h2
if(y>=1):y=1
if(y<1):y=0

V. Change the critical weights and try again until a solution is found
w2=w2+0.5
b1=b1+0.5
```

I'm overexcited by the solution. I need to get this little sheet of paper to a newspaper to get it published and change the course of history. I rush to the door, open it but find myself back in the present! I jump and wake up in my bedroom sweating.

I rush to my laptop while this time-travel dream is fresh in my mind to get it into Python for this book.



Why wasn't this deceiving simple solution found in 1969? Because *it seems simple today but wasn't so at that time like all inventions found by our genius predecessors*. Nothing is easy at all in artificial intelligence and mathematics.

Implementing a vintage XOR solution in Python with an FNN and backpropagation

I'm still thinking that implementing XOR with so little mathematics might not be that simple. However, since the basic rule of innovating is to be unconventional, I write the code.

To stay in the spirit of a 1969 vintage solution, I decide not to use NumPy, TensorFlow, Theano, or any other high-level library. Writing a vintage FNN with backpropagation written in high-school mathematics is fun.

This also shows that if you break a problem down into very elementary parts, you understand it better and provide a solution to that specific problem. You don't need to use a huge truck to transport a loaf of bread.

Furthermore, by thinking through the minds of children, I went against running 20,000 or more episodes in modern CPU-rich solutions to solve the XOR problem. The logic used proves that, basically, both inputs can have the same parameters as long as one bias is negative (the elder reasonable critical child) to make the system provide a reasonable answer.

The basic Python solution quickly reaches a result in 3 to 10 iterations (epochs or episodes) depending on how we think it through.

The top of the code simply contains a result matrix with four columns. Each represents the status (1=correct, 0=false) of the four predicates to solve:

```
#FEEDFORWARD NEURAL NETWORK(FNN) WITH BACK PROPAGATION SOLUTION FOR XOR
result=[0,0,0,0] #trained result
train=4 #dataset size to train
```

The train variable is the number of predicates to solve: (0,0), (1,1),(1,0),(0,1). The variable of the predicate to solve is pred.

The core of the program is practically the sheet of paper I wrote, as in the following code.

```
#II hidden layer 1 and its output
def hidden_layer_y(epoch,x1,x2,w1,w2,w3,w4,b1,b2,pred,result):
    h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
    h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2

#III.threshold I,a hidden layer 2 with bias
    if(h1>=1):h1=1;
    if(h1<1):h1=0;
    if(h2>=1):h2=1
    if(h2<1):h2=0

    h1= h1 * -b1
    h2= h2 * b2
#IV. threshold II and OUTPUT y
    y=h1+h2
    if(y<1 and pred>=0 and pred<2):
        result[pred]=1

    if(y>=1 and pred>=2 and pred<4):
        result[pred]=1
```

`pred` is an argument of the function from 1 to 4. The four predicates can be represented in the following table:

Predicate (<code>pred</code>)	x_1	x_2	Expected result
0	1	1	0
1	0	0	0
2	1	0	1
3	0	1	1

That is why y must be <1 for predicates 0 and 1. Then, y must be ≥ 1 for predicates 2 and 3.

Now, we have to call the following function limiting the training to 50 epochs, which are more than enough:

```
#I Forward and backpropagation
for epoch in range(50):
    if(epoch<1):
        w1=0.5;w2=0.5;b1=0.5
        w3=w2;w4=w1;b2=b1
```

At epoch 0, the weights and biases are all set to 0.5. No use thinking! Let the program do the job. As explained previously, the weight and bias of x_2 are equal.

Now the hidden layers and y calculation function are called four times, one for each predicate to train, as shown in the following code snippet:

```
#I.A forward propagation on epoch 1 and IV.backpropagation starting epoch 2
for t in range (4):
    if(t==0):x1 = 1;x2 = 1;pred=0
    if(t==1):x1 = 0;x2 = 0;pred=1
    if(t==2):x1 = 1;x2 = 0;pred=2
    if(t==3):x1 = 1;x2 = 0;pred=3
    #forward propagation on epoch 1
    hidden_layer_y(epoch,x1,x2,w1,w2,w3,w4,b1,b2,pred,result)
```

A simplified version of a cost function and gradient descent

Now the system must train. To do that, we need to measure the number of predictions, 1 to 4, that are correct at each iteration and decide how to change the weights/biases until we obtain proper results.

A slightly more complex gradient descent will be described in the next chapter. In this chapter, only a one-line equation will do the job. The only thing to bear in mind as an unconventional thinker is: so what? The concept of gradient descent is minimizing loss or errors between the present result and a goal to attain.

First, a cost function is needed.

There are four predicates (0-0, 1-1, 1-0, 0-1) to train correctly. We simply need to find out how many are correctly trained at each epoch.

The cost function will measure the difference between the training goal (4) and the result of this epoch or training iteration (result).

When 0 convergence is reached, it means the training has succeeded.

`result[0,0,0,0]` contains a 0 for each value if none of the four predicates has been trained correctly. `result[1,0,1,0]` means two out of four predicates are correct. `result[1,1,1,1]` means that all four predicates have been trained and that the training can stop. 1, in this case, means that the correct training result was obtained. It can be 0 or 1. The `result` array is the result counter.

The cost function will express this training by having a value of 4, 3, 2, 1, or 0 as the training goes down the slope to 0.

Gradient descent measures the value of the descent to find the direction of the slope: up, down, or 0. Then, once you have that slope and the steepness of it, you can optimize the weights. A derivative is a way to know whether you are going up or down a slope.

In this case, I hijacked the concept and used it to set the learning rate with a one-line function. Why not? It helped to solve gradient descent optimization in one line:

```
if (convergence<0) :w2+=0.05; b1=w2
```

By applying the vintage *children buying candy* logic to the whole XOR problem, I found that only `w2` needed to be optimized. That's why `b1=w2`. That's because `b1` is doing the tough job of saying something negative (-) all the time, which completely changes the course of the resulting outputs.

The rate is set at 0.05, and the program finishes training in 10 epochs:

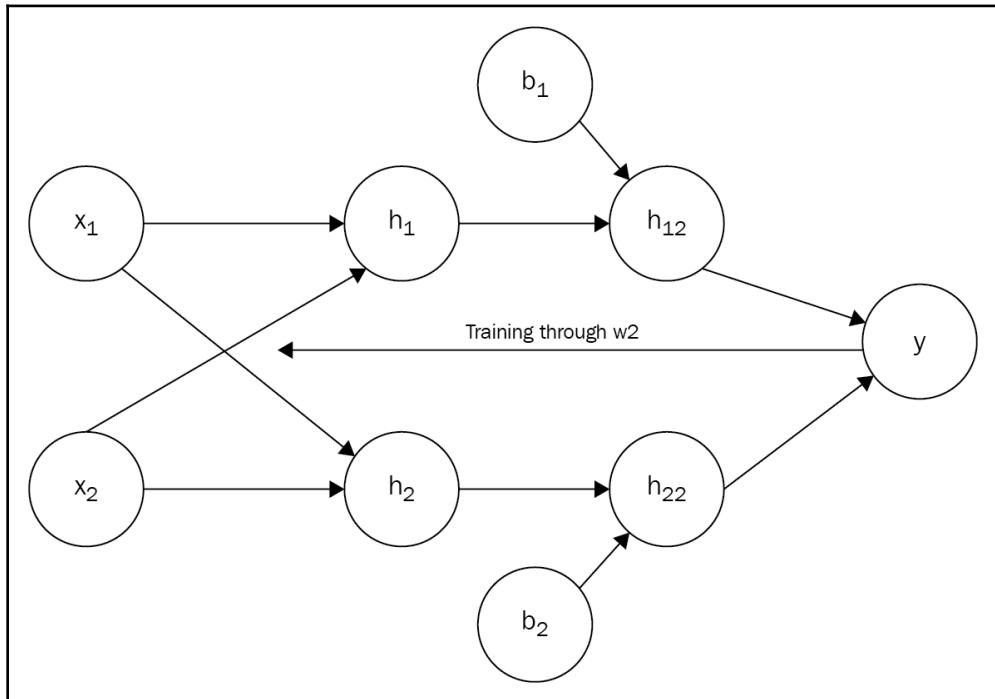
```
epoch: 10 optimization 0 w1: 0.5 w2: 1.0 w3: 1.0 w4: 0.5 b1: -1.0 b2: 1.0
```

This is not a mathematical calculation problem but a logical one, a *yes or no* problem. The way the network is built is pure logic. Nothing can stop us from using whatever training rates we wish. In fact, that's what gradient descent is about. There are many gradient descent methods. If you invent your own and it works for your solution, that is fine.

This one-line code is enough, in this case, to see whether the slope is going down. As long as the slope is negative, the function is going downhill to $cost = 0$:

```
convergence=sum(result)-train #estimating the direction of the slope
if(convergence>=-0.00000001): break
```

The following graph sums up the whole process:



Too simple? Well, it works, and that's all that counts in real-life development. If your code is bug-free and does the job, then that's what counts.

Finding a simple development tool means nothing more than that. It's just another tool in the toolbox. We can get this XOR function to work on a neural network and generate income.



Companies are not interested in how smart you are but how efficient (profitable) you can be.

Linear separability was achieved

Bear in mind that the whole purpose of this feedforward network with backpropagation through a cost function was to transform a linear non-separable function into a linearly separable function to implement classification of features presented to the system. In this case, the features had 0 or 1 value.



One of the core goals of a layer in a neural network is to make the input make sense, meaning to be able to separate one kind of information from another.

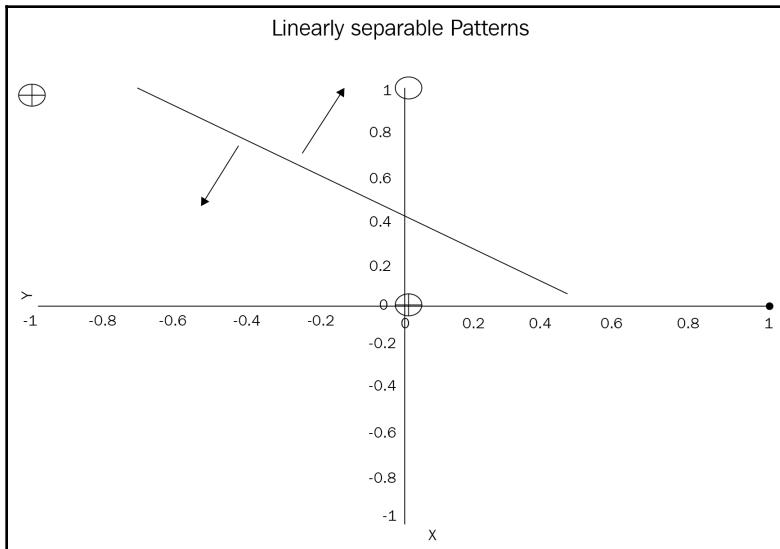
`h1` and `h2` will produce the Cartesian coordinate linear separability training axis, as implemented in the following code:

```
h1= h1 * -b1  
h2= h2 * b2  
print(h1,h2)
```

Running the program provides a view of the nonlinear input values once they have been trained by the hidden layers. The nonlinear values then become linear values in a linearly separable function:

```
linearly separability through cartesian training -1.0000000000000004  
1.0000000000000004  
linearly separability through cartesian training -0.0 0.0  
linearly separability through cartesian training -0.0 1.0000000000000004  
linearly separability through cartesian training -0.0 1.0000000000000004  
epoch: 10 optimization 0 w1: 0.5 w2: 1.0 w3: 1.0 w4: 0.5 b1: -1.0 b2: 1.0
```

The intermediate result and goal are not a bunch of numbers on a screen to show that the program works. The result is a set of Cartesian values that can be represented in the following linearly separated graph:



We have now obtained a separation between the top values (empty circle) representing the intermediate values of the 1,0 and 0,1 inputs, and the bottom values representing the 1,1 and 0,0 inputs. We now have *clouds* on top and *trees* below the line that separates them.

The layers of the neural network have transformed nonlinear values into linear separable values, making classification possible through standard separation equations, such as the one in the following code:

```
#IV. threshold II and OUTPUT y
y=h1+h2 # logical separation
if(y<1 and pred>=0 and pred<2):
    result[pred]=1

if(y>=1 and pred>=2 and pred<4):
    result[pred]=1
```

The ability of a neural network to make non-separable information separable and classifiable represents one of the core powers of deep learning. From this technique, many operations can be performed on data, such as subset optimization.

Applying the FNN XOR solution to a case study to optimize subsets of data

The case study described here is a real-life project. The environment and functions have been modified to respect confidentiality. But, the philosophy is the same one as that used and worked on.

We are 7.5 billion people breathing air on this planet. In 2050, there will be about 2.5 billion more. All of these people need to wear clothes and eat. Just those two activities involve classifying data into subsets for industrial purposes. **Grouping** is a core concept for any kind of production. Production relating to producing clothes and food requires grouping to optimize production costs. Imagine not grouping and delivering one t-shirt at a time from one continent to another instead of *grouping* t-shirts in a container and grouping many containers (not just two on a ship). Let's focus on clothing, for example.

A brand of stores needs to replenish the stock of clothing in each store as the customers purchase their products. In this case, the corporation has 10,000 stores. The brand produces jeans, for example. Their average product is a faded jean. This product sells a slow 50 units a month per store. That adds up to $10,000 \text{ stores} \times 50 \text{ units} = 500,000 \text{ units}$ or **stock keeping unit (SKU)** per month. These units are sold in all sizes grouped into average, small, and large. The sizes sold per month are random.

The main factory for this product has about 2,500 employees producing those jeans at an output of about 25,000 jeans per day. The employees work in the following main fields: cutting, assembling, washing, laser, packaging, and warehouse.

The first difficulty arises with the purchase and use of fabric. The fabric for this brand is not cheap. Large amounts are necessary. Each pattern (the form of pieces of the pants to be assembled) needs to be cut by wasting as little fabric as possible.

Imagine you have an empty box you want to fill up to optimize the volume. If you only put soccer balls in it, there will be a lot of space. If you slip tennis balls in the empty spaces, space will decrease. If on top of that, you fill the remaining empty spaces with ping pong balls, you will have optimized the box.



Building optimized subsets can be applied to containers, warehouse flows and storage, truckload optimizing, and almost all human activities.

In the apparel business, if 1% to 10% of fabric is wasted while manufacturing jeans, the company will survive the competition. At over 10%, there is a real problem to solve. Losing 20% on all the fabric consumed to manufacture jeans can bring the company down and force it into bankruptcy.



The main rule is to combine larger pieces and smaller pieces to make optimized cutting patterns.

Optimization of space through larger and smaller objects can be applied to cutting the forms which are patterns of the jeans, for example. Once they are cut, they will be assembled at the sewing stations.

The problem can be summed up as:

- Creating subsets of the 500,000 SKUs to optimize the cutting process for the month to come in a given factory
- Making sure that each subset contains smaller sizes and larger sizes to minimize loss of fabric by choosing six sizes per day to build 25,000 unit subsets per day
- Generating cut plans of an average of three to six sizes per subset per day for a production of 25,000 units per day

In mathematical terms, this means trying to find subsets of sizes among 500,000 units for a given day.

The task is to find six well-matched sizes among 500,000 units. This is calculated by the following combination formula:

$$C(n, r) = \frac{n!}{r!(n-r)!} = \frac{500000!}{6!(500000 - 6)!} = \log_{10} 10^{31.33}$$

At this point, most people abandon the idea and just find some easy way out of this even if it means wasting fabric. The problem was that in this project, I was paid on a percentage of the fabric I would manage to save. The contract stipulated that I must save 3% of all fabric consumption per month for the whole company to get paid a share of that. Or receive nothing at all. Believe me, once I solved that, I kept that contract as a trophy and a tribute to simplicity.

The first reaction we all have is that this is more than the number of stars in the universe and all that hype!

That's not the right way to look at it at all. The right way is to look exactly in the opposite direction. The key to this problem is to observe the particle at a microscopic level, at the **bits of information** level. This is a fundamental concept of machine learning and deep learning. Translated into our field, it means that to process an image, ML and DL process pixels. So, even if the pictures to analyze represent large quantities, it will come down to small units of information to analyze:

yottabyte (YB)	10^{24}	yobibyte (YiB)	2^{80}
----------------	-----------	----------------	----------

Today, Google, Facebook, Amazon, and others have yottabytes of data to classify and make sense of. Using the word **big** data doesn't mean much. It's just a lot of data, and so what?

You do not need to analyze individual positions of each data point in a dataset, but use the probability distribution.

To understand that, let's go to a store to buy some jeans for a family. One of the parents wants a pair of jeans, and so does a teenager in that family. They both go and try to find their size in the pair of jeans they want. The parent finds 10 pairs of jeans in size x . All of the jeans are part of the production plan. The parent picks one at *random*, and the teenager does the same. Then they pay for them and take them home.

Some systems work fine with random choices: random transportation (taking jeans from the store to home) of particles (jeans, other product units, pixels, or whatever is to be processed) making up that fluid (a dataset).

Translated into our factory, this means that a stochastic (random) process can be introduced to solve the problem.

All that was required is that small and large sizes were picked at random among the 500,000 units to produce. If six sizes from 1 to 6 were to be picked per day, the sizes could be classified as follows in a table:

$$\text{Smaller sizes} = S = \{1, 2, 3\}$$

$$\text{Larger sizes} = L = [4, 5, 6]$$

Converting this into numerical subset names, $S=1$ and $L=6$. By selecting large and small sizes to produce at the same time, the fabric will be optimized:

Size of choice 1	Size of Choice 2	Output
6	6	0
1	1	0
1	6	1
6	1	1

Doesn't this sound familiar? It looks exactly like our vintage FNN, with 1 instead of 0 and 6 instead of 1. All that has to be done is to stipulate that subset $S=value\ 0$, and subset $L=value\ 1$; and the previous code can be generalized.

If this works, then smaller and larger sizes will be chosen to send to the cut planning department, and the fabric will be optimized. Applying the randomness concept of Bellman's equation, a stochastic process is applied, choosing customer unit orders at random (each order is one size and a unit quantity of 1):

```
w1=0.5;w2=1;b1=1
w3=w2;w4=w1;b2=b1
s1=random.randint(1,500000)#choice in one set s1
s2=random.randint(1,500000)#choice in one set s2
```

The weights and bias are now constants obtained by the result of the XOR training FNN. The training is done; the FNN is simply used to provide results. Bear in mind that the word *learning* in machine learning and deep learning doesn't mean you have to train systems forever. In stable environments, training is run only when the datasets change. At one point in a project, you are hopefully using deep *trained* systems and are not just stuck in the deep *learning* process. The goal is not to spend all corporate resources on learning but on using trained models.



Deep learning architecture must rapidly become deep trained models to produce a profit or disappear from a corporate environment.

For this prototype validation, the size of a given order is random. 0 means the order fits in the S subset; 1 means the order fits in the L subset. The data generation function reflects the random nature of consumer behavior in the following six-size jeans consumption model.

```
x1=random.randint(0, 1)#property of choice:size smaller=0
x2=random.randint(0, 1)#property of choice :size bigger=1
hidden_layer_y(x1,x2,w1,w2,w3,w4,b1,b2,result)
```

Once two customer orders have been chosen at random chosen at random in the right size category, the FNN is activated, the FNN is activated and runs like the previous example. Only the result array has been changed because no training is required. Only a yes (1) or no (0) is expected, as shown in the following code:

```
#II hidden layer 1 and its output
def hidden_layer_y(x1,x2,w1,w2,w3,w4,b1,b2,result):
    h1=(x1*w1)+(x2*w4) #II.A.weight of hidden neuron h1
    h2=(x2*w3)+(x1*w2) #II.B.weight of hidden neuron h2

#III.threshold I,a hidden layer 2 with bias
    if(h1>=1):h1=1;
    if(h1<1):h1=0;
    if(h2>=1):h2=1
    if(h2<1):h2=0
    h1= h1 * -b1
    h2= h2 * b2

#IV. threshold II and OUTPUT y
    y=h1+h2
    if(y<1):
        result[0]=0
    if(y>=1):
        result[0]=1
```

The number of subsets to produce needs to be calculated to determine the volume of positive results required.

The choice is made of six sizes among 500,000 units. But, the request is to produce a daily production plan for the factory. The daily production target is 25,000. Also, each subset can be used about 20 times. There is always, on average, 20 times the same size in a given pair of jeans available.

Each subset result contains two orders, hence two units:

$$R=2 \times 20 = 120$$

Each result produced by the system represents a quantity of 120 for 2 sizes.

Six sizes are required to obtain good fabric optimization. This means that after three choices, the result represents one subset of potential optimized choices:

$$R = 120 \times 3 \text{ subsets of 2 sizes} = 360$$

The magic number has been found. For every 3 choices, the goal of producing 6 sizes multiplied by a repetition of 20 will be reached.

The production per day request is 25,000:

$$\text{The number of subsets requested} = 25000/3=8333.333$$

The system can run 8333 products as long as necessary to produce the volume of subsets requested. In this case, the range is set to 1000000 products because only the positive results are accepted. The system is filtering the correct subsets through the following function:

```
for element in range(1000000):
    if(result[0]>0):
        subsets+=1
        print("Subset:",subsets,"size subset #",x1," and ", "size subset #",x2,
              "result:",result[0],"order #", " and ",s1,"order #",s2)
        if(subsets>=8333):
            break
```

When the 8333 subsets have been found respecting the smaller-larger size distribution, the system stops, as shown in the following output.

```
Subset: 8330 size subset # 1 and size subset # 0 result: 1 order # and
53154 order # 14310
Subset: 8331 size subset # 1 and size subset # 0 result: 1 order # and
473411 order # 196256
Subset: 8332 size subset # 1 and size subset # 0 result: 1 order # and
133112 order # 34827
Subset: 8333 size subset # 0 and size subset # 1 result: 1 order # and
470291 order # 327392
```

This prototype proves the point. Not only was this project a success with a similar algorithm, but also the software ran for years in various forms on key production sites, reducing material consumption and generating a profit each time it ran. The software later mutated in a powerfully advanced planning and scheduling application.

Two main functions, among some minor ones, must be added:

- After each choice, the orders chosen must be removed from the 500,000-order dataset. This will preclude choosing the same order twice and reduce the number of choices to be made.

- An optimization function to regroup the results by trained optimized patterns, by an automated planning program for production purposes.

Application information:

- The core calculation part of the application is less than 50-lines long
- When a few control functions and dataset tensors are added, the program might reach 200 lines maximum
- This guarantees easy maintenance for a team



It takes a lot of time to break a problem down into elementary parts and find a simple, powerful solution. It thus takes much longer than just typing hundreds to thousands of lines of code to make things work. The simple solution, however, will always be more profitable and software maintenance will prove more cost effective.

Summary

Building a small neural network from scratch provides a practical view of the elementary properties of a neuron. We saw that a neuron requires an input that can contain many variables. Then, weights are applied to the values with biases. An activation function then transforms the result and produces an output.

Neuronal networks, even one- or two-layer networks, can provide real-life solutions in a corporate environment. The real-life business case was implemented using complex theory broken down into small functions. Then, these components were assembled to be as minimal and profitable as possible.

Customers expect quick-win solutions. Artificial intelligence provides a large variety of tools that satisfy that goal. When solving a problem for a customer, do not look for the best theory, but the simplest and fastest way to implement a profitable solution no matter how unconventional it seems.

In this case, an enhanced perceptron solved a complex business problem. In the next chapter, an FNN will be introduced using TensorFlow.

5

Manage the Power of Machine Learning and Deep Learning

Mastering machine learning and deep learning is proportional to your ability to design the architectures of these solutions. As developers, we tend to rush to some sample code, run it, and then try to implement it somehow. That's like going to a big city we don't know, with no map and no guiding system, and trying to find a street. Even worse, it's like trying to build a 50-storey building with no architect or plans.

An efficient, well-thought architecture will lead to a good solution. Deep learning networks are data flow graph calculations as shown in *Chapter 4, Become an Unconventional Innovator*. A node or edge is, in fact, a mathematical operation. The lines connecting these nodes are data flows and mathematical representations. Tools such as TensorFlow and TensorBoard have been designed for data flow graph representations and calculations. Without using TensorBoard, the graph representation of a network, there is little to no chance of understanding and designing deep networks.

The sponsors of a project need to understand the concepts of a given solution. This chapter explores how to use these tools and, at the same time, make a successful presentation to a CEO, top managers, or your team.

The following topics will be covered in this chapter:

- Building a **feedforward neural network (FNN)** with TensorFlow
- Using TensorBoard, a data flow graph, to design your program
- Using the data flow graph to write a program
- A cost function, a loss function
- Gradient descent
- Backpropagation in a feedforward network
- Presenting an FNN to a team or management

Technical requirements

- Python 3.6x 64-bit from <https://www.python.org/>
- NumPy for Python 3.6x
- TensorFlow from <https://deepmind.com/> with TensorBoard

Programs: GitHub Chapter05.



The Python programs delivered with this chapter constitute a sandbox for you to play around with to become familiar with TensorFlow and TensorBoard. Feel free to run them, modify the TensorFlow data flow graphs, and see what happens in TensorBoard.

- `FNN_XOR_Tensorflow.py`: This program shows how to train an FNN to solve the XOR problem in a few lines.
- `FNN_XOR_Tensorflow_graph.py`: This program adds TensorBoard metadata to display the data flow graph of the program.
- `FNN_XOR_Tensorflow_graph_only.py`: This program contains a data flow graph of the program just to show that mastering the graph means mastering the program.
- `Tensorboard.reader.py`: This program runs with the preceding programs. It finds the `/log` file containing the data to display in TensorBoard and runs the TensorBoard local service.
- `FNN_XOR_Tensorflow_tensorboard_MODEL1.py`: This program shows how to tweak a `FNN_XOR_Tensorflow_graph_only.py` to add some labels to prepare a corporate presentation.

Check out the following video to see the code in action:

<https://goo.gl/QrKrSA>

Building the architecture of an FNN with TensorFlow

Before applying an FNN to a problem, an architecture of the network must be built. A TensorFlow architecture and solution to the XOR function is the place to start. The architecture of the example differs from the vintage, built-from-scratch solution but the concepts remain the same.

Writing code using the data flow graph as an architectural roadmap

TensorFlow is a graph-driven solution based on graph theory, a branch of mathematics. Designing deep learning without graphs would prove quite difficult. My XOR FNN built from scratch in Chapter 4, *Become an Unconventional Innovator*, fits the need of the case study described. However, if thousands of nodes are required, TensorFlow, Keras (with TensorFlow backend), or other similar solutions will be needed. In this chapter, a neural network will be developed in TensorFlow.

The building blocks of graphs are nodes (or vertices, or points) and edges (or lines) that connect the nodes to one another.

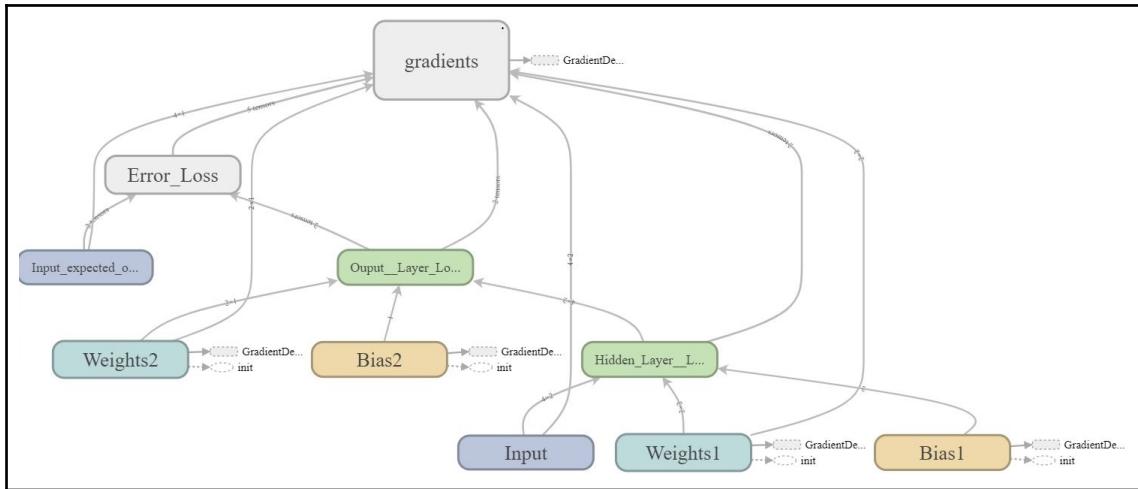
In TensorFlow parlance, nodes are cleverly named ops for operations. In Chapter 4, *Become an Unconventional Innovator*, the Python example shows exactly that: nodes as operations and calculations.

Mathematical models will be described through a data flow graph. The first step is to build the model, your architecture.

In a TensorFlow program, the first top source code lines are not just a new way to describe variables that you will use later. They are the program. They are critical to the computation of the graph. In fact, TensorFlow uses the graph to compute the operations. *The graph is the architecture, the roadmap that describes the whole computation process of a deep learning neural network.* The following graph has taken the code lines and transformed them into a graph. Imagine that the following graph is a network of water pipes in the city. First, the city builds the pipes. Only then does the water (data in our case) flow in those pipes. The pipes (TensorFlow variables, arrays, and other objects) are not a formality. They represent the core architecture of a system.

This graph is built with `FNN_XOR_Tensorflow_graph_only.py` to show how to use TensorBoard to analyze your network. It will prove easier to fix design errors with TensorBoard. Click on the nodes to explore the details of each part of the graph. Run the program; then run `TensorBoard_reader.py` to launch TensorBoard.

The following diagram illustrates this:



A dataflow graph

A data flow graph translated into source code

The data flow graph will be used to build the architecture of a project. It can be used as a project management tool and project optimization tool as shown in the last part of this chapter.

The input data layer

The input data on the graph represents each pair of two inputs computed at a time in this XOR model using `FNN_XOR_Tensorflow.py`.

This program shows how to build a TensorFlow network in a few lines. In other models, millions of pixels or other values can constitute the input. In this case, XOR is trained with this model:

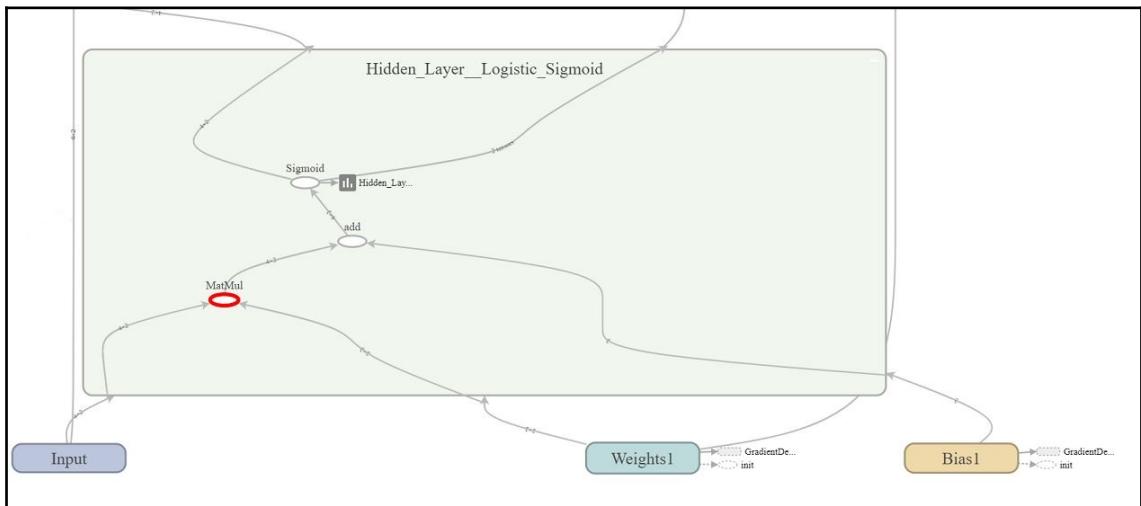
```
XOR_X = [[0,0],[0,1],[1,0],[1,1]]
```

The first two inputs are (0,0), for example. The input will contain $x_1=0$ and $x_2=0$. Those inputs are sent to the hidden layer along with the weights and biases:

```
W1 = tf.Variable(tf.random_uniform([2,2], -1, 1), name = "Weights1")
B1 = tf.Variable(tf.zeros([2]), name = "Bias1")
```

The hidden layer

The hidden layer contains a multiplication function of weights and an addition of biases, as shown in the following diagram:



A dataflow graph with multiplication function

The input is sent to the multiplication function along with the weights. Thus x_1 and x_2 will be multiplied by the weights as explained in Chapter 4, *Become an Unconventional Innovator*. In this model, the bias variable is added to the result of the multiplications to each of the hidden neurons (one for each x). Go back to Chapter 4, *Become an Unconventional Innovator*, again to see the low-level mechanics of the system if necessary.

Bear in mind that there are several mathematical ways, and thus there are several architectures represented in graphs that can solve the FNN XOR problem. Once the multiplication and addition have been computed, a logistic sigmoid function (see Chapter 2, *Think like a Machine*) is applied to the result.

Why is a logistic sigmoid function needed here?

In Chapter 4, *Become an Unconventional Innovator*, in my vintage program, I did not use one. I just rounded the values to 0 or 1. That works in a limited environment. However, as explained in Chapter 2, *Think like a Machine*, the output of the neurons needs to be **normalized** to be useful when the input data starts increasing and diversifying. To sum it up, suppose we obtain 200 as an input of a neuron and 2 as an output of another neuron. That makes it very difficult to manage this in the next layer. It's as if you went to a shop and had to pay for one product in a given currency and another product in a different currency. Now the cashier has to add that up. It's not practical at all in a shop and even less in a network with thousands of neurons.

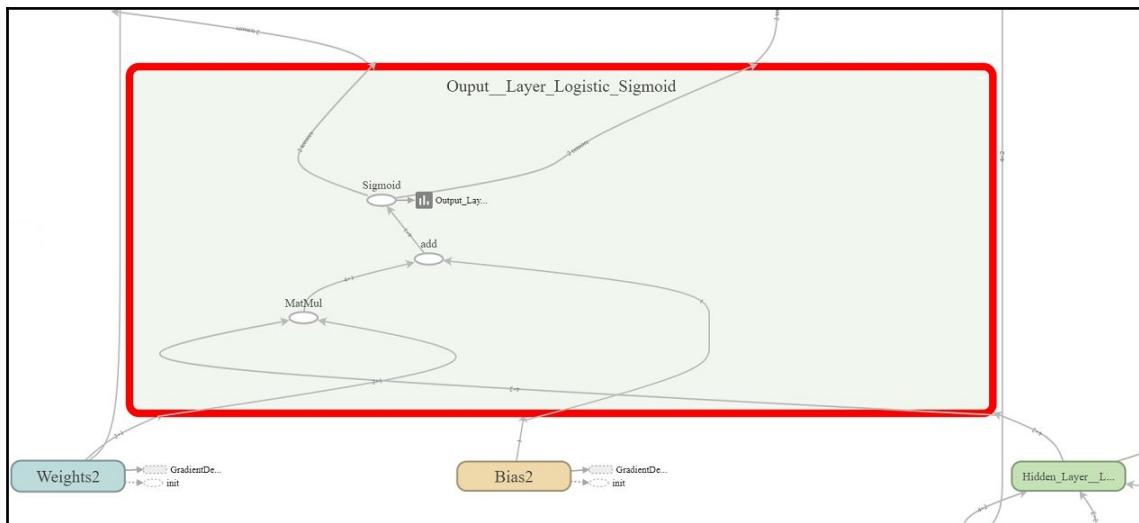
The following LS logistic sigmoid is thus a very handy tool for the outputs to level the values:

```
LS = tf.sigmoid(tf.matmul(x_, W1) + B1)
```

Basically, the values are squashed into small numbers that are compatible with the next step, which is the computation of the output layer.

The output layer

The output layer will now receive the result $y = (\text{input} * \text{weights} + \text{bias})$ squashed by logistic sigmoid (y), as shown in the following diagram:



A dataflow graph with output layer

The output layer takes the y output of the hidden layer (on the right-hand side of the graph) and receives the weights and bias to apply to this layer in this model. Once the multiplication of weights and addition of bias are computed, the logistic sigmoid function squashes this output as well. This is a very nice way of keeping values homogeneous over the network. It's a tidy way of always sending small computable values to the next layer as implemented in the following code.

```
Output = tf.sigmoid(tf.matmul(LS, W2) + B2)
```

Now that the output has been obtained, what is it worth? How is it evaluated and how are the weights adjusted?

The cost or loss function

The following cost function compares the expected result y_{-} with the present result produced by the network.

```
cost = tf.reduce_mean((y_- * tf.log(Output)) + ((1 - y_-) * tf.log(1.0 - Output)) * -1)
```

The `tf.log` is applied to the results to squash them for efficient comparisons.

The `tf.reduce_mean` supports a variety of cost functions; this is one way to calculate the error between the training dataset and the intermediate result.

Another way to calculate a cost function, as shown in the following code, is to compare the output of the current iteration with the target output and square both to obtain a visible difference.

```
cost = tf.reduce_mean(tf.square(y_-Output))
```

Choosing a cost function will depend on your model. This method in its minimized format is called **least squares**. It produces efficient results in this model.

Gradient descent and backpropagation

Gradient descent defines two concepts:

- Gradient or derivative measurement of the slope (up or down / how steep)
- Descent or reducing the error level between the present result, relying on the parameters (weights and biases), and the target training dataset

There are several ways to measure whether you are going up or down a slope. Derivatives are the most commonly used mathematical tool for this. Let us say you have 15 steps to go from one floor of a building down to the next floor. At each step, you feel you are going down.

The slope or derivative is the function describing you going down those stairs:

- S = slope of you going down the stairs
- dy = where you are once you have made a step (up, down, or staying on a step)
- dx = the size of your steps (one at a time, two at a time, and so on)
- $f(x)$ = the fact of going downstairs, for example from step 4 to step 3, step by step.
- The derivative or slope is thus:

$$\frac{dy}{dx}$$

$f(x)$ is the function of you going down (or up or stopping on a step). For example, when you move one step forward, you are going from step 4 to step 3 (down). Thus $f(x)=x-b$, in which $b = 1$ in this example. This is called a decreasing function. h = the number of steps you are going at each pace, for example, one step down if you are not in a hurry and two steps down if you are in a hurry. In this example, you are going down one step at a time from step 4 to 3; thus $h = 1$ (one step at a time).

We obtain the following formula:

$$\frac{dy}{dx} = \frac{f(x+h)-f(x)}{h} = -1$$

This means that we are at step 3 after taking one step. We started at step 4, so we went down -1 step. The minus sign means you are going downstairs from step 4 to step 3 or -1 step.

The gradient is the direction the function is taking. In this case, it is -1 step. If you go down two steps at a time, it would be -2. A straightforward way is just to take the derivative and use it. If it's 0, we're not moving. If it's negative we're going down (less computation or less remaining time to train).

If the slope is positive, we are in trouble, we're going up and increase the cost of the function (more training or more remaining time to train).

The goal of an FNN is to converge to 0. This means that as long as the parameters (weights and biases) are not optimized, the output is far from the target expected. In this case, for example, there are four predicates to train (1-1,0-0,1-0,0-1). If only two results are correct, the present situation is negative, which is $2 - 4 = -2$. When three results are correctly trained, the output gradient descent is $3 - 4 = -1$. This will be translated into derivative gradient descent form using the cost calculated. Descent means that 0 is the target when all four outputs out of four are correct. This arithmetic works for an example without complicated calculations. But TensorFlow provides functions for all situations to calculate the cost (cost function), see whether the training session is going well (down), and optimize the weights to be sure that the cost is diminishing (gradient *descent*). The following `GradientDescentOptimizer` will optimize the training of the weights.

```
cost = tf.reduce_mean(tf.square(y_Output))
train_step = tf.train.GradientDescentOptimizer(0.10).minimize(cost)
```

A current learning rate for the gradient descent optimizer is 0.01. However, in this model, it can be sped up to 0.10.



The `GradientDescentOptimizer` will make sure the slope is following a decreasing function's gradient descent by optimizing the weights of the network accordingly.

In this TensorFlow example, the means of the output values are calculated and then squashed with the logistic function. It will provide information for the inbuilt gradient descent optimizer to minimize the cost, with a small training step (0.01). This means that the weights will be slightly changed before each iteration. An iteration defines backpropagation. By going back running the FNN again, then measuring, and going back again, we are *propagating* many combinations of weights—hopefully in the right direction (down the slope)—to optimize the network.



Stochastic gradient descent (SGD) consists of calculating gradient descent on samples of random (stochastic) data instead of using the whole dataset every time.

Running the session

The last step is to run the architecture that has just been designed. The thinking was done in the previous steps.

To run the program, the minimum code is for opening a session and running iterations, as shown in the following code snippet:

```
#II.data

XOR_X = [[0,0],[0,1],[1,0],[1,1]]
XOR_Y = [[0],[1],[1],[0]]

#III.data flow graph computation

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

for epoch in range(50000):
    sess.run(train_step, feed_dict={x_: XOR_X, y_: XOR_Y})
```

Feeding the data into the data flow graph is done with `feed_dict`. **Feed** is a keyword in FNN. This is an important feature. It is possible to feed parts of a dataset, not all of the data in stochastic (picking only parts of the dataset to train) gradient descent models as explained in [Chapter 6, Don't Get Lost in Techniques – Focus on Optimizing Your Solutions](#).

Checking linear separability

The session has now taken the four XOR linearly non-separable (see [Chapter 4, Become an Unconventional Innovator](#)) predicates, that is, four possibilities, and computed a linearly separable solution:

The input of the output expected was the following:

```
XOR_Y = [[0],[1],[1],[0]]
```

One of the results obtained is the following:

```
Output [[ 0.01742549]
 [ 0.98353356]
 [ 0.98018438]
 [ 0.01550745]]
```

The results are separable as described in [Chapter 4, Become an Unconventional Innovator](#). If you plot the values, you can separate them with a line.

The results are displayed by printing the intermediate values as the session runs, as shown in the following code:

```
if epoch % 10000 == 0:  
    print('Epoch ', epoch)  
    print('Output ', sess.run(Output, feed_dict={x_: XOR_X, y_: XOR_Y}))  
    print('Weights 1 ', sess.run(W1))  
    print('Bias 1 ', sess.run(B1))  
    print('Weights 2 ', sess.run(W2))  
    print('Bias 2 ', sess.run(B2))  
    print('cost ', sess.run(cost, feed_dict={x_: XOR_X, y_: XOR_Y}))
```

These source code lines are difficult to visualize. Using a graph to represent your architecture helps to see the strong and weak points of a solution.

Using TensorBoard to design the architecture of your machine learning and deep learning solutions

FNN_XOR_Tensorflow_tensorboard_graph_only.py contains no program in the traditional sense. It contains an essential aspect of developing machine learning and neural networks: the architecture. This program is overloaded with basic graph summaries to grasp the core concept of the data flow graph. In subsequent programs in this book, the summaries will be less literal.



Master the *architecture* of a machine learning and deep learning program and you'll master your solutions.

Designing the architecture of the data flow graph

First import TensorFlow with the following code:

```
import tensorflow as tf
```

Then use the following code to give a name of the scope of each object you want to display in one line. Write your variable or object on the second line and include it in the TensorBoard summary as an image (histograms, scalars, and other summaries are possible), as shown in the following code:

```
with tf.name_scope("Input"):  
    x_ = tf.placeholder(tf.float32, shape=[4,2], name = 'x-input-predicates')  
    tf.summary.image('input predicates x', x_, 10)
```



name provides a label for the objects in the graph. scope is a way to group the names to simplify the visual representation of your architecture.

Repeat that with all the elements described in the previous section, as follows.

```
with tf.name_scope("Input_expected_output"):  
    y_ = tf.placeholder(tf.float32, shape=[4,1], name = 'y-expected-output')  
    tf.summary.image('input expected values of y', x_, 10)  
  
...  
  
with tf.name_scope("Error_Loss"):  
    cost = tf.reduce_mean(( (y_ * tf.log(Output)) + ((1 - y_) * tf.log(1.0 - Output)) ) * -1)  
    tf.summary.image("Error_Loss", cost, 10)
```

Then, once the session is created, set up the log writer directory, as shown in the following code:

```
init = tf.global_variables_initializer()  
sess = tf.Session()  
writer = tf.summary.FileWriter("./logs/tensorboard_logs", sess.graph)  
sess.run(init)  
#THE PROGRAM DATA FEED CODE GOES HERE. See FNN_XOR_Tensorflow.py  
writer.close()
```

Those are the basic steps to create the logs to display in TensorBoard.



Note that this source code only contains the architecture of the data flow graph and no program running the data. The program is the architecture that defines computations. The rest is defining the datasets—how they will be fed (totally or partially) to the graph. I recommend that you use TensorFlow to design the architecture of your solutions with the TensorFlow graph code. Only then, introduce the dynamics of feeding data and running a program.

Displaying the data flow graph in TensorBoard

A practical way to obtain a visual display of your architecture and results is to have your own little `TensorBoard_reader.py`, as shown in the following lines of code:

```
def launchTensorBoard():
    import os
    os.system('tensorboard --logdir=' + './logs/tensorboard_logs')
    return

import threading
t = threading.Thread(target=launchTensorBoard, args=[])
t.start()

#In your browser, enter http://localhost:6006 as the URL
#and then click on Graph tab. You will then see the full graph.
```

All you have to do is `git` the output directory to the log directory path in your program. It can be included in the main source code.

Once launched, you will see images of the previous section. You are now ready to enter corporate environments as a solid architect thinker and designer.



Use a TensorBoard reader such as this for your meetings. You will be viewed as an architect. You can use Microsoft PowerPoint or other tools naturally. But at one point, even a single view of this data flow graph, from a few seconds up to as long as your audience is interested, shows that you master the architecture of the solution. That is, you master the subject.

The final source code with TensorFlow and TensorBoard

After running the programs, to make sure nothing is missing, the TensorFlow and TensorBoard models can be merged and simplified as shown in `FNN_XOR_Tensorflow_graph.py`.

The summaries have been simplified, the cost function uses the optimized least squares method, and XOR linear separability has been proven with this model.

Using TensorBoard in a corporate environment

Choosing the right architecture for your machine learning or deep learning solution is key to the technical success of your project.

Then, being able to explain it fewer than 2 minutes to a CEO, a top manager, or even a member of your team is key to the commercial success of your project. If they are interested, they will ask more questions and you can drill down. First, you have to captivate their attention. You spend time on your work. However, selling that idea or work to somebody else is extremely difficult.

A slight change in the inputs can do the job, as shown in `FNN_XOR_Tensorflow_tensorboard_MODEL1.py` in the following code sample:

```
with tf.name_scope("input_store_products"):  
    x_ = tf.placeholder(tf.float32, shape=[4,2], name = 'x-input-predicates')#placeholder is an operation supplied by the feed  
    tf.summary.image('input store products', x_, 10)  
  
with tf.name_scope("input_expected_top_ranking"):  
    y_ = tf.placeholder(tf.float32, shape=[4,1], name = 'y-expected-output')  
    tf.summary.image('input expected classification', y_, 10)
```

Show the graph in full-screen mode and double-click on each node to highlight it while speaking about a particular aspect of your architecture. The graph can be grouped into large scopes by putting placeholders under the same `tf.name_scope`. For example, in the following code, `input_store_products` was added to make a presentation:

```
with tf.name_scope("input_store_products"):  
    x_ = tf.placeholder(tf.float32, shape=[4,2], name = 'x-input-predicates')#placeholder is an operation supplied by the feed  
    y_ = tf.placeholder(tf.float32, shape=[4,1], name = 'y-expected-output')
```

Use your imagination to find the best and clearest way to make your architecture understood.

Display the graph in full-screen mode and double-click on each node to highlight it while speaking about it.

Using TensorBoard to explain the concept of classifying customer products to a CEO

Designing solutions seem fine, but if you can't sell them to your team or management, they won't go far. More often than not, we think that since we have a good technical background, we'll be recognized. Nothing is more deceiving. In a world where knowledge is everywhere, competition will be at your door everyday. Furthermore, managers will have less and less time to listen to what we have to say.



The architecture of your solution depends on what your buyers (team managers, prospects, or customers) ask you. *Your freedom to implement your views depends on how well you present your architecture in a way that fits their needs.*

It's 8:30. The CEO has dropped into your office and is touring the company. The CEO is sharp, bright, and has no patience with people who can't express themselves clearly. You lower your head and look in the garbage can to find something that's not there. It doesn't work. The CEO comes straight to you, invites you to a meeting room and asks you to present what you are doing. A few other managers come in to see what's going on and whether you will survive the presentation.

Will your views on the project survive this meeting?

You can take your corporate PowerPoint out and bore your audience, or you can captivate it.

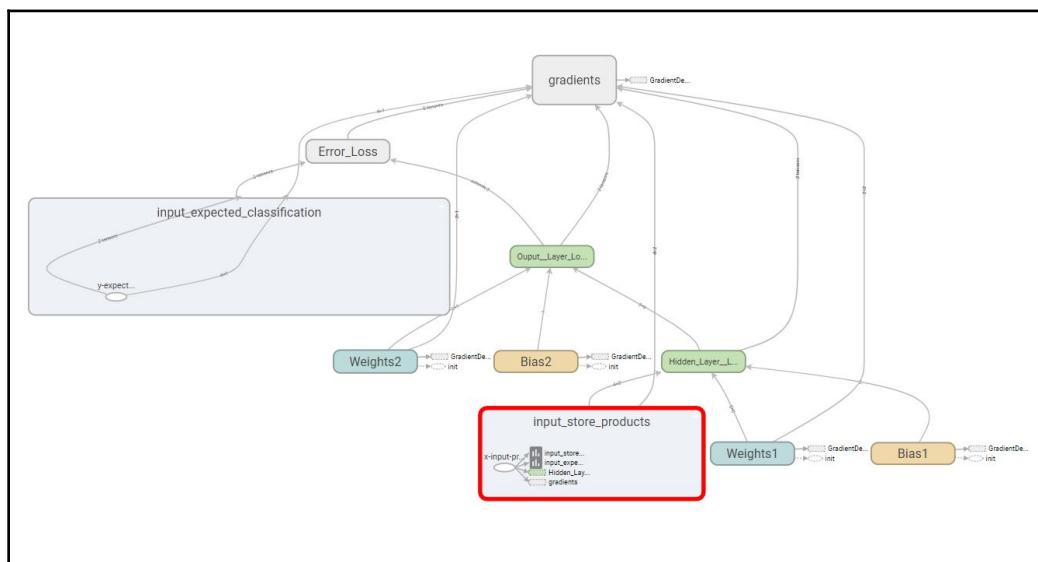
I have gone through these meetings hundreds of times with a 90% sales success rate and have refined a method that can be summed up in my 2-minute pitch, which always goes something like the following.

I will display a chart and the following graph and say:

"Let's go straight to the point. We're trying to analyze the details of products based on their images and find the most profitable ones. Here is a one-page chart showing the list of the most promising products. Here is TensorBoard; it comes with TensorFlow package. The computation power of TensorFlow's data flow guarantees disruptive innovative profits for this project.

At the bottom of the screen, you can see that we are going to input store products (point to the input_store_products box). What we expect is the profitable classification of products: the best ones that represent our brand AND the ones that also generate the most profit. So basically, there are two main types of features to extract: forms, shapes, and colors of the most trendy products; and profit.

To achieve that, we feed (that's literally the language we use in code) the system with the results we expect, a dataset of good choices (point to the input_expect_classification box). Then we run the system; it optimizes until it has been trained through gradient descent (point your arm and finger downwards), which is simply a cost function to lower (CEO's understand lowering costs in 1/1,000,000th of a second) the errors. Once the weights (just as we weigh decisions) are optimized (I point to the weights), that's it. We will provide you each day with reports to drive the marketing department."



A dataflow graph with input_store_products

Then I stop immediately, slowly look at everybody, and say:

"To sum it up, we select the most profitable products in real time with this system and provide the marketing department with real-time AI reports to boost their marketing campaigns. By analyzing the features of the products, they will have more precise marketing data to drive their sales."

If you read this aloud slowly, it should take you be under 2 minutes, which is your first-level survival time.

The rules I've applied successfully are as follows:

- Explain from off the top (global concepts).
- Use as few words as possible, but choose the right ones. Don't avoid technical concepts, but break them down into small key parts. This means that, once again, TensorBoard architecture will develop this ability.
- Speak slowly. Speaking slowly, stopping at each sentence, emphasizing each keyword, and pointing to the graph at the same time is the only way to make sure you are not losing anybody.
- After 2 minutes, sum your introduction up and ask: *"Am I clear? Do you want me to continue. Questions?"* If they want you to continue, then you can drill down, stopping at every key point to make sure that everybody is still with you.



Bear in mind that many CEOs are *business visionaries*. They see things we don't. So, if they come up with an idea, be very constructive and innovative.

Was the sales pitch convincing?

At this point, if you have a good project, the CEO will most probably say one of the following:

- *"Gee, that was clear and I understand what you are doing. Thanks. Bye."* That's not too bad.

- "That looks too simple to be true. Can you give some more details?" This could be an opportunity. The CEO wants me to explain more. So I drill down another 2 minutes around the way the weights are optimized in a clear way just like the first 2 minutes. At that point, in 20% of my meetings, I ended up having lunch or dinner with the CEO, who always has 100 other ideas for uses of such a program. Bear in mind that many CEOs are *business visionaries*. They see things we don't. So, if a new idea comes up, I generally say, "Great. I'll look into this and get back to you tomorrow." Then I spend the night coming up with something that works and is explained clearly.
- "Good. I'd like you to talk to X and see where this is going." Not bad either. Hopefully, X will listen to the CEO.

Statistically, being short and concise with real technical tangible objects to show worked 90% of the time for me. The 10% failures are lessons to learn from to keep the statistic up the next time around.



The percentage of failure of our presentations is our *cost function*. We have to bring that number down by learning from these failures by backpropagation as well. We need to go back and change our parameters to adapt to the next run.

Summary

Mastering the architecture of a machine learning and deep learning solution means first, and above all, becoming an expert in designing the architecture of a solution and being able to explain it.

The early lines of a source code program are not just variable declaration lines; they represent the data flow graph that drives the computation of a neural network. These lines define the architecture of your solutions. They are critical to the future of an artificial intelligence solution.

The architecture of the solution, represented by TensorBoard, for example, serves two purposes. The first purpose defines the way the computation of the graph will behave. The second purpose is to use the architecture as a communication tool to sell your project to your team, management, prospects, and customers. Our environment defines our project and not simply our technical abilities, no matter how much we know.

The next chapter focuses on analyzing the core problems of machine learning and deep learning programs.

6

Focus on Optimizing Your Solutions

No matter how much we know, the key point remains being able to deliver an artificial intelligence solution or not. Implementing a **machine learning (ML)** or **deep learning (DL)** program remains difficult and will become more complex as technology progresses at exponential rates. This will be shown in Chapter 13, *Cognitive NLP Chatbots*, on quantum computing, which may revolutionize computing with its mind-blowing concepts. There is no such thing as a simple or easy way to design AI systems. A system is either efficient or not, beyond being either easy or not. Either the designed AI solution provides real-life practical uses or it builds up into a program that fails to work in various environments beyond the scope of training sets.

This chapter doesn't deal with how to build the most difficult system possible to show our knowledge and experience. It faces the hard truth of real-life delivery and ways to overcome obstacles. Without data, your project will never take off. Even an unsupervised ML program requires unlabeled data in some form or other.

Beyond understanding **convolutional neural networks (CNN)** that can recognize **Modified National Institute of Standards and Technology (MNIST)** training sets of handwritten images as described in Chapter 10, *Applying Biomimicking to Artificial Intelligence*, efficiency comes first. In a corporate environment, you will often have to deal with designing datasets and obtaining data. Doubts follow quickly. Are the datasets for a given project reliable? How can this be measured? How many features need to be implemented? What about optimizing the cost function and training functions?

This chapter provides the methodology and tools needed to overcome everyday artificial intelligence project obstacles. k-means clustering, a key ML algorithm, will be applied to the increasing need for warehouse intelligence (Amazon and all online product-selling sites). Quickly showing the solution to a problem will keep a project alive. Focusing on the solution, and not the techniques, will get you there.

The following topics will be covered in this chapter:

- Designing datasets
- The design matrix
- Dimensionality reduction
- Determining the volume of a training set
- k-means clustering
- Unsupervised learning
- Data conditioning management for the training dataset
- Lloyd's algorithm
- Building a Python k-means clustering program from scratch
- Hyperparameters
- Test dataset and prediction
- Presenting the solution to a team

Technical requirements

- Python 3.6x 64-bit from <https://www.python.org/>
- sklearn
- pandas
- matplotlib

Programs, Chapter06:

- `k-means_clustering_ch6.py` with `data.csv`

Check out the following video to see the code in action:

<https://goo.gl/kBzrp6>

Dataset optimization and control

On day one of a project, a manager or customer will inevitably ask the AI expert what kind of data is being dealt with, and in which format. Providing an answer to that will take some hard work. The AI expert never really thought of that, having acquired expertise using the available, optimized, and downloadable training datasets to test the programs. The expert's name is Pert Ex (an anagram of expert) in this chapter. The expert likes to be called **Pert**.

The CEO of the corporation (or boss or customer) wants Pert to lead an **Amazonization** project. Pert doesn't quite understand what that means. The CEO has the patience to explain that wholesale e-commerce has overtaken traditional ways of doing business.

Wholesale means increasing use of warehouses. How those warehouses are managed will make a difference to the cost of each project and the sales price.

Although the business is now huge and the warehouse is expanding, the costs of retrieving products from the warehouse locations have been increasing. The corporation, which we will name AGV-AI in this chapter, represents the sum of several real-life projects I have managed on various warehouse and aerospace sites. Bear in mind that moving large components in an aerospace project requires precise scheduling. Moving large volumes of small products in an Amazonization process or small volumes of oversized products (25 meters or 82 feet for example) such as aerospace components can be costly if badly planned. In AGV-AI, **automatic guided vehicles** (AGVs) are surprisingly taking longer routes from a warehouse location to a pier. AGV-AI manufactures, stores, and delivers large volumes of consumer goods.

The CEO wants Pert's team to find out what is going on and Pert must lead the technical aspects of the project.

Designing a dataset and choosing an ML/DL model

Pert goes back to his desk in an open space occupied by a team of 50 people and tries to represent what is wrong in the Amazonization process at the warehouse level.

On paper, it comes down to moving from warehouse locations to piers represented by a distance. The sum of the distances **D** could be a way to measure the process:

$$D = \sum_{p1}^{pn} f(p)$$

$f(p)$ represents the action of going from a warehouse location to a pier and the distance it represents. Exactly like the distance, it takes you from a location in a shop where you picked (p) something up and the door of the shop on your way out. You can imagine that if you go straight from that location, pay, and go out, then that is a shorter way. But if you pick the product up, wander around the shop first, and then go out, the distance (and time) is longer. The sum of all of the distances of all the people wandering in this shop is D .

Pert now has a clear representation of the concept of the problem to solve: **find the wanderers**. How can an AGV wander? It's automatic. After spending a few days watching the AGVs in the warehouse, Pert understands why the AGVs are taking longer routes sometimes.



When an AGV encounters an obstacle either it stops or, in this case, it takes another route. Just like us in a shop when we have to walk around an obstacle to get where we want.

Pert smiles. Pert has a plan.

Approval of the design matrix

Pert's plan is to first obtain as much data as possible and then choose an ML/DL model. His plan is to represent all locations the AGVs come from on their way to the piers on a given day. It's a location-to-pier analysis. Since the AGVs were at that location at one point and their distances are recorded in their system, Pert goes over to the IT manager with a **design matrix**. A design matrix contains a different example on each row, and each column is a feature. Pert wants to play safe and comes up with the following format:

	AGV number	Start from location: Timestamp:yyyy,mm,dd,hh,mm	End at pier: Timestamp:yyyy,mm,dd,hh,mm	Location	Pier number	Distance
Ex1	1	year-month-day-hour-minute	year-month-day-hour-minute	80		
Ex2	2					
Ex3	3					
Ex4	4					
Ex5	5					

Agreeing on the format of the design matrix

The IT manager has been around AGV-AI for over 10 years and knows where all the existing data is stored. Pert explains that the design matrix is one of the best ways to start training a ML program. In this case:

- AGV number identifies the vehicle.
- Start indicates when the AGV left a location (*Location* column).
- End indicates when the AGV reached a pier (*Pier* column).
- Distance is expressed in the metric system. Pert explains that most space agencies around the world use the metric system.

The IT manager starts off by saying the following:

- The AGV number is not stored in the mainframe but the local system that manages the AGVs
- In the mainframe, there is a start time when an AGV picks up its load at the location and an end time when it reaches the pier
- The location can be obtained in the mainframe as well as the pier
- No distance is recorded

Pert would like to have access to the data in the AGV's local system to retrieve the AGV number and correlate it with the data in the mainframe.

The IT manager looks at Pert and decides to tell the artificial intelligence expert the simple truth:

1. Retrieving data from the AGV guiding system is not possible this fiscal year. Those vehicles are expensive and no additional budget can be allocated.
2. Nobody knows the distance it takes from a location to a pier. As long as the AGVs deliver the proper products on time at the right piers, nobody so far has been interested in distances.
3. The AGV mission codes in the mainframe are not the same as in the local AGV guiding system, so they cannot be merged into a dataset without development.

The project is slipping away. AGV-AI doesn't have the leisure or budget to invest man days in extracting data that may or may not be used to optimize warehouse distances in its Amazonization process. Pert has to find a solution very quickly before the very busy IT manager drops the whole idea. Pert decides to think of this overnight and come back quickly the next day.

Keeping an AI project alive means moving quickly.



Dimensionality reduction

Pert is in trouble, and he knows it. Not all the data Pert would have liked can be obtained for the moment. Some features must be left aside for the moment. He thinks back to his walk from the IT manager's office to his office and looks at his cup of coffee.

Dimensionality reduction comes up!

DL uses dimensionality reduction to reduce the number of features in, for example, an image. Each pixel of a 3D image, for example, is linked to a neuron, which in turn brings the representation down to a 2D view with some form of function. For example, converting a color image into shades of a now-gray image can do the trick. Once that is done, simply reducing the values to, for example, 1 (light) or 0 (dark) makes it even easier for the network. Using an image converted to 0 and 1 pixels makes some classification processes more efficient, just like when we avoid a car on the road. We just see the object and avoid it. We are not contemplating the nice color or the car to avoid or other **dimensions**.

We perform dimensionality reduction all day long. When you walk from one office to another on the same floor of a building requiring no stairs or an elevator, you are not thinking that the earth is round and that you're walking over a slight curve. You have performed a **dimensionality reduction**. You are also performing a **manifold** operation. Basically, it means that locally, on that floor, you do not need to worry about the global roundness of the earth. Your manifold view of earth in your dimensionality reduction representation is enough to get you from your office to another one on that floor.

When you pick up your cup of coffee, you just focus on not missing it and aiming for the edges of it. You don't think about every single **feature** of that cup, such as its size, color, decoration, diameter, and the exact volume of coffee in it. You just identify the edge of the cup and pick it up. That is dimensionality reduction. Without **dimensionality reduction**, nothing can be accomplished. It would take you 10 minutes to analyze the cup of coffee and pick it up in that case!

When you pick that cup of coffee up, you test to see whether it is too hot, too cold, too warm, or just fine. You don't put a thermometer in the cup to obtain the precise temperature. You have again performed a **dimensionality reduction of the features** of that cup of coffee. Furthermore, when you picked it up, you computed a **manifold** representation by just observing the little distance around the cup, reducing the dimension of information around you. You are not worrying about the shape of the table, whether it was dirty on the other side, and other features.

Pert begins to imagine a (CNN) with dimensionality reduction. But the data will be insufficient for a CNN (see Chapter 9, *Getting Your Neurons to Work*).

There is no time to build a complicated DL program to compute dimensionality reduction and satisfy this need. Automatic dimensionality reduction will be dealt with later when the project has been accepted.



ML and DL techniques such as dimensionality reduction can be viewed as tools and be used in any field in which you find them useful. Reducing the number of features or modifying them can be done on a spreadsheet by selecting the useful columns and/or modifying the data. By focusing on the solution, you can use any tool in any way you want as long as it works in a reliable way.

Pert then turns to a k-means clustering ML algorithm and decides to perform dimensionality reduction by analysis of the data and defining it for a computation. Each location will form a cluster as explained in the next section. With this intuition, Pert goes back and presents the following format:

	Location	Start from location: Timestamp:yyyy,mm,dd,hh,mm	End at location: Timestamp:yyyy,mm,dd,hh,mm
Ex1			

The IT manager is puzzled. Pert explains that as long as each location represents a separate AGV task from start time to the end time and the pier, the rest will be calculated.

The IT manager says that it can be extracted in .csv format within a couple of days for a test but asks how that makes the project possible. Pert says, *"Provide the data, and I'll be back with a prototype within a couple of days as well."*

The volume of a training dataset

Pert decides to keep the dataset down to six locations over two days. Six locations are chosen in the main warehouse with a group of truck loading points. Taking around 5,000 examples into account, that should represent the work of all the 25 AGVs purchased by AGI-AI running 24 hours a day. The IT manager delivers the data.csv file the next day.

Implementing a k-means clustering solution

Pert now has a real data.csv dataset. The dataset needs to be converted into a prototype to prove the financial value of the project he has in mind. The amount of the money saved must capture the imagination of the CEO. *Making a machine learning program work in a corporate environment means nothing without profit to justify its existence.*



Do not speak about a project in a corporate environment without knowing how much **profit** it represents and the **cost** of getting the job done.

The vision

Pert's primary goal can be summed up in one sentence: finding profit by optimizing AGVs. Achieving that goal will lead to obtaining a budget for a full-scale project.

The data provided does not contain distances. However, an estimation can be made per location as follows:

$$\text{distance} = (\text{end time} - \text{start time}) / \text{average speed of an AGV}$$

The start location is usually a loading point near a pier in this particular warehouse configuration.

The data

The data provided contains start times (*st*), end times (*endt*), and delivery locations. To calculate distances, Pert needs to convert those times into a distance (d_i) with the following estimation for each location (l_i):

$$d_i(l_i) = (\text{endt} - \text{st}) / v$$

- v = velocity of the AGV per minute
- $\text{endt} - \text{st}$ is expressed in minutes
- d_i = estimated distance an AGV has gone in a given time

A Python program reads the initial file format and data and outputs a new file and format as follows:

Distance	Location
55	53
18	17

Conditioning management

The data file is available at <https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter06/data.csv>.

Data conditioning means preparing the data that will become the input of the system. **Poor conditioning** can have two outcomes:

- Bad data making no difference (large volumes and minor errors)
- Bad data making a difference (regardless of volumes, the data influences the outcome)

In this particular case, out of the 5,000 records provided for the dataset, 25 distances are missing.

Pert has analyzed the warehouse and is lucky!

The location numbers start at #1. #1 is near the loading point of the products. The AGV has to bring the products to this point. To be more precise, in this warehouse configuration, the AGV goes and gets a box (or crate) of products and brings them back to location 1. At location 1, humans check the products and package them. After that, humans carefully load the products in the delivery trucks.

Having spent hours measuring distances in the warehouse, Pert knows that the distance from one location to the next is about 1 meter. For example, from location 1 to location 5, the distance is about 5 meters, or 5 m. Also since all locations lead to location 1 for the AGVs in this model, the theoretical distances will be calculated from location 1. Pert decides to generalize the rule and defines a distance d_i for a location l_j as follows:

$$d_i(l_j) = l_j$$

d_i is expressed in meters. Since the locations start at number 1 through n , the location number is equal to the approximate distance from the first location from which the AGVs depart. Looking at the data, it quickly appears that many distances are superior to their location numbers. That is strange because the distance should be about equal to the location.

The time has come to build a strategy and program.

The strategy

Pert's strategy can be summed up as follows:

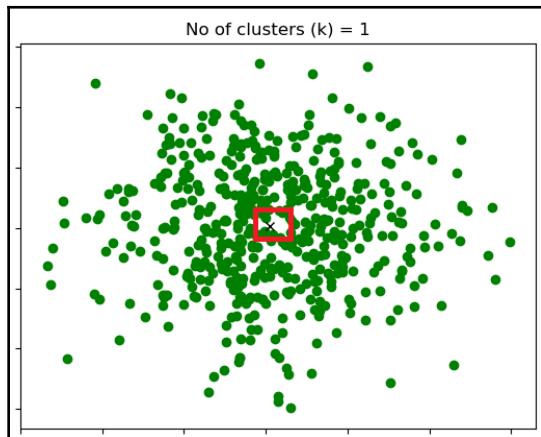
- Quickly write a program to represent the AGV's trajectories in a visual way
- Go to the warehouse to check the results
- Calculate the potential optimized profit with a solution that is yet to be found
- Go back to present this to the top management to get a project budget approval

The k-means clustering program

k-means clustering is a powerful unsupervised learning algorithm. We often perform k-means clustering in our lives. Take, for example, a lunch you want to organize for a team of about 50 people in an open space that can just fit those people. It will be a bit cramped but a project must be finished that day and it is a good way to bind the team for the final sprint.

A friend and a friend first decide to set up a table in the middle. Your friend points out that the people in that room will form a big cluster k , and with only one table in the geometric center (or centroid) c , it will not be practical. The people near the wall will not have access to the main table.

You have an idea. You call Pert, who runs a computation to confirm your friend's intuition. Pert shows them the problem with the following one-table plan:



The people not close to the table (rectangle in the middle) will not have easy access to the table.

You go to the room with your friend and Pert and try moving two tables c_1 and c_2 in various places for two clusters of people k_1 and k_2 .

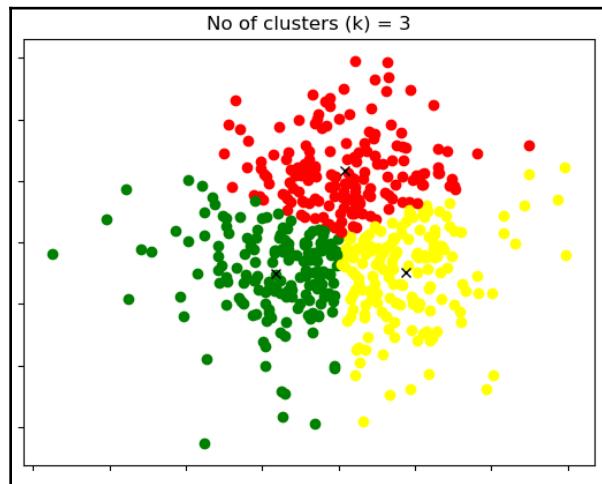
The people x_1 to x_n form a dataset X. Your friend still has doubts and says, "Look at table c_1 . It is badly positioned. Some people x near the wall will be right next to it and the others too far away. We need the table c_1 to be in the center of that cluster k_1 . The same goes for table c_2 ."

You and your friend move a table c , and then estimate that the mean distance of the people x around it will be about the same in their group or cluster k . They do the same for the other table. They draw a line with chalk on the floor to make sure that each group or cluster is at about the mean distance from its table.

Pert speaks up and says, "Gee, we can simulate that with this Python program I am writing for my project!":

- **Step 1:** You have been drawing lines with chalk to decide which group (cluster k) each person x will be in, by looking at the mean distance from the table c
- **Step 2:** You have been moving the tables around accordingly to optimize step 1

Pert shows them the two-table model computed by the k-means clustering program; it looks exactly like what you just did. Then they finally add a third table and it looks good. Pert says, "Look at what you did on the following screenshot.":



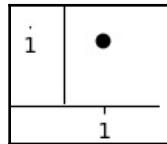
You and your friend look at Pert and say, "So what?"

Pert smiles and says, "Well, you just re-invented Lloyd's algorithm to solve a k-means clustering problem!"

You and your friend are curious. You want to know more. So you each get a paperboard, and ask Pert to explain to you the warehouse project or any project with this model in mathematical terms.

The mathematical definition of k-means clustering

The dataset X provides N points. These points or data points are formed by using distance as the x axis in a Cartesian representation and the location as the y axis in a Cartesian representation. If you have 1 person in on location 1 as the first record of your file, it will be represented as $x \text{ axis} = 1$ and $y \text{ axis} = 1$ by the black dot, which is the data point, as shown in the following diagram:



Cartesian representation

In this example, 5,000 records are loaded from `data.csv`, which is in the same directory as the program. The data is unlabeled with no linear separation. The goal is to allocate the X data points to K clusters. The number of clusters is an input value. Each cluster will have its geometric center or centroid. If you decide to have three clusters K, then the result will be as follows:

- Three clusters K in three colors in a visual representation
- Three geometric centers or centroids representing the center of the mean of the sum of distances of x data points of that cluster

If you decide on six clusters, then you will obtain six centroids, and so on.

Described in mathematical terms, the formula in respect of K_k, μ_k is as follows:

$$\text{Min} \sum_{k=1}^K \sum_{x_n \in K_k} \| x - \mu_k \|^2$$

This means that the sum of each k (cluster) from 1 to the number of clusters K of the sum of all distances of members x_i to x_n of each cluster K from their position to the geometric center (centroid) μ must be minimized.

The smaller the distance from each member x to centroid μ , the more the system is optimized. Note that the distance is squared each time because this is a Euclidean distance in this version of the equation.

Lloyd's algorithm

There are several variations of Lloyd's algorithm. But all of them follow a common philosophy.

For a given x_n (data point), the distance from the centroid μ in its cluster must be less than going to another center. Exactly like how a person in the lunch example wants to be closer to one table rather than having to go far to get a sandwich because of the crowd!

The best centroid μ for a given x_n is as follows:

$$x_n : \| x_n - \mu_k \|$$

This calculation is done for all μ (centroids) in all the clusters from k_1 to K .

Once each x_i has been allocated to a K_k , the algorithm recomputes μ by calculating the means of all the points that belong to each cluster and readjusts the centroid μ_k .

The goal of k-means clustering in this case study

The goal must be clear, not simple, not easy, for Pert's managers to understand. Pert decides to write a Python program first. Then if things go well, a full-scale project will be designed. The goal must be to find the *gain zone*. The gain zone will be the data points that could have been optimized in terms of distances.

In projects you build, distances can be replaced by costs, hours, or any optimization measurement feature. Location can be replaced by a feature such as the number of people, phone calls, or any other measurement.

k-means clustering can be applied to a range of applications as explained in Chapter 7, *When and How to Use Artificial Intelligence*.

The Python program

The Python program uses the `sklearn` library (which contains efficient algorithms), `pandas` for data analysis (only used to import the data in this program), and `Matplotlib` to plot the results as data points (the coordinates of the data) and clusters (data points classified in each cluster with a color). First the following models are imported:

```
from sklearn.cluster import KMeans
import pandas as pd
from matplotlib import pyplot as plt
```

NumPy is not a prerequisite since `sklearn` runs the computation.

1 – The training dataset

The training dataset consists of 5,000 lines. The first line contains a header for maintenance purposes (data checking), which is *not* used. k-means clustering is an **unsupervised learning** algorithm, meaning that it classifies unlabeled data into cluster-labeled data to make future predictions. The following code displays the dataset:

```
#I.The training Dataset
dataset = pd.read_csv('data.csv')
print (dataset.head())
print(dataset)
```

The `print (dataset)` line can be useful though not necessary to check the training data during a prototype phase or for maintenance purposes. The following output confirms that the data was correctly imported:

```
'''Output of print(dataset)
Distance location
0 80 53
1 18 8
2 55 38
...
'''
```

2 – Hyperparameters

Hyperparameters determine the behavior of computation method. In this case, two hyperparameters are necessary:

- The k number of clusters that will be computed. This number can and will be changed during the case study meetings to find out the best organization process, as explained in the next section. After a few runs, Pert intuitively set k to 6.
- The f number of features that will be taken into account. In this case, there are two features: distance and location.

The program implements a k-means function as shown in the following code:

```
#II.Hyperparameters  
# Features = 2  
k = 6  
kmeans = KMeans(n_clusters=k)
```

Note that the `Features` hyperparameter is commented. In this case, the number of features is implicit and determined by the format of the training dataset, which contains two columns.

3 – The k-means clustering algorithm

`sklearn` now does the job using the training dataset and hyperparameters in the following lines of code:

```
#III.k-means clustering algorithm  
kmeans = kmeans.fit(dataset) #Computing k-means clustering
```

The `gcenter` array contains the geometric centers or centroids and can be printed for verification purposes, as shown in this snippet:

```
gcenters = kmeans.cluster_centers_  
print("The geometric centers or centroids:")  
print(gcenters)  
'''Output of centroid coordinates  
[[ 48.7986755 85.76688742]  
[ 32.12590799 54.84866828]  
[ 96.06151645 84.57939914]]
```

```
[ 68.84578885 55.63226572]
[ 48.44532803 24.4333996 ]
[ 21.38965517 15.04597701]]
'''
```

These geometric centers need to be visualized with labels for decision-making purposes.

4 – Defining the result labels

The initial unlabeled data can now be classified into cluster labels as shown in the following code:

```
#IV.Defining the Result labels
labels = kmeans.labels_
colors = ['blue','red','green','black','yellow','brown','orange']
```

Colors can be used for semantic purposes beyond nice display labels. A color for each top customer or leading product can be assigned, for example.

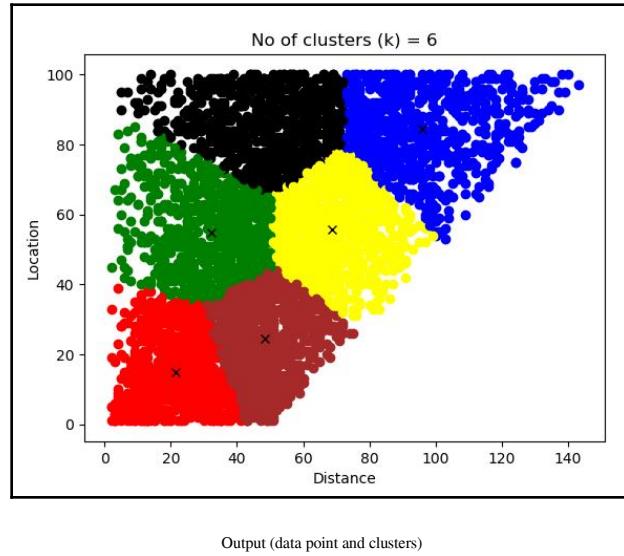
5 – Displaying the results – data points and clusters

To make sense to a team or management, the program now prepares to display the results as **data points** and **clusters**. The data will be represented as coordinates and the clusters as colors with a **geometric center** or **centroid**, as implemented in this code:

```
#V.Displaying the results : datapoints and clusters
y = 0
for x in labels:
    plt.scatter(dataset.iloc[y,0], dataset.iloc[y,1],color=colors[x])
    y+=1
for x in range(k):
    lines = plt.plot(gcenters[x,0],gcenters[x,1],'kx')

title = ('No of clusters (k) = {}'.format(k))
plt.title(title)
plt.xlabel('Distance')
plt.ylabel('Location')
plt.show()
```

The dataset is now ready to be analyzed. The data has been transformed into data points (Cartesian points) and clusters (the colors). The x points represent the geometric centers or centroids, as shown in the following screenshot:



Test dataset and prediction

In this case, the test dataset has two main functions. First, some test data confirms the **prediction** level of the trained and now-labeled dataset. The input contains random distances and locations. The following code implements the output that predicts which cluster the data points will be in:

```
#VI. Test dataset and prediction
x_test = [[40.0,67],[20.0,61],[90.0,90],
[50.0,54],[20.0,80],[90.0,60]]
prediction = kmeans.predict(x_test)
print("The predictions:")
print (prediction)
'''
Output of the cluster number of each example
[3 3 2 3 3 4]
'''
```

The second purpose, in future, will be to enter new warehouse data for **decision-making** purposes, as explained in the next section.

Analyzing and presenting the results

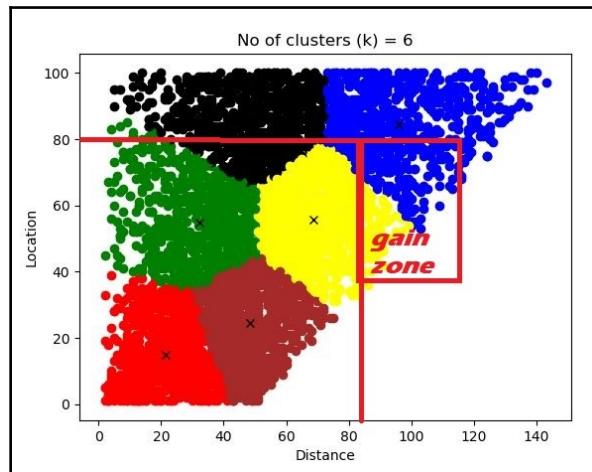
Pert knows the two-minute presentation rule.



If you have not captivated your manager or any audience within the two minutes of a presentation, it's over.

People will begin to look at their mail on their phones or laptops; pain replaces the pleasure of sharing a solution.

Pert decides to start the presentation with the following screenshot and a strong introduction:



Gain zone area

Pert points to the gain zone area and says:

From the computations made, that gain zone is only part of the losses of the present way AGVs are used in our company. It takes only a few locations into account. I would say that we could save 10% of the distances of our 25 AGVs on this site only. If we add nine other major sites in the world that adds up to:

*10% of (25 AGVs * 10 sites) = a gain of 25 AGVs*

That's because the AGVs are not going directly to the right locations but are *wandering* around unplanned obstacles.

Pert has everybody's attention!

In real life, I started my career with major corporations by making the following statement in similar cases (factories, warehouses, purchasing department, scheduling departments, and many other fields):

Pay me a percentage of that gain, and I will do the project for free until that is proven.

If the solution is mathematically solid and the data available, there is a risk to calculate to enter a corporation. When that is achieved once, it is easier the times after.

In this case, Pert's audience asks how this was estimated. The explanation is not simple but clear. The average distance from one location to another is 1 meter. The AGVs all start out from location 0 or 1. So the distance is strictly proportional to the locations in this particular example (which is not always the case).

To find the gain zone of a location, you draw a red horizontal line from location 80, for example, and a vertical line from distance 80 (add a couple of meters to take small variances into account).

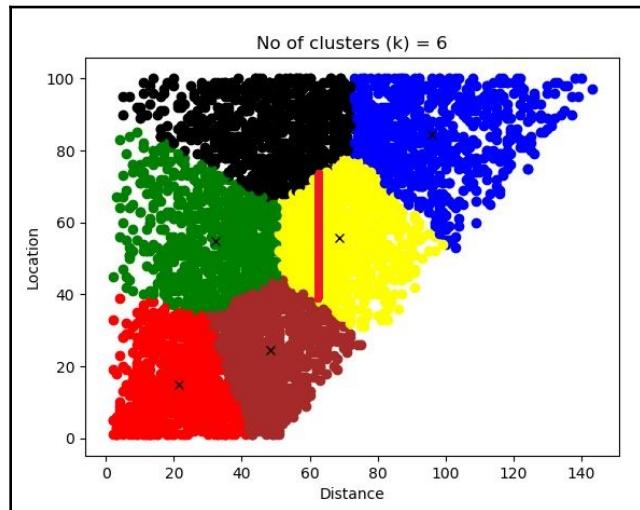
None of the data points on the 80 location line should be beyond the maximum limit. The limit is 80 meters + a small variance of a few meters. Beyond that line, on the right-hand side of the figure, is where the company is losing money and something must be done to optimize the distances. This loss zone is the gain zone for a project. The gain zone on the k-means cluster results shows that some of the locations of 40 to 60 exceed 80 meters distance.

Something is wrong, and a project can fix it.

AGV virtual clusters as a solution

The planners anticipate AGVs tasks. They send them to probable locations from which they will have to pick up products and bring them back to the truck-loading points.

One of the solutions is to provide AGV virtual clusters as a business rule, as shown in the following screenshot:



AGV virtual clusters

- **Rule 1:** The line in the middle represents a new business rule. In phase 1 of the project, an AGV used for locations 40 to 60 cannot go beyond that 60 meter + small variance line.
- **Rule 2:** A cluster will represent the pick-up zone for an AGV. The centroid will now be its parking zone. Distances will be optimized until all the clusters respect rule 1.

The financial manager and CEO approve Pert's prototype. But there are now two conditions for this to become a project:

- Present an architecture for large volumes for 10 sites, not just this site.
- Present the *same* model but with different features to optimize the e-commerce of the corporation's websites that now represent 90% of all sales. Using the solution for another project will generate more profit with this approach.

Summary

Up to this point, we have explored Python with NumPy, TensorFlow, sklearn, pandas, and matplotlib libraries. More platforms and libraries will be implemented in this book. In the months and years to come, even more languages, libraries, frameworks, and platforms will appear on the market. *However, artificial intelligence is not only about development techniques.* It is a branch of applied mathematics that requires a real-life interest in the field to deliver. Building a k-means clustering program from scratch requires careful planning. The program relies on data that is rarely available as we expect it. That's where our imagination comes in handy to find the right features for our datasets.

Once the dataset has been defined, poor conditioning can compromise the project. Some small changes in the data will lead to incorrect results.

Preparing the training dataset from scratch takes much more time than we would initially expect. Artificial intelligence was designed to make life easier but that's after a project has been successfully implemented. The problem is that building a solution requires major dataset work and constant surveillance.

Then comes the hard work of programming a k-means clustering solution that must be explained to a team. Lloyd's algorithm comes in very handy to that effect by reducing development time.

k-means clustering can be applied to many fields and architectures as will be explained in Chapter 7, *When and How to Use Artificial Intelligence*.

7

When and How to Use Artificial Intelligence

Using **artificial intelligence (AI)** in a corporate environment follows a normal process of risk assessment. Managers evaluate the feasibility of a project and return on investment before giving a green signal to an AI implementation.

This chapter builds on the k-means clustering presentation that Pert Ex, our AI expert, made in Chapter 6, *Don't Get Lost in Techniques – Focus on Optimizing Your Solutions*. The time has come to carefully deploy a large number of datasets by first showing that a classical approach will not work. Then we use the proper architecture to get the project on track.

Amazon Web Services(AWS) SageMaker and Amazon S3 provide an innovative way to train machine learning datasets. The goal here is not to suggest that AWS could be the best solution but to show the best of AWS. AWS will constantly evolve, just like Google, IBM, Microsoft, and other platforms.

However, even though machine learning platforms might differ, the implementation method will necessarily remain the same.

The following topics will be covered in this chapter:

- Proving that AI must or not be used
- Data volume issues
- Proving the NP-hard characteristic of k-means clustering
- The central limit theorem (CLT)
- Random sampling
- Monte Carlo sampling
- Stochastic sampling

- Shuffling a training dataset
- How to use AWS SageMaker for a machine learning training model

Technical requirements

You will need Python 3.6x 64-bit from <https://www.python.org/>:

```
from sklearn.cluster import KMeans
import pandas as pd
from matplotlib import pyplot as plt
from random import randint
import numpy as np
```

You will need programs from GitHub Chapter07:

- k-means_clustering_minibatch.py
- k-means_clustering_minibatch_shuffling.py

Check out the following video to see the code in action:

<https://goo.gl/1nRPH6>

Checking whether AI can be avoided

Trying to avoid AI in a book on AI may seem paradoxical. However, AI tools can be compared to having a real toolkit in real life. If you are at home and you just need to change a light bulb, you do not have to get your brand new toolkit to show off to everybody around you. You just change the light bulb and that's it.



In AI, as in real life, use the right tools at the right time. If AI is not necessary to solve a problem, do not use it.

Use a **proof of concept** (POC) approach to prove your point for an AI project. A POC should cost much less than the project itself and helps to build a team that believes in the outcome. The first step is exploring the data volume and the method that will be used.

Data volume and applying k-means clustering

Anybody who has run MNIST with 60,000 handwritten image examples on a laptop knows that it takes some time for a machine learning program to train and test these examples. Whether a machine learning program or a deep learning convolutional network is applied, it uses a lot of the local machine's resources. Even if you run it on training on **GPUs** (short for **graphics processing unit**) hoping to get better performance than with CPUs, it still takes a lot of time for the training process to run through all the learning epochs.

If you go on and you want to train your program on images, CIFAR-10, a-60,000 image subset of the tiny image dataset, will consume even more resources on your local machine.

Suppose now, in this case, Pert Ex has to start using k-means clustering in a corporation with hundreds of millions of records of data to analyze in a SQL Server or Oracle database. Pert calls you in to manage the POC phase of the project. For example, you find that you have to work for the phone operating activity of the corporation and apply a k-means clustering program to the duration of phone calls to locations around the world every morning for several weeks. That represents about several million records per day, adding up to 3 billion records in a month. Once that has been implemented, the top marketing manager wants to use the visual result of the clusters and centroids for an investment meeting at the end of the month.

You go to the IT department and ask for a server to do the k-means clustering training and simulation process. You get a flat number. Everybody knows that a machine learning k-means clustering training program running over 3 billion records will consume too much CPU/GPU. A shrewd colleague points out that 3 billion records represent only one feature. If you talk about location and duration, that represents two features, bringing the data volume to 6 billion cells in a matrix.

You clearly understand the point and agree that there is a resource problem here. You go back to the marketing manager with your "No" for the project to run on local servers. You want to obtain permission to use AWS (Amazon Web Services). Your marketing manager immediately lets you use it. Obtaining those centroids per location cluster is critical to the business plan of the company.

Proving your point

While you are gone, the IT department try to run a query on the SQL server database on one continent and the Oracle database on another. They come up with the duration per location but simply cannot provide the centroids and clusters. They are puzzled because they would like to avoid using AWS. Now that they've tried and failed, they are ready to move forward and get the project rolling. They need an explanation first.



Never force a team to do what you think is necessary without proving why it is necessary. The explanation must be clear and each part of the project must be broken down into elementary parts that everybody can understand, including yourself!

You start by explaining that solving a k-means clustering problem is **NP-hard**. The **P** stands for **polynomial** and the **N** for **non-deterministic**.

NP-hard – the meaning of P

The *P* here means that the time to solve or verify the solution of a problem is polynomial (*poly*=many, *nomial*=terms). For example, x^3 is a polynomial.

Once x is known, then x^3 will be computed. Applied to 3,000,000,000 records, an estimation can be applied:

$$x^3 = 3,000,000^3$$

It seems a lot but it isn't that much for two reasons:

- In a world of big data, the number can be subject to large-scale randomized sampling
- K-means clustering can be trained by mini-batches (subsets of the dataset) to speed up computations

The big difference between a polynomial function and an exponential function relies, in this case, on x being the variable, and 3 the exponent.

In an exponential function, x would be the variable and 3 the constant. With x as an exponent, the time to solve this problem would be exponential. It's certainly not a polynomial time you can predict.

$3^{3,000,000,000}$ is something you don't even want to think about!

Polynomial time means that this time will be more or less proportional to the size of the input. Even if the time it takes to train the k-means clustering remains a bit fuzzy, as long as the time it will take to verify the solution remains proportional to the size of the input, the problem remains a polynomial.

The IT department feels better about the problem. The volume of the training process will overload their servers. They do not have the time to purchase a server and set it up.

NP-hard – The meaning of non-deterministic

The IT department tried to find a solution with a query but did not succeed.

Non-deterministic problems require a heuristic approach, which implies some form of practical approach. The most widespread heuristic is trial and error. K-means goes beyond trial and error. It is more like a progressive-iterative approximation of the goals to attain.

In any case, k-means clustering requires a well-tuned algorithm to solve the volume of data involved in this project.

The meaning of hard

NP-hard, as opposed to NP, can solve a problem in a straightforward deterministic way, without a non-deterministic method.

The IT department is not only satisfied with the answer but also very pleased that you took the time to explain why AWS was necessary and did not just say, "The boss said so."

AI requires clear explanations and scientific proof of its necessity.



The last question comes up: *what are mini-batches and how are they used?*

Random sampling

A large portion of machine learning and deep learning contains random sampling in various forms. In this case, a training set of 3,000,000,000 elements will prove difficult to implement without random sampling.

Random sampling applies to many methods as described in this book: Monte Carlo, stochastic gradient descent, and many others. No matter what name the sampling takes, they share common concepts to various degrees depending on the size of the dataset. The following description fits the Monte Carlo philosophy.

The law of large numbers – LLN

In probability, the law of large numbers states that when dealing with very large volumes of data, significant samples can be effective enough to represent the whole set of data. We are all familiar with polling, for example, a population on all sorts of subjects.

This principle, like all principles, has its merits and limits. But whatever its limitations, this law applies to everyday machine learning algorithms.

In machine learning, sampling resembles polling. The right, smaller number of individuals makes up efficient datasets.

In machine learning, the word "mini-batch" replaces a group of people in the polling system.

Sampling mini-batches and averaging them can prove as efficient as calculating the whole dataset as long as a scientific method is applied:

- Training with mini-batches or subsets of data
- Using an estimator in one form or another to measure the progression of the training session until a goal has been reached

You may be surprised to see "until a goal has been reached" and not "the optimal solution."

The optimal solution may not represent the best solution. All the features and all the parameters are often not expressed. This makes being too precise useless.

In this case, the marketing manager wants to know where the centroids (geometric center) of each cluster (region of locations) have been computed for his marketing campaign. The corporation may decide to open a new company at the location of each centroid and provide additional services through local spin-offs to increase their market penetration.

Taking that into account, a centroid does not need to be extremely precise. Finding the nearest large city within a reasonable distance will prove good enough.

The LLN explains why random functions are widely used in machine learning and deep learning.

The central limit theorem

As shown in by the LLN applied to this business case, the k-means clustering project must provide a reasonable set of centroids and clusters (regions of locations for long-duration phone calls).

This approach can now be extended to the CLT, which states, in machine learning parlance, that when training a large dataset, a subset of mini-batch samples is sufficient. The following two conditions define the main properties of the central limit theorem:

- The variance between the data points of the subset (mini-batch) remains reasonable. In this case, filtering only long-duration calls solves the problem.
- The normal distribution pattern with mini-batch variances close to the variance of the whole dataset.

Using a Monte Carlo estimator

The name Monte Carlo comes from the casinos in Monte Carlo and gambling. Gambling represents an excellent memoryless random example. No matter what happens before a gambler plays, prior knowledge provides no insight.

Applied to machine learning, the sum of the distribution of $f(x)$ is computed. Then random samples are extracted from the following dataset: $x_1, x_2, x_3, \dots, x_n$.

No matter what form $f(x)$ takes, such as in a k-means clustering process, to estimate the average result, it can be represented by the following equation:

$$\hat{e} = \frac{1}{n} \sum_{i=1}^n f(x_i)$$

The estimator \hat{e} represents an average of the result of the k-means clustering algorithm or any model implemented.

Random sampling applications

Random sampling can be applied to many functions and methods as we will see in subsequent chapters in this book: Monte Carlo sampling, as described previously, and stochastic gradient descent by extracting samples from the dataset to evaluate the cost function and many other functions.

Randomness represents a key asset when dealing with large datasets.

Cloud solutions – AWS

Implementing large number data volumes involves using server resources in one form or the other. We have to use on-premise servers or cloud platforms. In this case, it has been decided to outsource server resources and algorithm development to face a large amount of data billion to train.

Several online cloud services offer the solution to machine learning projects. In this chapter, the project was carried out with AWS SageMaker on an AWS S3 data management platform.

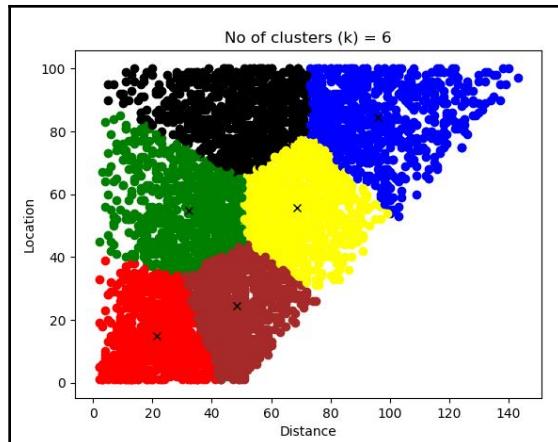
Preparing your baseline model

3,000,000 records containing two features, adding up to a 6,000,000 cell matrix dataset, require mini-batches as defined in the next section through hyperparameters.

In the AWS interface, the size of a mini-batch will be a number. To see what it does, let's apply a mini-batch to the Python program example of Chapter 6, *Don't Get Lost in Techniques – Focus on Optimizing Your Solutions*.

Training the full sample training dataset

In Chapter 6, *Don't Get Lost in Techniques – Focus on Optimizing Your Solutions*, 5,000 records and a six-cluster configuration produced six centroids (geometric centers), as shown here:



Six centroids

The problem now is how to avoid costly machine resources to train this k-means clustering dataset.

Training a random sample of the training dataset

Before spending resources on AWS, we must prepare a model locally and make sure it works. AWS possesses efficient mini-batch algorithms. But to master them, a local baseline will guarantee good AWS job management.

The `k-means_clustering_ch7_minibatch.py` file provides a way to verify the mini-batch solution.

The program begins by loading the data in the following lines of code:

```
dataset = pd.read_csv('data.csv')
print (dataset.head())
print(dataset)
```

Now a mini-batch dataset name `dataset1` will be randomly created using Monte Carlo's large data volume principle with a mini-batch size of 1,000. Many variations of the method's Monte Carlo approaches apply to machine learning. The estimator within the `k-means` function will be used as the Monte Carlo type averaging function, as shown in the following source code.

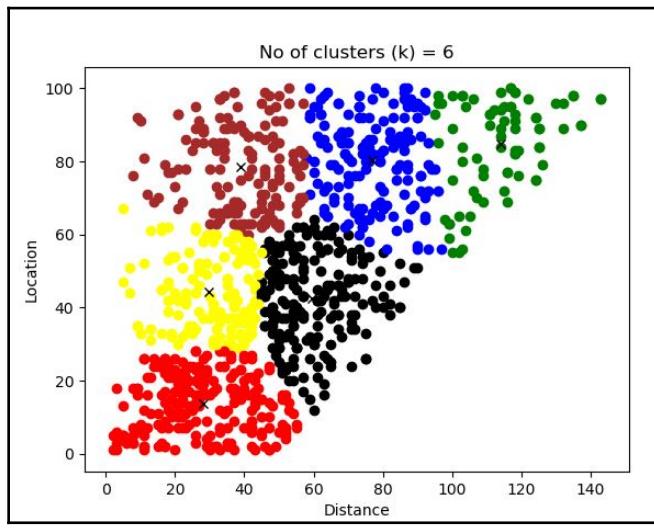
```
n=1000
dataset1=np.zeros(shape=(n, 2)) li=0
```

```
for i in range (n) :  
    j=randint (0, 4999)  
    dataset1[li][0]=dataset.iloc[j,0]  
    dataset1[li][1]=dataset.iloc[j,1]  
    li+=1
```

Finally, the k-means clustering algorithm runs on a standard basis, as shown in the following snippet.

```
#II.Hyperparameters  
# Features = 2 : implicit through the shape of the dataset (2 columns)  
k = 6  
kmeans = KMeans(n_clusters=k)  
  
#III.K-means clustering algorithm  
kmeans = kmeans.fit(dataset1) #Computing k-means clustering  
gcenters = kmeans.cluster_centers_ # the geometric centers or centroids  
print("The geometric centers or centroids:")  
print(gcenters)
```

The following screenshot, displaying the result produced, resembles the full dataset trained by k-means clustering in Chapter 6, *Don't Get Lost in Techniques – Focus on Optimizing Your Solutions*.



Output (k-means clustering)

The centroids obtained can solve the marketing problem as shown in this output.

```
The geometric centers or centroids:  
[[ 19.6626506 14.37349398]  
 [ 49.86619718 86.54225352]  
 [ 65.39306358 54.34104046]  
 [ 29.69798658 54.7852349 ]  
 [ 48.77202073 23.74611399]  
 [ 96.14124294 82.44067797]]
```

Shuffling as an alternative to random sampling

K-means clustering is an unsupervised training algorithm. As such, it trains *unlabeled* data. One random computation does not consume a high level of machine resources, but several random selections in a row can.

Shuffling can reduce machine consumption costs.

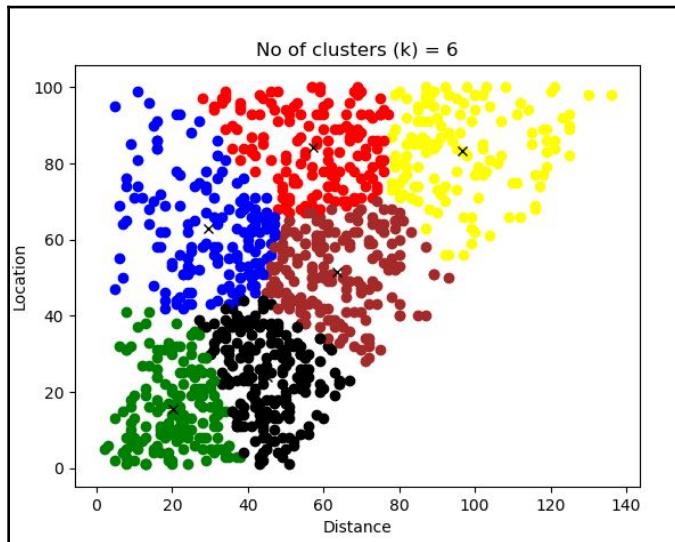
Proper shuffling of the data before starting training, just like shuffling cards before a *Poker* game, will avoid repetitive and random mini-batch computations. In this model, the loading data phase and training phase do not change. However, instead of one or several random choices for `dataset1`, the mini-batch dataset, we shuffle the complete dataset once before starting the training. The following code shows how to shuffle datasets.

```
sn=4999  
shuffled_dataset=np.zeros(shape=(sn, 2))  
for i in range (sn):  
    shuffled_dataset[i][0]=dataset.iloc[i,0]  
    shuffled_dataset[i][1]=dataset.iloc[i,1]
```

Then we select the first 1,000 shuffled records for training, as shown in the following code snippet.

```
n=1000  
dataset1=np.zeros(shape=(n, 2))  
for i in range (n):  
    dataset1[i][0]=shuffled_dataset[i,0]  
    dataset1[i][1]=shuffled_dataset[i,1]
```

The result in the following screenshot corresponds to the one with the full dataset and the random mini-batch dataset sample.



Full and random mini-batch dataset sample

The centroids produced can provide first-level results to confirm the model, as shown in the following output.

```
The geometric centers or centroids:  
[[ 29.51298701 62.77922078]  
 [ 57.07894737 84.21052632]  
 [ 20.34337349 15.48795181]  
 [ 45.19900498 23.95024876]  
 [ 96.72262774 83.27737226]  
 [ 63.54210526 51.53157895]]
```

Now that the model exists locally, the implementation on AWS can proceed.



Using shuffling instead of random mini-batches has two advantages: limiting the number of mini-batch calculations and avoiding training the same samples twice.

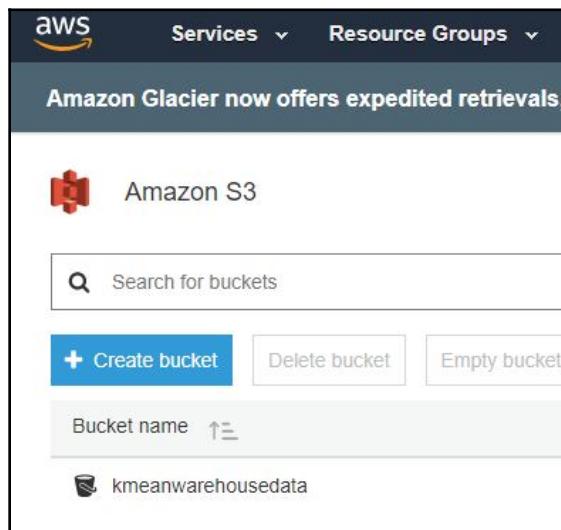
AWS – data management

No matter which cloud solution you choose, data management will involve the following:

- Security and permissions—this will depend on your security policy and the cloud platform's policies
- Managing input data and output data related to your machine learning model

Buckets

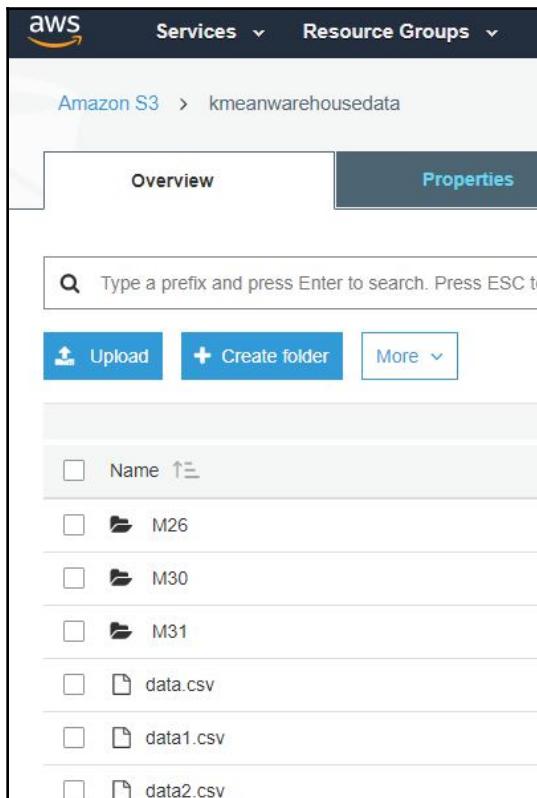
AWS has a data repository named buckets. You can create a bucket on Amazon S3. On other platforms, different names are used but the concept remains the same. Your data has to reach the platform one way or the other with an API, a data connection, or loading of the data. In the present example, the best choice remains a bucket, as shown in the following screenshot:



Amazon S3 bucket creation

Uploading files

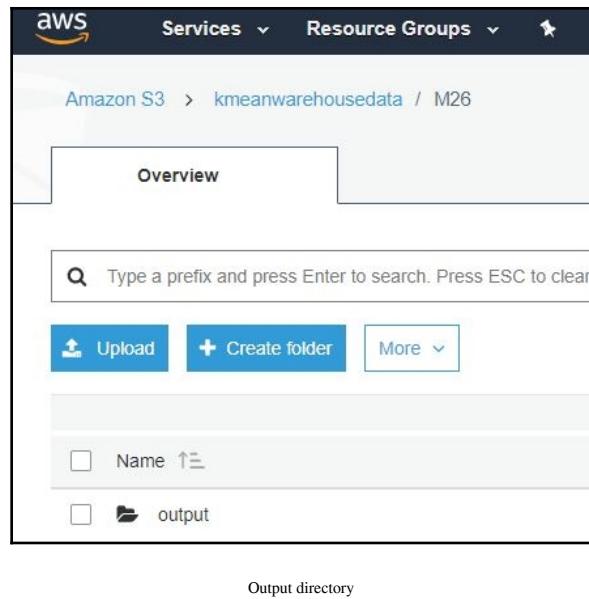
Once the bucket has been created, files can be loaded in various formats. In this example, `data.csv` was loaded without the headers. The `data.csv` used in the previous chapter has been loaded without headers. AWS requires format verification (number of columns, possible indexes) on an ever-evolving platform. The following screenshot shows how you can organize your data.



Amazon S3 bucket data

Access to output results

SageMaker automatically writes the results of the k-means clustering training session in an output directory in the following Amazon S3 bucket as shown in the following screenshot.

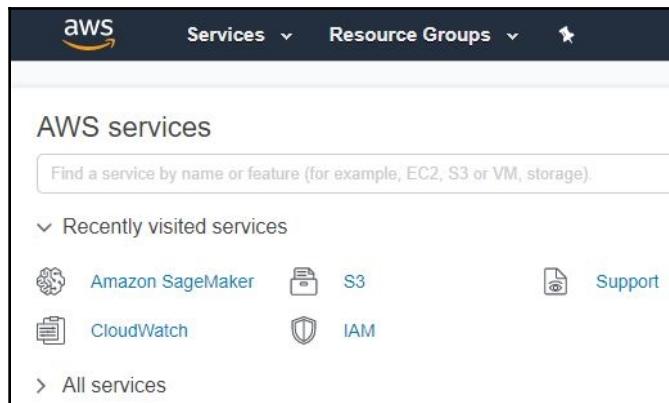


Output directory

SageMaker notebook

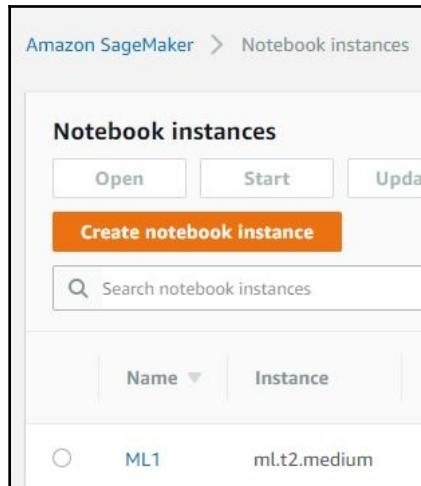
To create your machine jobs on any platform, you will have to configure an interface, use command lines, or write commands through APIs.

In this example, Amazon SageMaker notebooks will be managing the k-means clustering job. First, access AWS services with your account, as shown in this screenshot.



AWS services

Then go to SageMaker, as shown in the following screenshot.

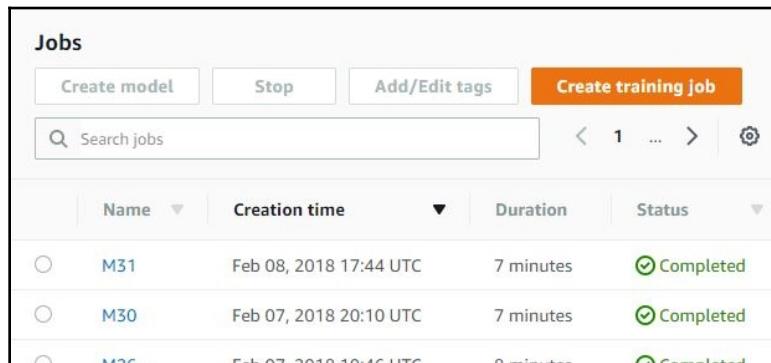


SageMaker notebook instances

ML1 is the notebook instance for the k-means clustering baseline model that will be run.

Creating a job

To create a job, we click on Create training job, as seen in the following screenshot.



Job creation

Once the input data path and output path have been created, the most important part consists of setting up the right hyperparameters (see the following screenshot) that will control the behavior of the machine learning job.

Key	Value
k	6
feature_dim	2
init_method	random ▾
mini_batch_size	1000
extra_center_factor	auto
local_lloyd_max_iter	300
local_lloyd_tol	0.0001
local_lloyd_init_method	kmeans++ ▾
local_lloyd_num_trials	auto
half_life_time_size	0
epochs	1
eval_metrics	["msd"] ▾

Setting hyperparameters

The main parameters are these:

- k: This determines the number of clusters and centroids that will be computed. For this example, setting it to 6 was used. If you are running handwritten letter recognition, you might want to have a centroid per letter.
- Feature_dim: The number of features in this example was set to 2. Many more features can be integrated into your model as long as they prove necessary.
- mini-batch_size: Mini-batch, as described in the previous section. The whole dataset batch cannot be trained with large data volumes. A batch size of 5,000 for large datasets can produce interesting results. However, finding the right parameter might take several tries.
- init_method: K-means generally starts with random ks centroids. The best way remains to get the system to work and then explore other avenues with AWS documentation.

- `extra_center_factor`: Keeping this value to auto to start with remains a good option. AWS put a lot of work into optimizing the algorithm by managing (adding centers when necessary) the number of ks. In the end, the algorithm will always return the right number of centroids.
- `local_lloyd_max_iter`: Lloyd's algorithm was described in the previous chapter. In brief, each data points goes to the closest centroid and becomes part of that cluster. The centroids are recalculated and the process continues until the data points stabilize. This parameter limits the number of iterations.
- `local_lloyd_tol`: Early stopping occurs when a local minimum is reached. The algorithm can reach stability although the optimizing process remains incomplete. The tolerance factor decides whether the variance is acceptable or not.
- `local_lloyd_num_trials`: The number of trials in a local computation measure with loss function.
- `half_life_time_size`: The k-means clustering algorithm has a loss function and evaluation metrics. As such, weights are updated as in other machine learning algorithms. Weight decay is just like when you are on a highway and have to get off at an exit. You must slow down or overshoot your exit goal. Weight decay reduces the values of weights exponentially has the loss function optimizes the clusters. At half-time, the weight will reach half of their value. You can choose not to trigger weight decay off by setting this parameter to 0.
- `epochs`: The number of episodes the algorithm will optimize the data points.
- `eval_metrics`: `["msd"]` or `["ssd"]` methods can be used as evaluation metrics. In any case, distance measurement remains a key factor in k-means optimizing algorithms.

Running a job

Running starts by clicking on **Create training job**:



Creating training job

After a few minutes or hours, depending on the volume of data provided or the complexity of the machine learning algorithm chosen, AWS SageMaker provides the status of the training session, as shown in the following screenshot.

Name	Creation time	Duration	Status
M31	Feb 08, 2018 17:44 UTC	7 minutes	Completed

Status of the training session—Completed

If the status column displays **Completed**, then the results will be analyzed. If not, we must solve the errors in the configuration process.

Reading the results

As explained previously, the results appear in the Amazon S3 bucket. In this case, a TAR file was created as shown in the following screenshot.



TAR file

The TAR file contains `model_algo-1`, a data-serialized file.

To read the file, `mxnet` can be imported, the file opened, and the result displayed in a short `read.py` program, as shown in this code.

```
import mxnet as mx
r=mx.ndarray.load('model_algo-1')
print(r)
```

The following result comes close to the local model.

```
[ [ 23.70976257 16.17502213]
[ 97.42507935 81.57003784]
[ 58.39919281 45.75806427]
[ 29.69905663 65.11740875]
[ 61.08079529 81.22251892]
[ 56.15731812 12.13902283] ]
<NDArray 6x2 @cpu(0)>]
```

Recommended strategy

Taking the data volume issue into account, data shuffling could prove effective. Then, depending on corporate practices, the data could be streamed to AWS in batches or loaded in batches.

Training will take many hours of working on the hyperparameters and result analyzing.

Summary

In any machine learning project, we should start by proving that classical programs using SQL queries will not suffice. Then, a baseline or prototype using a sample should prove that the model will work.

Using a cloud platform such as AWS SageMaker with S3 will save corporate investment resources in order to focus on the training job. This proves once again that the future of artificial intelligence experts relies upon their ability to imagine solutions and not to get lost in programming.

No matter what happens, the basic steps will always remain the same, which include proving that artificial intelligence is necessary, building a prototype model, dataset management, and hyperparameter and job optimization.

Amazon, Google, IBM, Microsoft, and others will be offering more platforms, frameworks, and resources for machine learning. These platforms will constantly evolve and change. As these giants innovate, programming will become less and less necessary so that all companies, small or large, can access the AI cloud platforms. Programming AI will remain fascinating as well as standard non-AI development. However, nothing will stop cloud ready-to-use platforms from expanding.

These platforms are the cutting edge of artificial intelligence and will lead you to the fascinating frontier of present knowledge. You can cross the border into the wild and discover new ways of solving hard problems. A cloud platform has limits that you can push much further, as we will discover in the next chapter.

8

Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies

Contrary to media hype, artificial intelligence has just begun innovating human processes. A huge amount of work remains to be done. To achieve progress, everybody must get involved, as shown in this chapter. Even if Google, Amazon, Microsoft, IBM, and others offer a solution, it does not mean it cannot be improved by third parties as add-ons to existing solutions or new standalone products. After all, once Ford invented the Model T over a hundred years ago, it did not mean it was no use inventing other better cars. On the contrary, look around you!

This chapter first explains the differences between inventions, innovations, disruption, high-priced products, revolutions, and how to innovate in AI no matter what.

Then Google Translate will be explored to illustrate innovation concepts using natural language processing. This is a prerequisite to building not only translation solutions but also chatbots (see Chapter 13, *Cognitive NLP Chatbots*, and Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*).

Google Translate entered the market in 2006 and was enhanced by neural networks in 2016, but still it often produces bad answers. Is that good news or bad news? Let's find out by exploring Google's API in a Python program, adding a k-nearest neighbor algorithm and measuring the results statistically.

The following topics will be covered in this chapter:

- The difference between inventions and innovations
- The difference between standard sales and disruptive marketing
- Google Translate API implementation in Python
- Introducing linguistics (prerequisites to building chatbots)
- Natural language processing prerequisites (for translations and chatbots)
- k-nearest neighbor (KNN) algorithm

Technical requirements

You will need Python 3.6x 64-bit from <https://www.python.org/>.

You will also need these libraries:

```
from googleapiclient.discovery import build
import html.parser as htmlparser
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
```

And you will need programs from GitHub Chapter08:

- google_translate.py
- Google_translate_a_few_test_expressions.py
- knn_polysemy.py (download v1.csv in the same directory)
- Google_Translate_customized.py (download v1.csv in the same directory)

Check out the following video to see the code in action:

<https://goo.gl/aMcfcf>

Is AI disruptive?

The word "disruptive" is more often than not associated with artificial intelligence. Media hype tells us that AI robots will soon have replaced humans around the world. Although media hype made it much easier to obtain AI budgets, we need to know where we stand if we want to implement an AI solution. If we want to innovate, we need to find the cutting edge and build new ideas from there.

Why not just plunge into the tools and see what happens? Is that a good idea or a risky one? Unlike corporations with huge budgets, a single human has limited resources. If you spend time learning in the wrong direction, it will take months to gather enough experience in another better direction to reach your goal. For example, suppose you have trouble classifying large amounts of data (see the example in *Chapter 7, When and How to Use Artificial Intelligence*); you spend months on a project on a server in your company and fail. It could cost you your job. If you find the right cloud platform as shown in *Chapter 7, When and How to Use Artificial Intelligence* and learn the key concepts, your project can take off in a few days.

Before diving into a project, find out the best way to start it.

What is new and what isn't in AI

Most AI theory and tools have been around for decades, if not centuries. *We often tend to think that since something appears new to us, it has just been invented or discovered.* This mistake can prove fatal in many projects. If you know that a theory or function has been around for decades or centuries, you can do some deep research and use solutions found 100+ years ago to solve your present problems.

Finding out what is new and what is not will make a major difference in your personal or professional AI projects.

AI is based on mathematical theories that are not new

Artificial intelligence theory presently relies heavily on applied mathematics. In *Chapter 1, Become an Adaptive Thinker*, the **Markov Decision Process (MDP)** was described. Google Brain has proudly and rightly so won some competitions using the MDP combined with neural networks.

However, Andrey Andreyevich Markov was a Russian mathematician born in 1856 who invented the MDP among other functions. Richard Bellman published an enhanced version of the Markov Decision Process in 1957.

Bellman also coined the expression "curse of dimensionality" and published books on mathematical tools widely used today in AI. It is now well known that dimensionality reduction can be performed to avoid facing thousands of dimensions (features, for example) in an algorithm.

The logistic function (see Chapter 2, *Think like a Machine*) can be traced back to Pierre François Verhulst (1844-1845), a Belgian mathematician. The logistic function uses e , the natural logarithm base, which is also named after Euler's number. Leonhard Euler (1707-1783), a Swiss mathematician who worked on this natural logarithm.

Thomas Bayes (1701-1761) invented the theorem that bears his name: Bayes' Theorem. It is widely used in artificial intelligence.

In fact, almost all of the applied mathematics in artificial intelligence, machine learning, and deep learning can be traced from 17th-century to 20th-century mathematicians.

We must look elsewhere for 21st century AI innovations.

Neural networks are not new

Neural networks as explained in Chapter 4, *Become an Unconventional Innovator*, date back to the 1940s and 1950s. Even **convolutional neural networks** (CNN) date back to the 20th century. Yann LeCun, a French computer scientist, laid down the basics of a CNN (see Chapter 9, *Getting Your Neurons to Work*) in the 1980s; he successfully applied them as we know them today in the 1990s.

We must again look elsewhere for 21st AI innovations.

Cloud server power, data volumes, and web sharing of the early 21st century started to make AI disruptive

The first sign of AI's innovative disruption appeared between 2000 and 2010. Before then, the internet existed, and servers existed. But starting from around 2005, cloud servers were made widely available. With that kind of computer power, developers around the world could try using the highly greedy resources required by machine learning and deep learning.



At the same moment, as powerful servers became available, the internet provided the largest library of knowledge in the history of mankind.

On top of that, social networking became widely used. Sharing discoveries and source code became commonplace. The **World Wide Web (WWW)** encouraged open source software, boosting local research and development.

The era of artificial intelligence became possible for local experts starting from the middle of the first decade of the 21st century.



What makes artificial appear as an innovation today is the conjunction of more powerful machines and the availability of intellectual resources.

Public awareness contributed to making AI disruptive

Public awareness of artificial intelligence remained dim for several years after the cloud architectural revolution occurred from around 1995 to 2005.

Artificial intelligence hit us hard by around 2015 when we all woke up realizing that AI could massively replace humans and create job displacement at levels never seen in human history.

Worse, we realized that machines could beat us in fields we took pride in, such as chess (*Chapter 3, Apply Machine Thinking to a Human Problem*), the game of Go, other video games, manufacturing jobs with robots, office jobs with bots, and more fields that appear every day.

For the first time in human history, the human species can be surpassed by a new species: bots. As developers, we thought we were safe until Google presented AutoML, a solution that could create machine learning solutions better than humans. At the same time, ready-to-use machine learning platforms have spread that can reduce and even replace AI software development.

Artificial intelligence forces awe and fear upon us. How far will machines go? Will we simply experience job displacement or will it go as far as species replacement?

Could this be an opportunity for many? Who knows? In any case, this chapter provides solutions to constantly innovating and being useful as humans.

Inventions versus innovations

Some artificial intelligence programs, especially deep learning algorithms, remained inventions and not innovations until they were widely used by Google and other major players.

If you have invented a better algorithm than Google for some applications, it remains an invention until it actually *changes* something in your corporation or on the Web.

Suppose you find a quicker way to recognize an image through an optimized number of neurons and a new activation function. If nobody uses it, then that invention remains a personal theoretical finding no matter how good it appears. When others begin to use this new algorithm, then it becomes an innovation.



An invention becomes an innovation only when it changes a process within a company or by a sufficient number of private users.

Revolutionary versus disruptive solutions

Suppose the new image recognition algorithm described just now becomes an innovation in a significant corporation. *This new algorithm has gone from being an invention (not really used) to an innovation (a solution making a difference).*

The corporation now widely uses the algorithm. Every subsidiary has access to it. For this corporation, the new image recognition algorithm has attained revolutionary status. Corporate sales have gone up, and profit margins have as well.

But maybe this corporation does not dominate the market and nobody has followed its example. The innovation remains revolutionary but has not become disruptive.

Then the person who created the algorithm decides to leave the company and start a business with the algorithm. It appears on GitHub as an open source program. Everybody wants it and the number of downloads increases daily until 1,000,000+ users have begun to implement it. Some very low priced add-ons are provided on the company website. Within a year, it becomes the new way of recognizing images. All companies must follow suit or stay behind. *The solution has become disruptive because it has globally changed its market.*

Where to start?

We have finished reading about such fantastic innovations and have tested mind-blowing AI solutions, knowing that all of this can be discouraging. Where to start? First of all, even if a solution exists, it has limits and can be improved, customized, packaged, and sold. *This means that if there is a limit, there is a market.*

Never criticize the flaws you find in an AI solution. They are gold mines!



Where there is a limit, there lies an opportunity.

Let's go to the cutting edge and then over the border into uncharted territory using Google Translate to illustrate this.

Discover a world of opportunities with Google Translate

Starting out with Google Translate to explore **natural language processing (NLP)** is a good way to prepare for chatbots. Any disruptive web-based solution must be able to run in at least a few languages.

Google provides many resources to use, explore, or improve Google Translate. Getting a Python code to run and then assess the quality of the results will prove vital before implementing it for crucial translations in a company.

Getting started

Go the Google's developers API Client Library page: <https://developers.google.com/api-client-library/>. On that page, you will see libraries for many languages: Java, PHP, .NET, JavaScript, Objective-C, Dart, Ruby, Node.js, Go, and probably more to come.

This does not change a thing about reaching the limits of Google Translate and entering the adventurous frontier of innovating the linguistic dimension of AI described in this chapter.

Then click on the **GET STARTED** option under **Python**. Follow the instructions to sign up, create a project in the Google API Console, and install the library.

To install the library, use pip (preferred):

```
$ pip install--upgrade google-api-python-client
```



If you encounter problems doing this part or do not wish to install anything yet, this chapter is self-contained. The source code is described in the chapter.

You are now ready to go ahead, whether you installed the tools or not.

The program

The goal of this section is to implement Google Translate functionality. You can follow the steps and get it working or simply read the chapter to get the idea.

The header

The standard Google header provided by Google was enough for the program I wrote, as shown in the following code.

```
from googleapiclient.discovery import build
```

Considering the many languages Google manages, special characters are a major problem to handle. Many forums and example programs on the Web struggle with the UTF-8 header when using Google Translate. Many solutions are suggested, such as the following source code header.

```
# -*- coding: utf-8 -*-
```

Then, when Google translate returns the result, more problems occur and many develop their own functions. They work fine but I was looking for a straightforward one-line solution. The goal here was not to have many lines of code but focus on the limit of Google Translate to discover the cutting-edge interpreting languages in AI.

So, I did not use the UTF-8 header but implemented this HTML parser.

```
import html.parser as htmlparser
parser = htmlparser.HTMLParser()
```

When confronted with a result, the following one-line HTML parser code did the job.

```
print("result:", parser.unescape(result))
```

It works well because, in fact, Google will return an HTML string or a text string depending on what option you implement. This means that the HTML parser can do the job.

Implementing Google's translation service

Google's translation service needs at least three values to return a result:

- developerKey: This is the API key obtained at the end of the "getting started" process described previously
- q="text to translate": In my code, I used source
- target="abbreviation of the translated text": en for English, fr for French, and so on

More options are available as described in the following sections.

With this in mind, the translation function will work as follows:

```
def g_translate(source,target1):  
    service = build('translate', 'v2',developerKey='your Key')  
    request = service.translations().list(q=source, target=target1)  
    response = request.execute()  
    return response['translations'][0]['translatedText']
```

In the Google_translate.py program, q and target will be sent to the function to obtain a parsed result:

```
source="your text"  
target1="abbreviation of the target language"  
result = g_translate(source,target1)  
print(result)
```

To sum up the program, let's translate Google Translate into French, which contains accents parsed by the HTML parser:

```
from googleapiclient.discovery import build  
import html.parser as htmlparser  
parser = htmlparser.HTMLParser()  
  
def g_translate(source,target1):  
    service = build('translate', 'v2',developerKey='your key')  
    request = service.translations().list(q=source, target=target1)  
    response = request.execute()  
    return response['translations'][0]['translatedText']  
  
source='Google Translate is great!'  
  
target1="fr"  
result = g_translate(source,target1)  
print(result)
```

Google_Translate.py works fine. The result will come out with the correct answer and the parsed accent:

Google Translate est génial!

At this point, everyone is happy in our corporation, named Example-Corporation:

- The developer has the starting point for whatever solution is requested
- The project manager has reached the goal set by the company
- The manager has a ready-to-use AI solution to translate commercial offers and contracts into many languages in many countries

Google Translate has satisfied everyone. It is disruptive, has changed the world, and can replace translators in many corporations for all corporate needs.

Fantastic!

In fact, we could end the chapter here, go to our favorite social network, and build some hype on our translation project!

Happy ending?

Well, not yet! The CEO, who happens to be the main shareholder of Example-Corporation, wants to make sure that everything really works before firing the 20 freelance translators working in the corporation.

The CEO phones a linguist from Example_University and asks for some expert advice. The next day, professor Ustiling comes in, gray-haired and smiling, "*You can call me Usty. So let's get to work!*"

Google Translate from a linguist's perspective

Usty is fascinated by Google Translate as presented previously. The possibilities seem endless.

Usty applies a linguist's approach to the program.



By the time this book is published, maybe Google will have improved the examples in this chapter. But don't worry; in this case, you will quickly find hundreds of other examples that are incorrect. The journey has just begun!

Playing with the tool

They open the source code and Usty adds some paragraphs. This source code is saved as `Google_translate_a_few_test_expressions.py`. Usty has spent quite some time on automatic translating as a professor.

Usty, like many linguists, wants to play with the program first to get a feel of the program. The first two examples go well from English to French. Then Usty complicates the third example, as shown in the following code.

```
source='Hello. My name is Usty!'
>>>result:Bonjour. Je m'appelle Usty

source='The weather is nice today'
>>>result: Le temps est beau aujourd'hui

source='Ce professor me cherche des poux.'
>>>result: This professor is looking for lice!
```

The first two examples look fine in French although the second translation is a bit strange. But in the third test, the expression `chercher des poux` means *looking for trouble* in English not *looking for lice* as translated into French.

Usty has a worried look on his face. The corporate project teams ask him what is wrong.

Usty starts an assessment of Google Translate.

Linguistic assessment of Google Translate

Professor Usty both wants to assess Google Translate correctly and likes limits.

Limits are the boundaries researchers crave for! We are frontiersmen!



Usty starts an expert assessment that will lead the project team to the frontier and beyond.

Lexical field theory

Lexical fields describe word fields. A word only acquires its full meanings in a context. This context often goes beyond a few other words or even a sentence.

Chercher des poux translated as such means *look for lice*. But in French, it can mean *looking for trouble* or literally *looking for lice*. The result Google Translate comes up with contains three basic problems.

```
source='chercher des poux'  
>>>result: look for lice
```

Problem 1, the lexical field: There is no way of knowing whether this means really looking for lice or looking for trouble without a context.

Problem 2, metaphors or idiomatic expressions: Suppose you have to translate *this is giving you a headache*. There is no way of knowing whether it is a physical problem or a metaphor meaning *this is driving you crazy*. These two idiomatic expressions happen to have the same metaphors when translated into French. But the *lice* metaphor in French means nothing in English.

Problem 3: *chercher* is an infinitive in French and the result should have been *looking* for lice in English. But *chercher des limites est intéressant* provides the right answer, which is *looking for*:

```
source='Chercher des limites est intéressant.'  
>>>result:Looking for boundaries is interesting.
```

The answer is correct because *is* splits the sentence into two, making it easier for Google Translate to identify *chercher* as the first part of a sentence, thus using *looking* in English.

Professor Usty looks excited by these limits but this worries the project team. Will this project be shelved by the CEO or not?

Jargon

Jargon arises when fields specialize. In AI, hidden neurons are jargon. This expression means nothing for a lawyer for example. A lawyer may think you have hidden intelligence on the subject somewhere or are hiding money in a cryptocurrency called **hidden neuron**.

In the same way, if somebody asks an artificial intelligence expert to explain how to file a motion, that would prove difficult.

In a legal corporate environment, beyond using Google Translate as a dictionary, translating sentences might be risky if only a random number of results prove to be correct.

Translating is not just translating but interpreting

Usty likes challenges. He goes online and takes a sentence from a standard description of French commercial law:

```
source='Une SAS ne dispense pas de suivre les recommandations en vigueur  
autour des pratiques commerciales.'
```

```
>>>result:An SAS does not exempt from following the recommendations in  
force around commercial practices.
```

The French sentence means that a company such as a SAS (a type of company like inc., ltd., and so on). In English, **SAS** means **Special Air Forces**. Then comes the grammar that does not sound right.

A translator would write better English and also specify what a SAS is:

"A SAS (a type of company in France) must follow the recommendations that cover commercial practices."



Translating often means interpreting, not simply translating words.

In this case, a legal translator may interpret the text in a contract and go as far as writing:

The COMPANY must respect the legal obligation to treat all customers fairly.

The legal translator will suggest that **COMPANY** be defined in the contract so as to avoid confusions such as the one Google Translate just made.

At this point, the project team is ready to give up. They are totally discouraged!



When reading about natural language processing, chatbots, and translation, everything seems easy. However, really working on Google Translate can easily turn into a nightmare!

How can they go back to their CEO with Usty and explain that Google Translate provides a random quality of results?

In fact, Google Translate is:

- Sometimes correct
- Sometimes wrong
- Sometimes partly correct and partly wrong

Usty has much more to say but the team has had enough!

However, Usty will not give up and will want to explain a global fundamental problem. Then Usty offers to provide solutions. Google Translate is worth exploring and improving because of the available online datasets provided through the API.

He tries to translate the following:

The project team is all ears

Google Translate provides the output in French:

```
source:"The project team is all ears".  
>>>result: L'équipe de projet est tout ouïe.
```

In French, as in English, it is better just to say *project team* and not use *of* to say *the team of the project*. In French *équipe projet* (*équipe* = team used before *project*).

How to check a translation

How can you check a translation if you do not know the language?



Be careful. If Google Translate provides randomly correct answers in another language, then you have no way of knowing whether the translation is reliable or not.

You may even send the opposite of what you mean to somebody important for you. You may misunderstand a sentence you are trying to understand.

In a travel company, you write an email stating that the coach stopped and people were complaining:

```
source='The coach stopped and everybody was complaining.'
```

Google Translate, for lexical field reasons, got it wrong and translated coach as a trainer (sports) in French, which would mean nothing in this context:

```
result: L'entraîneur s'est arrêté et tout le monde se plaignait.
```

Even in this situation, if you do not speak French, you have to trust professor Usty on this issue.

Now the situation gets worse. To help Google Translate, let's add some context.

```
source='The coach broke down and stopped and everybody was complaining.'
```

The answer is worse. Google Translate translates *broke down* correctly with the French expression *en panne* but still translates *coach* as *entraîneur* (trainer) in French, meaning the *trainer* broke down, not the *coach* (bus).

```
result: L'entraîneur est tombé en panne et s'est arrêté et tout le monde se plaignait.
```

Google will no doubt improve the program as it has done since 2006. Maybe this will be achieved even by the time you read this book or with some workarounds as shown in the next section. But a human translator will find hundreds of expressions Google Translate cannot deal with yet.



Understanding a sentence in your native language can prove difficult when a word or an expression has several meanings. Adding a translation function to the issue makes it even more difficult to provide a reliable answer.

And that is where we reach the frontier, just beyond the cutting edge.

As a researcher, Usty seems overexcited in contrast to the look of despair on the project team's faces. But soon the project team will share the excitement of discovering new AI lands.

AI as a new frontier

Google has a great but very limited translation program. Use the flaws to innovate! AI research and development has just scratched the surface of the innovations to come.



First, implement an AI solution. Then use it for what it is. But don't accept its limits. Don't be negative about it. Innovate! Imagine ideas or listen to other ideas you like, and build solutions in a team! Google might even publish your solutions!

Usty's excitement is contagious. The project team would like to build a prototype program. Usty suggests forgetting about generalizing the solution for the moment and just focusing on customizing Google Translation for their corporation. Usty decides to dive directly with the team into a prototype Python program: `Google_Translate_Customized.py`.

Lexical field and polysemy

A **lexical field** contains words that form sets and subsets. They differ from one language to another. A language itself forms a set and contain subsets of lexical fields.

Colder countries have more words describing water in its frozen form than tropical countries where snow hardly ever falls. A lexical field of cold could be a subset of C:

$C = \{ice, hail, snowflakes, snowman, slushy, powder, flake, snowball, blizzard, melting, crunch....n\}$

The curse of dimensionality, well known in artificial intelligence, applies here. Words contain an incredible number of dimensions and definitions. To translate certain expressions, Google Translate suppresses their dimension and reduces them.



Google Translate often uses n-grams to translate. An n-gram is a fixed-length sequence of tokens. Tokens can be a word, a character or even numerical representations of words and characters.

The probability that token n means something is calculated given the preceding/following $n - x$ or $n + x$ tokens. x is a variable depending on the algorithm applied.

For example, slushy has a special meaning in the expression slushy snow. The snow is partly melting, it's watery and making slush sound when we walk through it. Melting is only a component of the meaning of slush.

Google Translate, at this point, will only translate slushy snow in French by:

neige (snow) fondante (melting)

Google Translate will also translate melting snow by:

neige (snow) fondante (melting)

To translate slushy into French, you have to use a phrase. To find that phrase, you need some imagination or have some parsed (searched) novels or other higher levels of speech representations. That takes time and resources. It will most probably take years before Google Translate reaches an acceptable native level in all the languages publicized.

Another dimension to take into account is polysemy.



Polysemy means a word can have several very different meanings in a language. The equivalent word in another language may simply have one meaning or other, very different meanings.

"Go + over" in English can mean *go over a bridge* or *go over some notes*. At this point (hopefully, it will improve by the time you read this book), it is translated in both cases in French by *aller sur*. This means *to go on* (not over), which is incorrect in both cases.

Prepositions in English constitute a field in themselves, generating many meanings with the same word. The verb go can have a wide list of meanings: go up (upstairs), go up (stock market), go down (downstairs), go down (fall apart), and many more possibilities.

Linguistics involves many more concepts and techniques than just statistics. More will be described in Chapter 13, *Cognitive NLP Chatbots*, and Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*.

The prototype customized program starts with defining x, a small dataset to translate that will be more than enough to get things going:

```
X=['Eating fatty food can be unhealthy.',  
'This was a catch-22 situation.',  
'She would not lend me her tote bag',  
'He had a chip on her shoulder',  
'The market was bearish yesterday',  
'That was definitely wrong',  
'The project was compromised but he pulled a rabbit out of his hat',  
'So just let the chips fall where they may',  
'She went the extra mile to satisfy the customer',  
'She bailed out when it became unbearable',  
'The term person includes one or more individuals, labor unions,  
partnerships, associations, corporations, legal representatives, mutual  
companies, joint-stock companies, trusts, unincorporated organizations,  
trustees, trustees in bankruptcy, or receivers.',  
'The coach broke down, stopped and everybody was complaining']
```

These sentences will be automatically translated by Google Translate.

x1 , as implemented in the following code, defines some keywords statistically related to the sentences; it applies the n-gram probability theory described previously.

```
x1=['grasse',
'insoluble',
'sac',
'agressif',
'marché',
'certainement',
'chapeau',
'advienne',
'supplémentaire',
'parti',
'personne',
'panne']
```

Each x1 line fits the corresponding x line. As explained, this only remains a probability and may not be correct.

We are not seeking perfection at this point but an improvement.

Exploring the frontier – the program

Now it's time to add some customized novelties. The use of the vectors in this section will be explained in the next section, again through the source code that uses them.

A trigger vector will force the program to try an alternate method to translate a mistranslated sentence. When the sentence has been identified and if its value in x2 is equal to 1, it triggers a deeper translation function as implemented here:

```
x2=[0,0,0,1,0,0,0,0,0,0,1]
```

0 and 1 are flags. Each value represents a line in x.



Note for developers: To use this method correctly, all the values of this vector should be set to 1. That will automatically use several alternate methods to translate Google Translate errors. A lot of work remains to be done here.

The case study is a transportation business. A transport phrase dictionary should be built. In this case, a general phrase_translate dictionary has been implemented with one expression, as shown in the following array.

```
phrase_translation=['','','','Il est agressif','','','','','','','','','','']
```

What remains to be done to fill up this dictionary ?

- Scan all the documents of the company—emails, letters, contracts, and every form of written documents.
- Store the words and sentences.
- Run an embedding Word2Vector program (see Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*); it will find correlations between words. Add an expression correlation program to this search.
- Train the team to use the program to improve it by providing feedback (the right answer) in a learning interface when the system returns incorrect answers.

What Google Translate cannot do on a global scale, you can implement at a local scale to greatly improve the system.



k-nearest neighbor algorithm

No matter how you address a linguistic problem, it will always boil down to the word "context". When somebody does not understand somebody else, they say, *you took my words out of their context or that is not what I meant; let me explain...*



In Chapter 16, *Improve the Emotional Intelligence Deficiencies of Chatbots*, the embedding Word2Vector method will be explored. In any case, "proximity" and "nearness" remain the key concepts.

As explained before, in many cases, you cannot translate a word or expression without a lexical field. The difficulty remains proportional to the polysemy property as the program will show.

Using the **k-nearest neighbor** (KNN) algorithm as a classification method can prove extremely useful. In fact, any language interpretation (translation or chatbot) will have to use a context-orientated algorithm.

By finding the words closest (neighbors) to each other, KNN will create the lexical fields required to interpret a language. Even better, when provided with the proper datasets, it will solve the polysemy problem as shown in the upcoming sections.

The KNN algorithm

Generally, a word requires a context to mean something. Looking for "neighbors" close by provides an efficient way to determine where the word belongs.

KNN is supervised because it uses the labels of the data provided to train its algorithm. KNN, in this case, is used for classification purposes. For a given point p , KNN will calculate the distances to all other points. Then k represents the k -nearest neighbors to take into account.

Let's clear this up through an example. In English, the word "coach" can mean a trainer on a football field, a bus, or a railroad passenger car. Since the prototype is for a transportation company, "coach" will mostly be a bus that should not be confused with a trainer:

- **Step 1:** Parsing (examining in a detailed manner) texts with "coach" as a bus and "coach" as a trainer. Thus the program is searching for three target words: trainer, bus, and coach.
- **Step 2:** Finding some words that appear close to the target words we are searching. As recommended do the following:
 - Parse all the company documents you can use with a standard program
 - Use a Python function such as `if(ngram in the source) then store the data`

In this case, the `V1.csv` shown in the following output excerpt provided with the source code contains the result of such a parsing function:

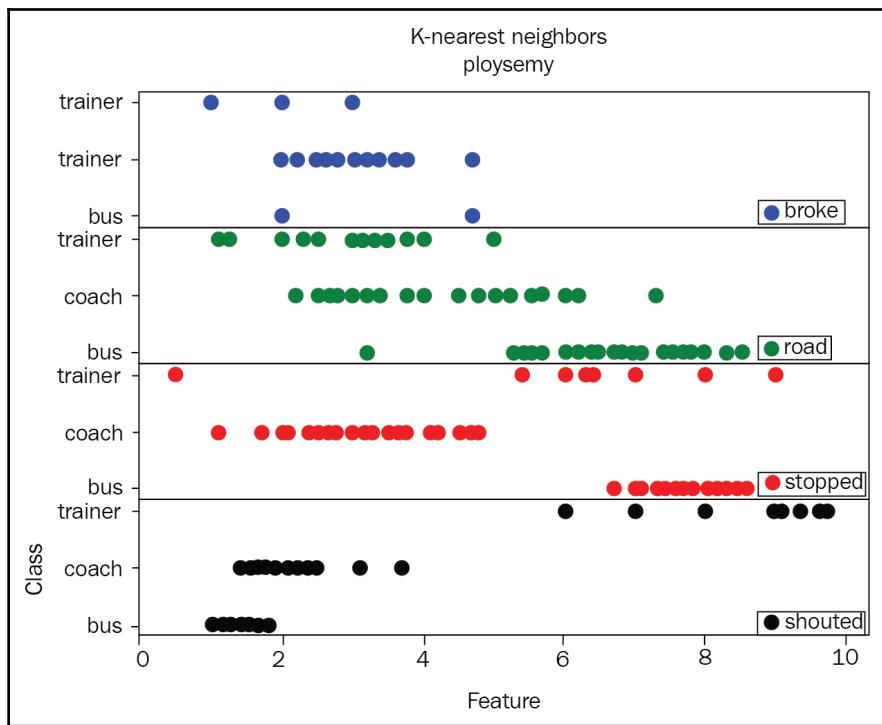
```
broke,road,stopped,shouted,class
1,3.5,6.4,9,trainer
1,3.0,5.4,9,trainer
1,3.2,6.3,9,trainer
...
6.4,6.2,9.5,1.5,bus
2,3.2,9,1,bus
6.4,6.2,9.5,1.5,bus
...
3.3,7.3,3.0,2.5,coach
4.7,5.7,3.1,3.7,coach
2.0,6.0,2.7,3.1,coach
```

The program parsed emails, documents, and contracts. Each line represents the result of parsing of one document. The numbers represent the occurrences (number of times the word was present). The numbers have been "squashed" (divided again and again) to remain small and manageable.

Progressively, the words that came out with "trainer" were "shouted" more than "stopped." For a bus, "broke" (broken down as in breaking down), "road" and "stopped" appeared more than "shout."

"Coach" appeared on an average of "shouted", "stopped", "road" and broke because it could be either a trainer or a bus, hence the problem we face when translating this word. The polysemy (several meanings) of "coach" can lead to poor translations.

The KNN algorithm loaded the v1.csv file that contains the data to be trained and finds the following result:



The knn_polysemy.py program determined the following:

- The verb "broke" in blue has a better chance of applying to a bus (x axis value >6) than to a trainer (x axis value <4). However, "coach" remains above trainer and below "bus" because it can be both.
- The word "road" follows the same logic as the blue chart.
- The verb "stopped" can apply to a trainer and more to a bus. "Coach" remains undecided again.

-
- The verb "shouted" applies clearly to a trainer more than a bus. "Coach" remains undecided again.

Note that the coordinates of each point in these charts are as follows:

- **y axis:** bus = 1, coach=2, trainer=3
- **x axis:** The value represents the "squashed" occurrence (the number of times the word appeared) values.

This is the result of the search of those words in many sources.

When a new point, a data point named P_n , is introduced in the system, it will find its nearest neighbor(s) depending on the value of k .

The KNN algorithm will calculate the Euclidean distance between P_n and all the other points from P_1 to P_{n-1} using the Euclidean distance formula. The k in KNN represents the number of "nearest neighbors" the calculation will take into account for classification purposes. The Euclidean distance (d_1) between two given points, for example, between $P_n(x_1, y_1)$ and $P_1(x_2, y_2)$, is:

$$d_1(P_n, P_1) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Considering the number of distances to calculate, a function such as the one provided by `sklearn.neighbors` proves necessary.

The knn_polysemy.py program

The program imports the `V1.csv` file described before, prints a few lines, and prepares the labels in the right arrays in their respective x axis and y axis, as shown in this source code example.

```
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
# Import data
df = pd.read_csv('V1.csv')
print (df.head())
# Classification labels
X = df.loc[:, 'broke':'shouted']
Y = df.loc[:, 'class']
```

Then the model is trained as shown in the following code:

```
# Trains the model
knn = KNeighborsClassifier()
knn.fit(X,Y)
```

Once the model is trained, a prediction is requested and is provided by the following code.

```
# Requesting a prediction
#broke and stopped are
#activated to see the best choice of words to fit these features.
# brock and stopped were found in the sentence to be interpreted.
# In X_DL as in X, the labels are : broke, road, stopped,shouted.
X_DL = [[9,0,9,0]]
prediction = knn.predict(X_DL)
print ("The prediction is:",str(prediction).strip('[]'))
```

This is the result displayed:

```
The prediction is: 'bus'
```

The initial data is plotted for visualization purposes, as implemented in the following code.

```
#Uses the same V1.csv because the parsing has
# been checked and is reliable as "dataset lexical rule base".
df = pd.read_csv('V1.csv')
# Plotting the relation of each feature with each class
figure,(sub1,sub2,sub3,sub4)=plt.subplots(4,sharex=True,sharey=True)
plt.suptitle('k-nearest neighbors')
plt.xlabel('Feature')
plt.ylabel('Class')
X = df.loc[:, 'broke']
Y = df.loc[:, 'class']
sub1.scatter(X, Y,color='blue',label='broke')
sub1.legend(loc=4, prop={'size': 5})
sub1.set_title('Polysemy')
X = df.loc[:, 'road']
Y = df.loc[:, 'class']
sub2.scatter(X, Y,color='green',label='road')
sub2.legend(loc=4, prop={'size': 5})
X = df.loc[:, 'stopped']
Y = df.loc[:, 'class']
sub3.scatter(X, Y,color='red',label='stopped')
sub3.legend(loc=4, prop={'size': 5})
X = df.loc[:, 'shouted']
```

```
Y = df.loc[:, 'class']
sub4.scatter(X, Y, color='black', label='shouted')
sub4.legend(loc=4, prop={'size': 5})
figure.subplots_adjust(hspace=0)
plt.show()
```

A compressed version of this program has been introduced in `Google_Translate_Customized.py` as shown here:

```
def knn(polysemy, vpolysemy, begin, end):
    df = pd.read_csv(polysemy+'.csv')
    X = df.loc[:, 'broke':'shouted']
    Y = df.loc[:, 'class']
    knn = KNeighborsClassifier()
    knn.fit(X, Y)
    prediction = knn.predict(vpolysemy)
    return prediction
```

The description is as follows:

- `polysemy` is the name of the file to read because it can be any file
- `vpolysemy` is the vector that needs to be predicted
- In future, in the to do list, `begin` should replace `broke` and `end` should replace `shouted` so that the function can predict the values of any vector
- The KNN classifier is called and the prediction returned

Implementing the KNN compressed function in `Google_Translate_Customized.py`

This program requires more time and focus because of the concepts of linguistics involved. The best way to grasp the algorithm is to run it in order.

Google Translate offers two translation methods, as shown in the following code:

```
#print('Phrase-Based Machine Translation(PBMT) model:base') : #m='base'
print('Neural Machine Translation model:nmt')
```

These are explained as follows:

- **Phrase-Based Machine Translation (PBMT):** This translates the whole sequence of words. The phrase or rather phraseme (multi-word expression) is not always quite a sentence.

-
- **Neural Machine Translation (NMT):** This uses neural networks such as **Recurrent Neural Network (RNN)**, which will be detailed later in this book. This method goes beyond the phraseme and takes the whole sentence into account. On the dataset presented in this chapter, this neural network method provides slightly better results.

Both methods, though interesting, still require the type of additional algorithms, among others, presented in this chapter. As you have seen so far, the subject is extremely complex if you take the lexical fields and structures of the many languages on our planet into account.

Step 1: Translating the X dataset line by line from English into French

The following code calls the translation function:

```
for xi in range(len(X)):  
    source=X[xi]  
    target1="fr";m='nmt'  
    result = g_translate(source,target1,m)
```

The code is explained here:

- `xi` is the line number in `X`.
- `source` is the `xi` line in `X`.
- `target1` is the target language, in this case, `fr` (French).
- `m` is the method (PBT or NMT) as described previously. In this case, `nmt` is applied.
- Then the Google Translate function is called as described earlier in this chapter. The result is stored in the `result` variable.

Step 2: Back translation

How can somebody know the correctness of a translation from language L1 to language L2 if L1 is the person's native language and L2 is a language the person does not understand at all?

This is one of the reasons, among others, that translators use back translation to check translations:

Translation = Initial translation from L1 to L2

Back translation = Translation back from L2 to L1

If the initial text is not obtained, then there is probably a problem. In this case, the length of the initial sentence L1 can be compared to the length of the same sentence translated back to L1. Back translation is called by the following code:

```
back_translate=result
back_translate = g_translate(back_translate,target1,m)
print("source:",source,":",len(source))
print("result:",result)
print("target:",back_translate,":",len(back_translate))
```

Length comparison can be used to improve the algorithm :

$$\text{Length of the initial } n\text{-gram} = \text{Length of back translation}$$

If it's equal, then the translation may be correct. If not, it could be incorrect. More methods must be applied during each translation. In this case, the source (initial sentence) is compared to the back translation in the following code:

```
if(source == back_translate):
    print("true",)
    t+=1
else:
    f+=1;print("false")


- t is a true counter
- f is a false counter

```

The first line of X runs as follows:

```
source: Eating fatty food can be unhealthy. : 35
result: Manger de la nourriture grasse peut être malsain.
target: Eating fat food can be unhealthy. : 33
false
```

Eating fatty food is back translated as *eating fat food*, which is slightly wrong. Something may be wrong.

The French sentence sounds wrong too. Fatty food cannot be translated as such. Usually, the common sentence is *manger gras* meaning *eating (manger) fatty (gras)* which cannot be translated into English as such.

X[2] an X[3] come back as false as well.

At example X[4], I programmed a phrase-based translation using a trigger in the false condition in the following code.

```
else:  
f+=1;print("false")  
if(X2[xi]>0):  
DT=deeper_translate(source,xi)  
dt+=1
```

Since I did not write a complete application for this book but just some examples that can be extended in the future, I used `X2` as a trigger. If `X2[x1]>0`, then the `deeper_translate` function is activated.

Step 3: Deeper translation with phrase-based translations

`Deeper_translate` has two arguments:

- `source`: The initial sentence to translate
- `x1`: The target sentence

In this case, the problem to solve is an idiomatic expression that exists in English but does not exist in French:

```
source: He had a chip on his shoulder : 29  
result: Il avait une puce sur son épaule  
target: He had a chip on his shoulder : 29  
false
```

To have *a chip on the shoulder* means to have an issue with something or somebody. It expresses some form of tension.

Google translated *chip* by assuming computer chip, or *puce* in French, which means both *computer chip* and *flea*. The translation is meaningless.

Chip enters three categories and should be labeled as such:

- Idiomatic expression
- Jargon
- Polysemy

At this point, the following function I created simulates the phrase-based solution to implement deeper translations.

```
def deeper_translate(source, index):  
    dt=source  
    deeper_response=phrase_translation[index]  
    print("deeper translation program result:",deeper_response)
```

The `deeper_translation` looks for the translated sentence containing *chip* in the following `phrase_translation` array (list, vector, or whatever is necessary).

```
phrase_translation=['',' ',' ','Il est agressif','',' ',' ',' ',' ',' ',' ',' ',' ']
```

The final result comes out with a translation, back translation, term search, and phrase translation. The following is the result produced, with comments added here before each line:

```
Initial sentence:  
source: He had a chip on his shoulder : 29  
Wrong answer:  
result: Il avait une puce sur son épaule  
The back-translation works:  
target: He had a chip on his shoulder : 29  
term: aggressif  
false  
deeper translation program result: Il est agressif
```

The question is, where did the `term` come from?

`term` comes from `x1`, a list of keywords that should be in the translation. `x1` has been entered manually but it should be a list of possibilities resulting from a KNN search on the words in the sentence viewed as classes. This means that the sentence to be translated should have several levels of meaning, not just the literal one that is being calculated.

The actual true/false conditions contain the following deeper translation-level words to check:

```
if(source == back_translate):  
    print("true")  
    if((term not in words) and (xi!=4)):  
        t+=1  
    else:  
        f+=1;print("false")  
        if(X2[xi]>0):  
            DT=deeper_translate(source,xi)  
            dt+=1
```

In the present state of the prototype, only example four activates a phrase-based translation. Otherwise, true is accepted. If false is the case, the deeper translation is only activated for two cases in this sample code. The flag is in `x2` (0 or 1).

`Deeper_translation` is called for either the phrase-based translation (described previously) or the KNN routine, which is activated if the phrase-based translation did not work.

If the translation did not work, an n-gram is prepared for the KNN algorithm, as shown in the following code:

```
if(len(deeper_response)<=0):  
    v1=0  
    for i in range(4):  
        ngram=v1[i]  
        if(ngram in source):  
            vpolysemy[0][i]=9  
    v1=1
```

`v1[i]` contains the keywords (n-grams) described in the preceding KNN algorithm for the transport lexical field, as shown in the following code:

```
V1=['broke','road','stopped','shouted','coach','bus','car','truck','break',  
'broke','roads','stop']
```

The source (sentence to translate) is parsed for each n-gram. If the n-gram is found, the polysemy vector is activated with a nine value for that n-gram. The initial values are set to 0, as shown in the following code.

```
vpolysemy=[[0,0,0,0]]
```

The variable `v1` is activated, which informs the program that `V1.csv` must be read for this case. An unlimited number of KNN references should be automatically created as described previously in the KNN section.

In this case, only `v1` is activated. But after several months of working on the project for the company to customize their local needs, many other files should be created.

In this case, when `v1` is activated, it fills out the variables as follows.

```
if(v1>0):  
    polysemy='V1'  
    begin=str(V1[0]).strip('[]');end=str(V1[3]).strip('[]')  
    sememe=knn(polysemy,vpolysemy,begin,end)
```

- `polysemy` indicates the KNN file to open
- `begin` is the first label of the `V1` vector and `end` is the last label of the `V1` vector
- `sememe` is the prediction we expect

Now a condensed version of the KNN algorithm is called as described before for `knn.polysemy.py` is called in the following code:

```
def knn(polysemy,vpolysemy,begin,end):  
    df = pd.read_csv(polysemy+'.csv')  
    X = df.loc[:,begin:end]  
    Y = df.loc[:, 'class']  
    knn = KNeighborsClassifier()  
    knn.fit(X,Y)  
    prediction = knn.predict(vpolysemy)  
    return prediction
```

The example, in this case, is the polysemy feature of *a coach* as explained in the KNN section. The output will be produced as follows:

```
Source: The coach broke down, stopped and everybody was complaining : 59  
result: L'entraîneur est tombé en panne, s'est arrêté et tout le monde se  
plaignait  
target: The coach broke down, stopped, and everyone was complaining : 59  
term: bus  
false
```

The translation is false because Google Translate returns *trainer* instead of *bus*.

The term *bus* is identical in English and French.

The KNN routine returned *bus* in English as the correct word to use when *broke down* and *stopped* were found as shown in the KNN section.

The goal of the rest of the source code in the `deeper_translate` function is to replace *coach*—the word increasing the polysemy feature to translate—by a better word (limited polysemy) to translate: `sememe`.

The `sememe` variable is initialized by the KNN function in the following code:

```
sememe=knn(polysemy,vpolysemy,beg_in,end)
for i in range(2):
    if(V1_class[i] in source):
        replace=str(V1_class[i]).strip('[]')
        sememe=str(sememe).strip('[]')
        dtsource = source.replace(replace,sememe)
        targetl="fr";m='base'
        result = g_translate(dtsource,targetl,m)
        print('polysemy narrowed result:',result,:Now true")
```

The function replaces *coach* by *bus* found by the KNN algorithm in the English sentence and then asks Google Translate to try again. The correct answer is returned.



Instead of trying to translate a word with too many meanings (polysemy), the `deeper_translate` function first replaces the word with a better word (less polysemy). Better results will often be attained.

A frequentist error probability function is added to measure the performance, as shown in the following code:

```
def frequency_p(tnumber,cnumber):
    ff=cnumber/tnumber #frequentist interpretation and probability
    return ff
```

- `cnumber` is the number of false answers returned by Google Translate
- `tnumber` is the number of sentences translated
- `ff` gives a straightforward error (translation) probability, ETP

The function is called when a translation is false, or `f>0`, as implemented in the following code:

```
if(f>0):
    B1=frequency_p(xi+1,f) #error detection probability before deep
    translation
    B2=frequency_p(xi+1,f-dt) #error detection probability after deep
    translation
    if(f>0):
        print("ETP before DT",round(B1,2),"ETP with DT",round(B2,2))
    else:
        print('Insufficient data in probability distribution')
```

- B1 is the error (translation) probability (ETP) before the deeper_translate function is called
- B2 is the ETP after the deeper_translate function is called

At the end of the program, a summary is displayed, as shown in the following output:

```
print("-----Summary-----")
print('Neural Machine Translation model:nmt')
print('Google Translate:',"True:",t,"False:",f,'ETP',round(f/len(X),2))
print('Customized Google Translate:',"True:",t,"False:",f-
dt,'ETP',round((f-dt)/len(X),2))
a=2.5;at=t+a;af=f-a #subjective acceptance of an approximate result
print('Google Translate
acceptable:',"True:",at,"False:",af,'ETP',round(af/len(X),2))
#The error rate should decrease and be stabilized as the KNN knowledge base
increases
print('Customized Google Translate acceptable:',"True:",at,"False:",af-
dt,'ETP',round((af-dt)/len(X),2))
```

- A subjective acceptance of an approximate result has been added to increase the true probability.
- The error rate should decrease as the quality of the KNN knowledge base increases. In frequent probability theory, this means that a stabilized prediction rate should be reached.

Conclusions on the Google Translate customized experiment

The final error (translation) probability produced is interesting, as shown in the following output:

```
>>>-----Summary-----
>>Neural Machine Translation model:nmt
>>Google Translate: True: 2 False: 8 ETP 0.67
>>Customized Google Translate: True: 2 False: 7 ETP 0.58
>>Google Translate acceptable: True: 4.5 False: 5.5 ETP 0.46
>>Customized Google Translate acceptable: True: 4.5 False: 4.5 ETP 0.38
```

Even with its NMT model, Google Translate is still struggling.

This provides great opportunities for AI linguists, as shown with some of the methods presented to improve Google Translate at a local level that could even go further.

This experiment with Google Translate shows that Google has just scratched the surface of real-life translations that sound right to the native speakers that receive these translations. It would take a real company project to get this on track with a financial analysis of its profitability before consuming resources.

The disruptive revolutionary loop

As you now see, Google Translate, like all artificial intelligence solutions, has limits. Once this limit has been reached, you are at the cutting edge.



Cross the border into AI Frontierland; innovate on your own or with a team.

If you work for a corporation, you can create a revolutionary customized solution for hundreds of users. It does not have to go public. It can remain a strong asset to your company.

At some point or another, the revolutionary add-on will reach beyond the company and others will use it. It will become disruptive.

Finally, others will reach the limit of your now-disruptive solution. They will then innovate and customize it in their corporation as a revolutionary solution. This is what I call the disruptive revolutionary loop. It is challenging and exciting because it means that AI developers will not all be replaced soon by AutoAI bots!

Summary

Designing a solution does not mean it will be an invention, an innovation, revolutionary, or disruptive. But that does not really matter. What a company earns with a solution represents more than the novelty of what it sells as long as it is profitable. That is rule number #1. That said, without innovating in its market, that company will not survive through the years.

If a product requires quality for security reasons, it should remain in its invention state as long as necessary. If a product can produce sales at the low end of the market before its total completion, then the company should sell it. The company will acquire a reputation for innovation, get more money to invest, and take over the territory of its competitors.

Google Translate is a good example of disruptive marketing. As shown the theory, the model and even the cloud architecture are over 10 years old. But each time one of the hundreds of millions of users stumbles across it, it creates more disruption by hooking the user onto Google solutions. The user will come back again and again to view more advertisements, and everyone is happy!

Artificial intelligence has only begun. Google Translate has been around since 2006. However, the results still leave room for developers, linguists, and mathematicians to improve upon. Google has added a neural network, Google NMT, to improve translations by analyzing whole sentences. How long will it take to be really reliable? In the meantime, the next chapter will describe a CNN in more detail and provide ideas to help our community to move AI forward beyond the cutting edge into Frontierland. Before moving on to the next chapter, check your knowledge of some aspects of disruptive innovations.

9

Getting Your Neurons to Work

The invention of convolutional neural networks (CNNs) applied to vision represents by far one of the most innovative achievements in the history of applied mathematics. With its multiple layers (visible and hidden), CNN has brought artificial intelligence from machine learning to deep learning.

A CNN relies on two basic tools of linear algebra: kernels and applying them to convolutions as described in this chapter. These tools have been used in applied mathematics for decades.

However, it took the incredible imagination of Yan LeCunn, Yoshua Bengio, and others—who built a mathematical model of several layers—to solve real-life problems with CNNs.

This chapter describes the marvels of CNNs, one of the pillars of Artificial Neural Networks (ANNs). A CNN will be built from scratch, trained, and saved. The classification model described will detect production failures on a food-processing production line.

A Python Keras program running with TensorFlow on the backend will be built layer by layer and trained. Additional sample programs will illustrate key functions.

The following topics will be covered in this chapter:

- The differences between 1D, 2D, and 3D CNNs
- Adding layers to a convolutional neural network
- Kernels and filters
- Shaping images
- ReLU activation function
- Kernel initialization
- Pooling
- Flattening
- Dense layers

- Compiling the model
- Cross-entropy loss function
- Adam optimizer
- Training the model
- Saving the model
- Visualizing the PNG of a model

Technical requirements

You will need Python 3.6x 64-bit from <https://www.python.org/>:

You will also need the following libraries:

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.preprocessing.image import ImageDataGenerator
from keras.models import model_from_json
from keras.models import save_model
from keras.models import load_model
from keras import backend as K
from pprint import pprint

import Numpy as np

import matplotlib.image as mpimg
import scipy.ndimage.filters as filter
import matplotlib.pyplot as plt
```

Programs, GitHub Chapter09:

- CNN_STRATEGY_MODEL.py
- Edge_detection_Kernel.py
- ReLU.py

Check out the following video to see the code in action:

<https://goo.gl/cGY1wK>

Defining a CNN

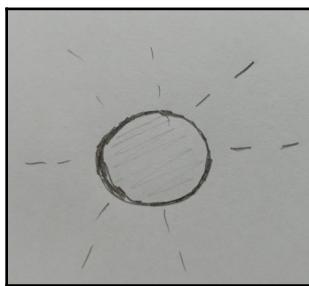
This section describes the basic components of a CNN. `01_CNN_SRATEGY_MODEL.py` will illustrate the basic CNN components the chapter used to build the case study model for a food-processing conveyor belt. CNNs constitute one of the pillars of deep learning (multiple layers and neurons).

In this chapter, a Keras neural network written in Python will be running on top of TensorFlow. If you do not have Python or do not wish to program, the chapter is self-contained with graphs and explanations.

Defining a CNN

A convolutional neural network takes an image, for example, and processes it until it can be interpreted.

For example, imagine you have to represent the sun with an ordinary pencil and a piece of paper. It is a sunny day, and the sun shines very brightly, too brightly. You put on a special pair of very dense sunglasses. Now you can look at the sun for a few seconds. You have just applied a color reduction filter, one of the first operations of a convolutional network. Then you try to draw the sun. You draw a circle and put some gray in the middle. You have just applied an edge filter. Finally, you go over the circle several times to make it easy to recognize, reducing the image you saw progressively into a representation of it. Now with the circle, some gray in the middle, and a few lines of the rays around it, anybody can see you drew a sun. You smile; you did it! You took a color image of the sun and made a mathematical representation of it as a circle, which would probably look something like this:

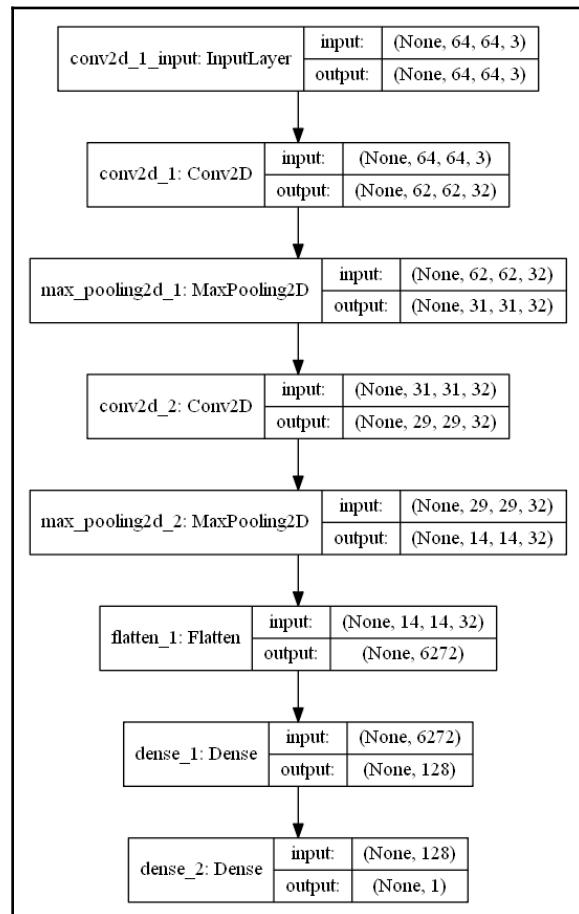


Mathematical representation of a circle

You just went through the basic processes of a convolutional network.

The word **convolutional** means that you transformed the image of the sun you were looking at into a drawing, area by area. But, you did not look at the whole sky at once. You made many eye movements to capture the sun, area by area, and you did the same when drawing. If you made a mathematical representation of the way you transformed each area from vision to your paper abstraction, it would be a kernel.

With that concept in mind, the following graph shows the successive mathematical steps to follow in this chapter's model for a machine to process an image just as you did. A convolutional network is a succession of steps that will transform what you see into a classification status. In your example, it would serve to find out whether your drawing represents the sun. This falls under the binary classification model (yes or no, 1 or 0).



Notice that the size of the outputs diminishes progressively until the outputs reach 1, the binary classification status that will return (1 or 0). These successive steps, or layers, represent what you did when you went from observing the sun to drawing it. In the end, if we draw poorly and nobody recognizes the sun, it means that we have to go back to step 1 and change some parameters (weights in this case). That way, we train to represent the sun better until somebody says, "Yes, it is a sun!" That is probability = 1. Another person may say that it is not a sun (probability = 0; thus, more training would be required).

If you carry out this experiment of drawing the sun, you will notice that, as a human, you transform one area at a time with your eye and pencil. You repeat the way you do it in each area. That mathematical repetition you perform is your kernel. Using a kernel per area is the fastest way to draw. For us humans, in fact, it is the only way we can draw. A CNN is based on this process.

This section describes each component of this model. Hundreds of different models can be implemented. However, once you understand one model, you can implement the many others.

Initializing the CNN

The model used is a Keras `sequential()` model; it requires adding one layer at a time in a certain order with TensorFlow as the backend:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from keras.layers import Convolution2D, MaxPooling2D, Activation
from keras.models import Sequential
from keras.layers import Flatten
from keras.layers import Dense
from PIL import Image
from keras.preprocessing.image import ImageDataGenerator

model = Sequential()
```

The import libraries will be used for each layer of the model. The model can now be built.

Adding a 2D convolution

In the example of a drawing (image), we used our inbuilt human spacial module, working on a two-dimensional piece of paper with (x, y) coordinates. Two-dimensional relationships can be real-life images and also many other objects as described in this chapter. This chapter describes a two-dimensional network, although others exist:

- A one-dimensional CNN mostly describes a temporal mode, for example, a sequence of sounds (phonemes = parts of words), words, numbers and any other type of sequence
- A volumetric module is a 3D convolution, such as recognizing a cube or a video

In this chapter, a spatial 2D convolution module will be applied to images of different kinds. The main program, `CNN_STRATEGY_MODEL.py`, will describe how to build and save a model.

`classifier.add` will add a layer to the model. The name **classifier** does not represent a function but simply the arbitrary name that was given to this model in this particular program. The model will end up with n layers. Look at the following line of code:

```
classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation =  
'relu'))
```

This line of code contains a lot of information: the filters (applied with kernels), the input shape, and an activation function. The function contains many more options. Once you understand these in depth, you can implement other options one by one, as you deem necessary, for each project you have to work on.

Kernel

Just to get started, intuitively let's take another everyday model. This model is a bit more mathematical and closer to CNN's kernel representation. Imagine a floor of very small square tiles in an office building. You would like each tile of the floor to be converted from dirty to clean, for example. You can imagine a cleaning machine converting 3×3 small tiles (pixels) at a time from dirty to clean. You would laugh if you saw somebody come with 50 cleaning machines to clean all of the 32×32 tiles (pixels) at the same time. You know it would consume more resources, intuitively. Not only is a kernel an efficient way to filter but also a kernel convolution is a time-saving resource process. The single cleaning machine is the kernel (dirty-to-clean filter), which will save you that time to perform the convolution (going over all of the tiles to clean, 3×3 area by 3×3 area) of transforming the floor from dirty to clean.

In this case, 32 different filters have been added with 3×3 sized kernels:

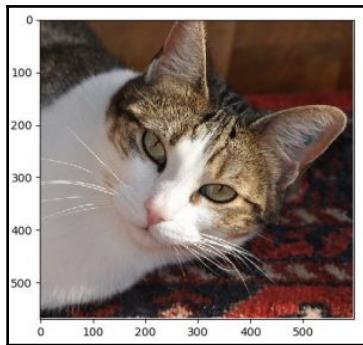
```
classifier.add(Conv2D(32, (3, 3) ...
```

The use of kernels as filters is the core of a convolutional network. $(32, (3,3))$ means (number of filters, (size of kernels)).

Intuitive approach

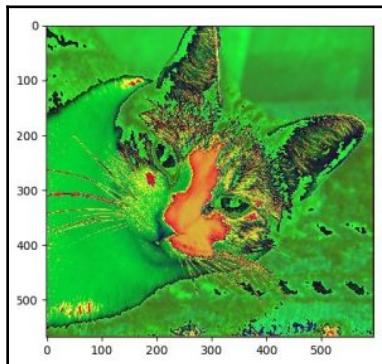
To understand a kernel intuitively, keep the sun and cleaning tiles examples in mind. In this section, a photograph of a cat will show how kernels work.

In a model analyzing cats, the initial photograph would look like this:



Cat photograph for model analyzing

On the first run of this layer, even with no training, an untrained kernel will transform the photograph.



Cat photograph transformation

The first layer already has begun isolating the features of the cat. The edges have begun to appear: the cat's body, ears, nose, and eyes. In itself, this first filter (one of 32) with a size 3 x 3 kernel—in the first layer and with no training—already produces effective results.

Each subsequent layer will make the features stand out much better with smaller and smaller matrices and vectors until the program obtains a clear mathematical representation.

Developers' approach

Developers like to see the result first to decide how to approach a problem.

Let's take a quick, tangible shortcut to understand kernels through `Edge_detection_Kernel.py` with an edge detection kernel.

```
#I.An edge detection kernel
kernel_edge_detection = np.array([[0.,1.,0.],
[1.,-4.,1.],
[0.,1.,0.]])
```

The Kernel is a 3 x 3 matrix, like the cat example. But the values are preset, and not trained with weights. There is no learning here; only a matrix needs to be applied. The major difference with a CNN network is that it will learn how to optimize kernels itself through weights and biases.

`imp.bmp` is loaded, and the 3 x 3 matrix is applied to the pixels of the loaded image, area by area:

```
#II.Load image and convolution
image=mpimg.imread('img.bmp')[:, :, 0]
shape = image.shape
```

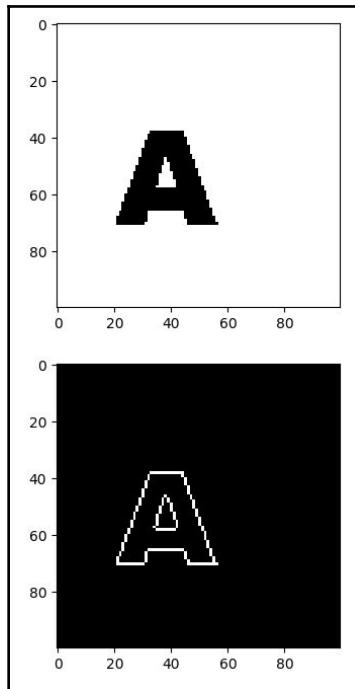
The image before the convolution applying the kernel is the letter A (letter recognition):



Now the convolution transforms the image, as shown in the following code:

```
#III.Convolution
image_after_kernel =
filter.convolve(image,kernel_edge_detection,mode='constant', cval=0)
```

The edges of A now appear clearly in white, as shown in the following graph:



White edge of A is visible

The original image on top displayed a very thick A. In the preceding graph it displays a thin identifiable A feature through thin edges that a neural network can classify within a few mathematical operations. The first layers of a convolutional network train to find the right weights to generate the right kernel automatically.

Mathematical approach

The initial image has a set of values you can display, as follows:

```
#II.Load image  
image=mpimg.imread('img.bmp')[:, :, 0]  
shape = image.shape  
print("image shape", shape)
```

The code will print a numerical output of the image as follows:

```
image shape (100, 100)
image before convolution
[[255 255 255 ..., 255 255 255]
 [255 255 255 ..., 255 255 255]
 [255 255 255 ..., 255 255 255]
 ...
]
```

The convolution filter is applied using `filter.convolve`, a mathematical function, to transform the image and filter it.

The convolution filter function uses several variables:

- The spatial index for the 3×3 kernel to apply; in this case, it must know how to access the data. This is performed through a spatial index, j , which manages data in grids. Databases also use spatial indexes to access data. The axes of those grids determine the density of a spatial index. Kernels and the image are convolved using j over W , the weights kernel.
- W is the weights kernel.
- I is the input image.
- k is the coordinate of the center of W . The default value is 0 in this case.

These variables then enter the `filter.convolve` function as represented by the following equation:

$$C_i = \sum_j I_{i+j-k} W_j$$

A CNN relies on kernels. Take all the time you need to explore convolutions through the three dimensions required to master AI: intuitive approach, development testing, and mathematical representation.

Shape

Shape defines the size of the image, which is 64×64 pixels (height \times width), as shown here.

```
classifier.add(...input_shape = (64, 64, 3)...)
```

3 indicates the number of channels. In this case, 3 indicates the three parameters of an RGB color. Each number can have a given value of 0 to 255 .

ReLU

Activation functions provide useful ways to influence the transformation of the input data weight bias calculations. Their output will change the course of a classification, a prediction, or whatever goal the network was built for. This model applies **rectified linear unit (ReLU)**, as shown in the following code:

```
classifier.add(..., activation = 'relu'))
```

ReLU activation functions apply variations of the following function to an input value:

$$f(x) = \max\{0, x\}$$

The function returns 0 for negative values; it returns the positive values as x ; it returns 0 for 0 values.

ReLU appears to be easier to optimize. Half of the domain of the function will return zeros. This means that when you provide positive values, the derivative will always be 1. This avoids the squashing effect of the logistic sigmoid function, for example. However, the decision to use one activation function rather than another will depend on the goal of each ANN model.

In mathematical terms, a rectified linear unit function will take all the negative values and apply 0 to them. And all the positive values remain unchanged.

The `ReLU.py` program provides some functions including a NumPy function to test how ReLU works.

You can enter test values or use the ones in the source code:

```
import numpy as np
nx=-3
px=1
```

`nx` expects a negative value and `px` expects a positive value for testing purposes for the `relu(x)` and `lrelu(x)` functions. Use the `f(x)` function if you wish to include zeros in your testing session.

The `relu(x)` function will calculate the ReLU value:

```
def relu(x):
    if(x<0):ReLU=0
    if(x>0):ReLU=1
    return ReLU
```

In this case, the program will return the following result:

```
negative x= -3 positive x= 5
ReLU nx= 0
ReLU px= 5
```

The result of a negative value becomes 0, and the positive value remains unchanged. The derivative or slope is thus always one, which is practical in many cases and provides good visibility when debugging a CNN or any other ANN.

The NumPy function defined as follows will provide the same results.

```
def f(x):
    vfx=np.maximum(0,x)
    return vfx
```

Through trial and error, ANN research has come up with several variations of ReLU.

One important example occurs when many input values are negative. ReLU will constantly produce zeros, making gradient descent difficult if not impossible.

A clever solution was found using a leaky ReLU. A leaky ReLU does not return a zero for a negative value but a small value you can choose, 0.1 instead of 0, for example. See the following equation:

$$f(x) = \max\{0.1, x\}$$

Now gradient descent will work fine. In the sample code, the function is implemented as follows:

```
def lrelu(x):
    if(x<0):lReLU=0.01
    if(x>0):lReLU=x
    return lReLU
```

Although many other variations of ReLUs exist, with this in mind, you have an idea of what it does.

Enter some values of your own, and the program will display the results as shown here:

```
print("negative x=",nx,"positive x=",px)
print("ReLU nx=",relu(nx))
print("ReLU px=",relu(px))
print("Leaky ReLU nx=",lrelu(nx))
print("f(nx) ReLu=",f(nx))
print("f(px) ReLu=",f(px))
print("f(0) :",f(0))
```

The results will display the ReLU results as follows:

```
negative x= -3 positive x= 5
ReLU nx= 0
ReLU px= 5
Leaky ReLU nx= 0.01
```

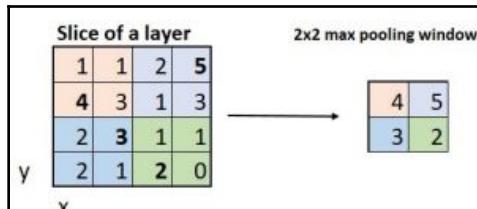
Pooling

Pooling reduces the size of an input representation, in this case, an image. Max pooling consists of applying a max pooling window to a layer of the image:

```
classifier.add(MaxPooling2D(pool_size = (2, 2)))
```

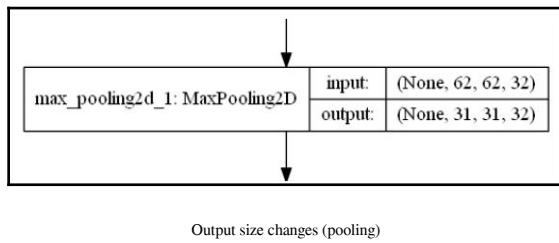
This pool_size 2 x 2 window will first find the maximum value of the 2 x 2 matrix on the top left of the image matrix. This first maximum value is 4. It is thus the first value of the pooling window on the right.

Then the max pooling window hops over two squares and finds that 5 is the highest value. 5 is written in the max pooling window. The hop action is called a **stride**. A stride value of 2 will avoid overlapping although some CNN models have strides that overlap. It all depends on your goal. Look at the following diagram:



Pooling example

The output size has now gone from a $62 \times 62 \times 32$ (number of filters) to a $31 \times 31 \times 32$, as shown in the following diagram:



Other pooling methods exist, such as average pooling, which will use the average of the pooling window and not the maximum value. This depends on the model and shows the hard work to put in to train a model.

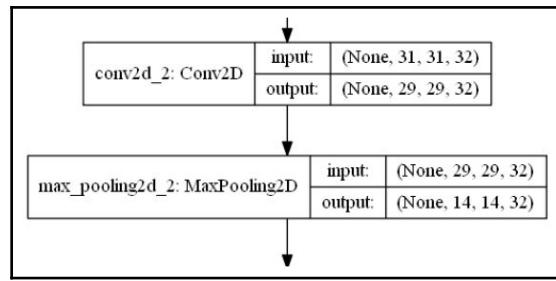
Next convolution and pooling layer

The next two layers of the CNN repeat the same method as the first two described previously, and it is implemented as follows in the source code:

```
# Adding a second convolutional layer
print("Step 3a Convolution")
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
imp=classifier.input
outputs = [layer.output for layer in classifier.layers]

print("Step 3b Pooling")
classifier.add(MaxPooling2D(pool_size = (2, 2)))
```

These two layers have drastically downsized the input to $14 \times 14 \times 32$, as shown in this diagram:



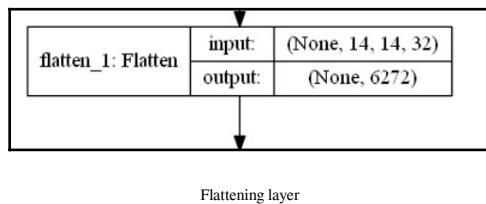
The next layer can now flatten the output.

Flattening

The flattening layer takes the output of the max pooling layer and transforms into a vector of size $x * y * z$ in the following code:

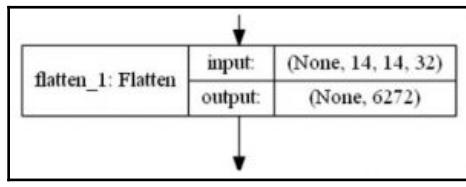
```
# Step 4 - Flattening
print("Step 4 Flattening")
classifier.add(Flatten())
```

In this case, the layer vector will be $14 * 14 * 32 = 6,272$, as shown in the following diagram:



Flattening layer

This operation creates a standard layer with 6,272 very practical connections for the dense operations that follow. After flattening has been carried out, a fully connected, dense network can be implemented.

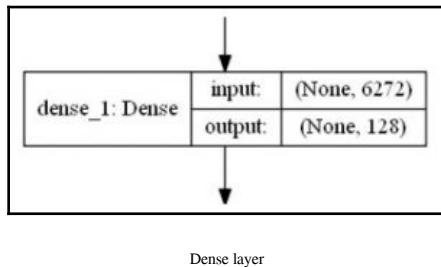


Flattening layer

Dense layers

Dense layers are fully connected. This has been made possible by the reductions in size calculated up to here, as shown before.

The successive layers in this sequential model have brought the size of the image down enough to use dense layers to finish the job. Dense 1 comes first, as shown here:



The flattening layer produced $14 \times 14 \times 32$ size 6,272 layer with a weight for each input. If it had not gone through the previous layers, the flattening would have produced a much larger layer, and the features would not have been extracted by the filters. The result would produce nothing effective.

With the main features extracted by the filters through successive layers and size reduction, the dense operations will lead directly to a prediction using the ReLu on the dense 1 operation and the logistic sigmoid function on the dense operation.

Dense activation functions

This is how to apply the ReLU activation function.

The domain of the ReLU activation function is applied to the result of the first dense operation. The ReLU activation function will output the initial input for values ≥ 0 and will output 0 for values < 0 :

$$f(\text{input value}) = \max\{0, \text{input_value}\}$$

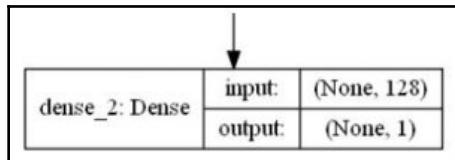
This is how to apply the logistic sigmoid activation function.

The logistic activation function applied to the second dense operation, as described in Chapter 2, *Think like a Machine*, will produce a value between 0 and 1:

$$LS(x)=\{0,1\}$$

We will now discuss the last dense layer after the LS activation function.

The last dense layer is size 1 and will classify the initial input, an image in this case:



Dense layer 2

The layers of the model have now been added. Training can begin.

Training a CNN model

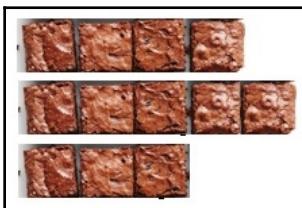
Training a CNN model involves four phases: compiling the model, loading the training data, loading the test data, and running the model through epochs of loss evaluation and parameter-updating cycles.

In this section, the choice of the theme of the training dataset will be a real-life case study from the food-processing industry.

The goal

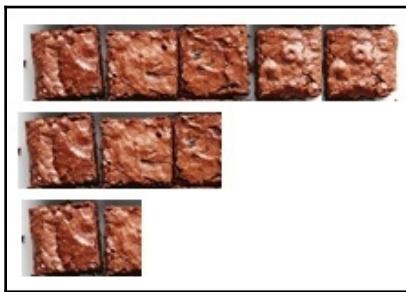
The primary goal of this model consists of detecting production efficiency flaws on a food-processing conveyor belt. The use of CIFAR-10 (images) and MNIST (handwritten digit base) proves useful to understand and train some models (see Chapter 10, *Applying Biomimicking to Artificial Intelligence*). However, at one point, real-life datasets must be used to sell and implement deep learning and artificial intelligence in general.

The following photograph shows a section of the conveyor belt that contains an acceptable level of products, in this case, portions of chocolate cakes:



Portions of chocolate cake example

However, sometimes the production slows down, and the output goes down to an alert level, as shown in the following photograph:



Portions of chocolate cake example

The alert-level image shows a gap that will slow down the packaging section of the factory dramatically.

Chapter 10, *Applying Biomimicking to Artificial Intelligence*, describes how this model can apply to the food-processing industry and be transferred to other applications.

Compiling the model

Compiling a Keras model requires a minimum of two options: a loss function and an optimizer. You evaluate how much you are losing and then optimize your parameters, just as in real life. A metric option has been added to measure the performance of the model. With a metric, you can analyze your losses and optimize your situation, as shown in the following code:

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy',
metrics = ['accuracy'])
```

Loss function

The loss function provides information on how far the state of the model y_1 (weights, biases) is from its target state y .

A description of the quadratic loss function precedes that of the binary cross-entropy functions applied to the case study model in this chapter.

Quadratic loss function

For gradient descent, see Chapter 5, *Manage the Power of Machine Learning and Deep Learning*. Let us refresh the concept. Imagine you are on a hill and want to walk down that hill. Your goal is to get to y , the bottom of the hill. Presently you are at a . Google Maps shows you that you still have to go a certain distance:

$$y - a$$

That formula is great for the moment. But now suppose you are almost at the bottom of the hill, and the person walking in front of you dropped a coin. You have to slow down now, and Google Maps is not helping much because it doesn't display such small distances that well. You open an imaginary application called **tiny maps** that zooms into small distances with a quadratic objective (or cost) function:

$$O = (y - a)^2$$

To make it more comfortable to analyze, O is divided by 2, producing a standard quadratic cost function:

$$\text{Cost} = \frac{(y - a)^2}{2}$$

y is the goal. a is the result of the operation of applying the weights, biases, and finally the activation functions.

With the derivatives of the results, the weights and biases can be updated. In our hill example, if you move one meter (y) per step (x), that is much more than moving 0.5 meters (y) per step. Depending on your position on the hill, you can see that you cannot apply a constant learning rate (conceptually the length of your step); you adapt it just like Adam, the optimizer, does.

Binary cross-entropy

Cross-entropy comes in handy when the learning slows down. In the hill example, it slowed down at the bottom. But, remember, a path can lead you sideways, meaning you are momentarily stuck at a given height. Cross-entropy solves that by being able to function well with very small values (steps on the hill).

Suppose you have the following structure:

- Inputs is $\{x_1, x_2, \dots, x_n\}$
- Weights is $\{w_1, w_2, \dots, w_n\}$
- A bias (or sometimes more) is b
- An activation function(ReLU, logistic sigmoid, or other)

Before the activation, z represents the sum of the classical operations::

$$z = \sum_{x_i} w_i x_i + b$$

Now the activation function is applied to z to obtain the present output of the model.

$$y_1 = act(z)$$

With this in mind, the cross-entropy loss formula can be explained:

$$\text{Loss} = \frac{1}{n} \sum_x [y \log y_1 + (1 - y) \log(1 - y_1)]$$

In this function:

- n is the total number of items of the input training, with multiclass data. $M > 2$ means situations in which a separate loss of each class label is calculated. The choice of the logarithm base (2, e, 10) will produce different effects.
- y is the output goal.
- y_1 is the present value, as described previously.

This loss function is always positive; the values have a minus sign in front of them, and the function starts with a minus. The output produces small numbers that tend to zero as the system progresses.

The loss function in Keras, which uses TensorFlow, uses this basic concept with more mathematical inputs to update the parameters.

A binary cross-entropy loss function is a binomial function that will produce a probability output of 0 or 1 and not a value between 0 and 1 as in standard cross-entropy. In the binomial classification model, the output will be 0 or 1.

In this case, the sum Σ is not necessary when M (*number of classes*) = 2. The binary cross-entropy loss function is then as follows:

$$\text{Loss} = -y \log y_1 + (1 - y) \log(1 - y_1)$$

The whole concept of this loss function method is for the CNN network to be used in specific real-life case studies described in the training dataset section and Chapter 10, *Applying Biomimicking to Artificial Intelligence*.

Adam optimizer

In the hill example, you first walked with big strides down the hill using momentum (larger strides because you are going in the right direction). Then you had to take smaller steps to find the object. You are adapting your estimation of your moment to your need; hence, the name **adaptive moment estimation (Adam)**.

Adam constantly compares the mean past gradients to present gradients. In the hill example, it compares how fast you were going.

The Adam optimizer represents an alternative to the classical gradient descent method or stochastic gradient descent method (see Chapter 5, *Manage the Power of Machine Learning and Deep Learning*). Adam goes further by applying its optimizer to random (stochastic) mini-batches of the dataset. This makes it a version of stochastic gradient descent.

Then, with even more inventiveness, Adam adds **root-mean-square deviation (RMSprop)** to the process by applying per-parameter learning weights. It analyzes how fast the means of the weights are changing (such as the gradients in our hill slope example) and adapts the learning weights.

Metrics

Metrics are there to measure the performance of your model. The metric function behaves like a loss function. However, it is not used to train the model.

In this case, the accuracy parameter was this:

```
...metrics = ['accuracy'])
```

Here, a value that descends towards 0 shows whether the training is on the right track and moves up to one when the training requires Adam function optimizing to set the training on track again.

Training dataset

The training dataset is available on GitHub, along with the installation instructions and explanations. The dataset contains the photo shown previously for the food-processing conveyor belt example.

The class A directory contains the acceptable level images of a production line that is producing acceptable levels of products. The class B directory contains the alert-level images of a production line that is producing unacceptable levels of products.

The number of images of the dataset is limited because of the following:

- For the purpose of training this limited industrial model, the images produced good results
- The training-testing phase runs faster to study the program

The goal of the model is to detect the alert levels as described in more detail in Chapter 10, *Applying Biomimicking to Artificial Intelligence*, when it will be applied to the trained images and generalized to similar ones.

Data augmentation

Data augmentation increases the size of the dataset by generating distorted versions of the images provided.

The `ImageDataGenerator` function generates batches of all images found in tensor formats. It will perform data augmentation by distorting the images (shear range, for example). Data augmentation is a fast way to use the images you have and create more virtual images through distortions:

```
train_datagen = ImageDataGenerator(  
    scale = 1./255,  
    shear_range = 0.2,  
    zoom_range = 0.2,  
    horizontal_flip = True)
```

The code description is as follows:

- `scale` will rescale the input image if not 0 (or none). In this case, the data is multiplied by 1/255 before applying any other operation.
- `shear_range` will displace each value in the same direction determined, in this case by the 0.2. It will slightly distort the image at one point, giving some more virtual images to train.

- `zoom_range` is the value of zoom.
- `horizontal_flip` is set to true. This is a Boolean that randomly flips inputs horizontally.

`ImageDataGenerator` provides many more options for real-time data augmentation, such as rotation range, height shift, and more.

Loading the data

Loading the data goes through the `train_datagen` preprocessing `image` function (described previously) and is implemented in the following code:

```
print("Step 7b training set")
training_set = train_datagen.flow_from_directory(directory+'training_set',
target_size = (64, 64),
batch_size = batchs,
class_mode = 'binary')
```

The flow in this program uses the following options:

- `flow_from_directory` sets the directory + 'training_set' to the path where the two binary classes to train are stored.
- `target_size` will all be resized to that dimension. In this case, it is 64 x 64.
- `batch_size` is the size of batches of data. The default value is 32 and set to 10 in this case.
- `class_mode` determines the label arrays returned: `none` or `categorical` will be 2D one-hot encoded labels. In this case, `binary` returns 1D binary labels.

Testing dataset

The testing dataset flow follows the same structure as the training dataset flow described previously. However, for testing purposes, the task can be made easier or more difficult depending on the choice of the model. This can be done by adding images with defects or noise. This will force the system into more training and the project team into more hard work to fine-tune the model. Data augmentation provides an efficient way of producing distorted images without adding images to the dataset. Both methods, among many others, can be applied at the same time when necessary.

Data augmentation

In this model, the data only goes through rescaling. Many other options could be added to complicate the training task to avoid overfitting, for example, or simply because the dataset is small:

```
print("Step 8a test")
test_datagen = ImageDataGenerator(rescale = 1./255)
```

Loading the data

Loading the testing data remains limited to what is necessary for this model. Other options can fine-tune the task at hand:

```
print("Step 8b testing set")
test_set = test_datagen.flow_from_directory(directory+'test_set',
target_size = (64, 64),
batch_size = batchs,
class_mode = 'binary')
```

Never underestimate dataset fine-tuning. Sometimes, this phase can last weeks before finding the right dataset and arguments.

Training with the classifier

The classifier has been built and can be run:

```
print("Step 9 training")
print("Classifier",classifier.fit_generator(training_set,
steps_per_epoch = estep,
epochs = ep,
validation_data = test_set,
validation_steps = vs))
```

The `fit_generator` function, which fits the model generated batch by batch, contains the main hyperparameters to run the training session through the following arguments in this model. The hyperparameters settings determine the behavior of the training algorithm:

- `training_set` is the training set flow described previously.
- `steps_per_epoch` is the total number of steps (batches of samples) to yield from the generator. The variable used in the following code is `estep`.

- epochs is the variable of the total number of iterations made on the data input. The variable used is ep in the preceding code.
- validation_data=test_set is the testing data flow.
- validation_steps=vs is used with the generator and defines the number of batches of samples to test as defined by vs in the following code:

```
estep=100 #10000  
vs=100 #5000  
ep=2 #50
```

While the training runs, measurements are displayed: loss, accuracy, epochs, information on the structure of the layers, and the steps calculated by the algorithm.

Here is an example of the loss and accuracy data displayed:

```
Epoch 1/2  
- 23s - loss: 0.1437 - acc: 0.9400 - val_loss: 0.4083 - val_acc: 0.5000  
Epoch 2/2  
- 21s - loss: 1.9443e-06 - acc: 1.0000 - val_loss: 0.3464 - val_acc:  
0.5500
```

Saving the model

Saving the model, in this case, consists in saving three files.

First, we will discuss the model file.

model.json, saved in the following code, contains serialized data describing the model itself without weights. It contains the parameters and options of each layer. This information is very useful to fine-tune the model:

```
# serialize model to JSON  
model_json = classifier.to_json()  
with open("model.json", "w") as json_file:  
    json_file.write(model_json)
```

More details on this file are provided in the next section, in the *Loading the model* section.

- Now we will discuss the weights file.

To understand a CNN model or any other type of model, having access to the weights can provide useful information when nothing works or to solve some of the problems that come up.

The weights are saved in the following code:

```
# serialize weights to HDF5
classifier.save_weights("model.h5")
from keras.utils import plot_model
plot_model(classifier, to_file='model.png', show_shapes=True )
print("Model saved to disk")
```

- A `model.png` file

This file will display the layers of the model, the input, and output sizes. The following code saves the model in `.png` format:

```
from keras.utils import plot_model
plot_model(classifier,
to_file=directory+'model/model.png', show_shapes=True )
print("Model saved to disk")
```

`model.png` can prove useful to understand how the model went down from the number of pixels of the original image to the binary classification layer containing the classification status. The diagram at the beginning of the chapter, displaying the layers and input/output values, was produced by this program.

Next steps

The model has been built and trained. In [Chapter 10, Applying Biomimicking to artificial intelligence](#), it will be loaded and implemented in the food-processing company and other similar applications through transfer learning.

Summary

Building and training a CNN will only succeed with hard work at choosing the model, the right dataset, and hyperparameters. Convolutions, pooling, flattening, dense layers, activations, and optimizing parameters (weights and biases) form solid building blocks to train and use a model.

Training a CNN to solve an everyday industrial problem helps sell AI to a manager or a sales prospect. In this case, using the model to help a food-processing factory solve a conveyor belt productivity problem takes artificial intelligence a step further into everyday corporate life.

Saving the model provides a practical way to use it by loading it and applying it to new images to classify them. This chapter concluded after we had trained and saved the model.

Chapter 10, *Applying Biomimicking to Artificial Intelligence*, will dive deeper into how neural networks were inspired by human neural networks and describe how to analyze a neural network with TensorBoard tools in more detail. Then the saved model CNN model will be applied to the food industry through transfer learning; and we'll expand far beyond those limits through domain learning in Chapter 11, *Conceptual Representation Learning*.

10

Applying Biomimicking to Artificial Intelligence

Heated debates have been raging over biomimicking humans in various forms of artificial intelligence for decades. In the 1940s, McCulloch and Pitts (see Chapter 2, *Think like a Machine*) came up with a *human* neuron. Then Rosenblatt came up with a *human* perceptron (Chapter 4, *Become an Unconventional Innovator*). Mimicking humans seemed to have conquered the world of artificial intelligence networks. Then, in 1969, Minsky slowed research down by proving that a perceptron could not solve the XOR problem (see Chapter 4, *Become an Unconventional Innovator*).

Today, deep learning networks reproduce our mental way of thinking. Neuroscientists now work on representing our physical brain functions. These two different fields, though different, still inspire each other.

Neural networks provide good mathematical models of how our mind works on the number side. Mental images on top of words and numbers to describe them constitute the larger part of human thinking. As shown in Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations Small to Large Companies*, the uncharted territory of AI must be continuously explored. Our community needs to make progress in this direction with tools that will represent our way of thinking at higher levels.

The following topics will be covered in this chapter:

- Showing how a neural network mimics the way a human mind represents images
- Showing how to use the graph structure representation of deep learning neural networks to build and debug them
- Exploring how the mathematical model neural networks are built today with TensorBoard

- Understanding the cutting edge of the mathematical data flow graph structures such as TensorFlow, which will lead us to the cutting edge of biomimicking humans with these methods
- The difference between neuroscience computing models and deep learning models
- A TensorFlow MNIST classifier
- TensorBoard dashboards
- TensorFlow scopes, placeholders, variables, and other components
- Analyzing accuracy, cross-entropy, and weights during the training process

This chapter prepares the reader for Chapter 11, *Conceptual Representation Learning*, which goes right over the cutting edge and into the uncharted territory of **conceptual representation learning (CRL)** and meta networks.

Technical requirements

You will need Python 3x with the following libraries:

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import matplotlib.pyplot as plt
import keras
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
from keras.models import model_from_json
from keras.models import load_model
import numpy as np
import cv2
from PIL import Image

import json
from pprint import pprint
```

For the TensorFlow classifier you will need this:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

#argparse is a Python command-line parsing module
import argparse
import os
import sys
```

Programs from GitHub can be found here:

- https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter10/Google_Tensor_Classifier.py
- https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2001/Chapter10/Tensorboard_reader.py

Check out the following video to see the code in action:

<https://goo.gl/psrZzs>

Human biomimicking

In this section, the difference between neuroscience research on the brain and deep learning research on the mind has led to TensorFlow and TensorBoard. With TensorFlow, our representation of image recognition in a mathematical graph data flow structure has become an extremely efficient model. TensorBoard provides a visual representation of what we are building.

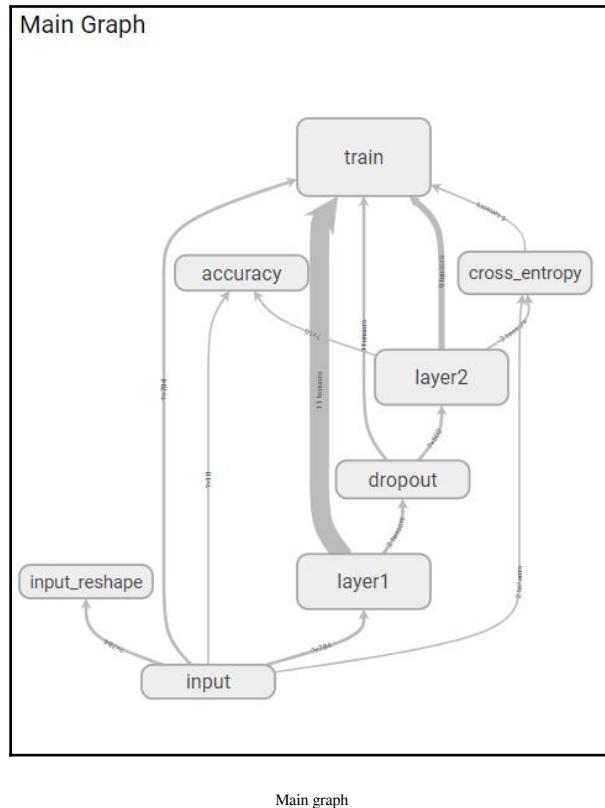
TensorFlow represents how our mind recognizes images (and other elements). TensorBoard enables our minds to visualize that representation and improve it.

TensorFlow, an open source machine learning framework

TensorFlow relies on graph representations of data flow. The graph computes dependencies between operations.

When using Keras, as in Chapter 9, *Getting Your Neurons to Work*, the details of graphs and sessions do not appear. However, to compare our brain with deep learning, some details need to be examined using `Google_Tensor_Classifier.py`, which classifies MNIST handwritten images and saves the summaries (information about the model).

The graph of the program is as shown in the following diagram:



The nodes (labels with lines around them) and edges (lines connecting the labels) constitute the graph structure. The graph structure, in turn, indicates how the operations are connected to one another.

Chapter 9, *Getting Your Neurons to Work*, described all the nodes except dropout. In that chapter, two **dense layers** were added to the classifier model. Sometimes, when constructing a dense layer, some elements will not be trained. This is done to avoid overfitting (see the transfer learning section later), which means that the model will know the training set too well and will not adapt efficiently to new objects (images, sounds, or words). This method, among others, helps to fine-tune the model during the regularization process (methods to avoid overfitting, for example). The dropout regularization rate can be added to the graph structure. Dropouts can be used several times and at different places in a given model.

Does deep learning represent our brain or our mind?

After reading Chapter 9, *Getting Your Neurons to Work*, and the beginning of this chapter, it takes a major effort to see how a data flow graph structure representing a deep learning model has something to do with our brain.

We have gone a long way since the McCulloch-Pitts neuron proudly tried to represent a neuron in our brain (see Chapter 2, *Think like a Machine*).

Neuroscience research that tries to create computer models of our brain is now named **computational neuroscience**. Computer systems represent mathematical models of the brain. Understanding how the brain works remains a critical field of physiology, nervous system, cognitive ability, and related research. It sometimes provides some ideas for deep learning networks.

But a TensorFlow data flow graph structure represents a deep learning model (or other models) of a mathematical solution to a classification or prediction problem.

Deep learning networks have moved away from neuroscience, and their source of inspiration comes mostly from other areas such as applied mathematics (linear algebra), probability, information theory (for entropy, for example), and other related fields.

Naturally, the media hype surrounding artificial intelligence seems a bit confusing because marketing managers make some impressive demonstrations of robots that look like us, try to act like us, and speak like us. No matter what those systems do right now, their learning model relies on data flow graph structures based mostly on mathematical representations of our mind, not our brain.

Our mind thinks and reasons. When we think, we do not feel our brain following through neurons and synapses. Our mind acts like a high-level API between what we want to do and what our brain does for us.

To improve our API, we create abstract representations of almost everything around us. We try to make sense of the chaos of all the incoming perceptions and thoughts we have to manage. The better our representations, the more efficient our API to our brain will become.



The TensorFlow data flow graph structure represents our mind's API relaying information seamlessly to our brain, not our brain structure which we cannot see when we think.

In the end, a deep learning network is a way of mimicking humans.



Biomimicking in deep learning simply mimics a **deep learning network** and imitates our **mathematical mind**.

A TensorBoard representation of our mind

Nobody is sure how our brains can learn quickly with a few labeled objects and then apply the patterns to unsupervised, unlabeled learning. Research on the subject will eventually produce some helpful explanations.

In the meantime, we must rely on our mind's representations to train deep learning models.



If you want to follow the explanations of this chapter through programs, first run `Google_Tensor_Classifier.py` as it is. Then run `Tensorboard_reader.py` and follow the instructions in the code to open TensorBoard in your browser. Otherwise, the chapter is self-contained with source code excerpts and screenshots.

Many of the concepts in the `TensorFlow_Classifier.py` program were explained in Chapter 9, *Getting your Neurons to Work*.

The following sections focus on TensorFlow objects that are not always perceived with high-level interfaces, such as Keras.

Input data

This program is a **unit-testing** model. It uses unit-testing data in a unit-testing deep learning model:

```
mnist =  
    input_data.read_data_sets(FLAGS.data_dir, one_hot=True, fake_data=FLAGS.fake_  
    data)
```

`fake_data` refers to an MNIST (handwritten digits) unit testing dataset and `FLAGS` are command-line parameters (in this case, path information).

`one-hot=true` means that the labels of the data must be encoded into values. The presence of labels means that this is a supervised (with labels) training classifier and not an unsupervised (training with no labels) classifier.

The one-hot function (see Chapter 2, *Think like a Machine*) will encode labels into distinct one-hot values as in the following example:

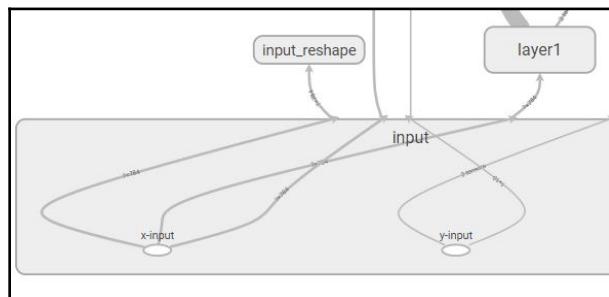
```
# [[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
# [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
# ...]
```

The inputs are stored in placeholders as in the following code:

```
with tf.name_scope('input'):  
    x = tf.placeholder(tf.float32, [None, 784], name='x-input')  
    y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')
```

Placeholders are **not** just variables that will be filled up later. They define one of the fundamental building blocks of a TensorFlow data flow graph. They contain the methods that will be used in the **node** they are in.

`tf.name_scope` is the title label that will appear on TensorBoard, and `name` identifies this element of the graph. The following graph displays the input scope with its two inputs.



Input scope with two inputs

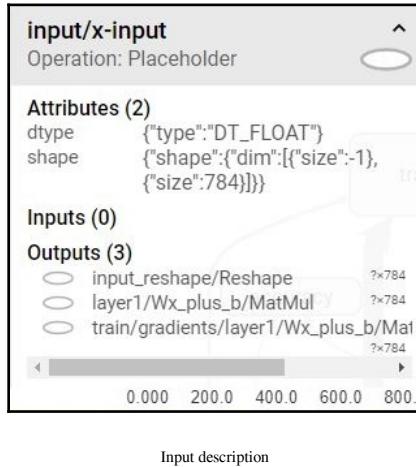
TensorBoard provides the visual representation our mind needs to think. The placeholders are in nodes that are linked to other operations through edges (lines) that represent the flow of data and operations.

`y-input`, as you can see, skips the `input_reshape` function in layer 1; it goes straight to the deeper operations of cross-entropy and accuracy as explained later.

`x-input` goes through the `input-reshape` node in the following code:

```
with tf.name_scope('input_reshape'):  
    image_shaped_input = tf.reshape(x, [-1, 28, 28, 1])  
    tf.summary.image('input', image_shaped_input, 10)
```

`tf.name_scope` defines the title, `input_reshape`. The interesting part is the details you will see if you click on the node and drill down, as shown in the following screenshot:



Input description

We usually represent the `reshape` function as one channel output when in fact there are three outputs to subsequent operations.

The **IMAGES** section provides a visual representation of the reshaped images as shown in the following screenshot:

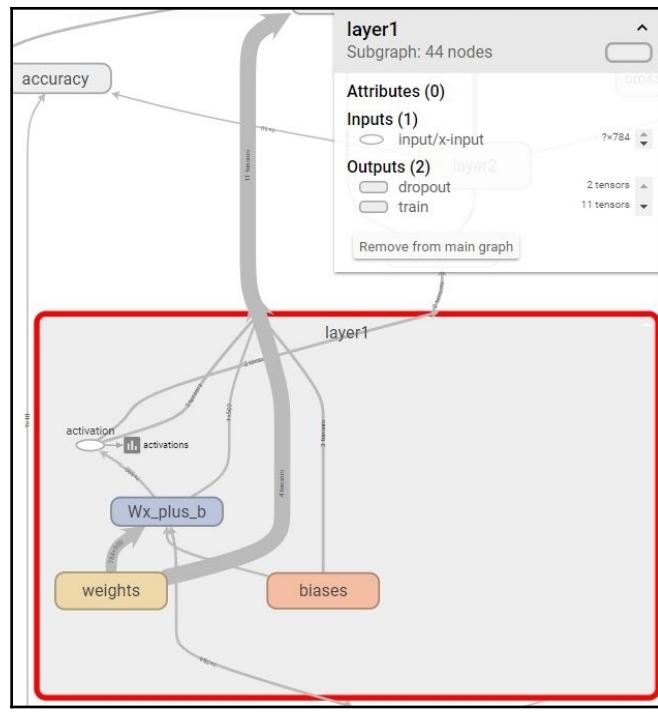


Reshaped image

When building a model, TensorBoard visual representations of the reshaped images can prove helpful for validation purposes.

Layer 1 – managing the inputs to the network

As with the Keras, implementing a layer in TensorFlow is done in a few lines but with TensorBoard. Layer 1 takes `x-input`, calculates a weight, applies an activation function and sends its output directly to the training node and also to the dropout node. The following graph represents layer 1 and the result of the operations to **OUTPUTS** (dropout and train):



Layer 1 and result of operations

Weights, biases, and preactivation

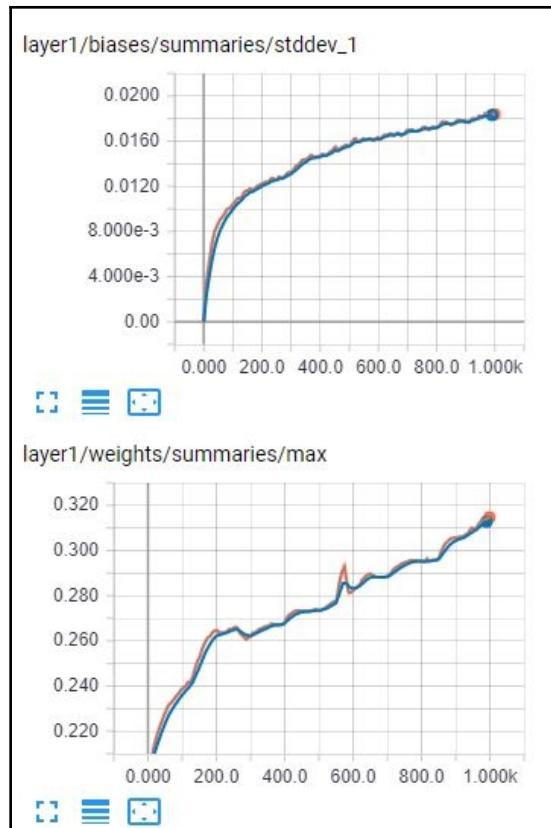
The core lines have the visual representation TensorBoard `tf.name_scope` functions as described before as implemented in the following code.

```
with tf.name_scope(layer_name):
    # This Variable will hold the state of the weights for the layer
    with tf.name_scope('weights'):
        weights = weight_variable([input_dim, output_dim])
        variable_summaries(weights)
    with tf.name_scope('biases'):
        biases = bias_variable([output_dim])
        variable_summaries(biases)
    with tf.name_scope('Wx_plus_b'):
        preactivate = tf.matmul(input_tensor, weights) + biases
        tf.summary.histogram('pre_activations', preactivate)
        activations = act(preactivate, name='activation')
        tf.summary.histogram('activations', activations)
    return activations
```



Notice the `tf.summary` module in the code. Not only does TensorFlow provide a data flow graph structure with the `tf.name_scope` that shows the operation, but also the values of the variables are stored with `tf.summary`. It stores the structure of the operations as well.

The values themselves can be viewed in following charts:



Data flow graph

Each chart has its statistics described in a `summary` function that defines the information needed to fine-tune and debug a model.

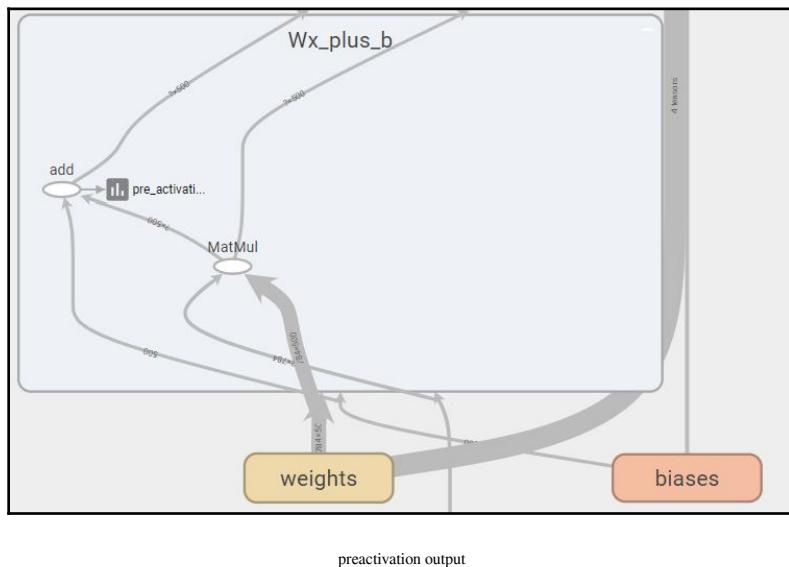
`def variables_summaries(var)` contains a list of very useful pieces of information. The following Python source code contains comments that provide information on the variables defined:

```
#Defining many summaries for TensorBoard Visualization of:  
# a)weights that will send weights to this function  
:variable_summaries(weights)  
# b)biases that will send biases to this function:  
variable_summaries(biases)  
# TensorBoard will then display charts of summaries described as follows on  
the related  
# TensorBoard dashboards  
# These weight and bias summaries provide a clear view, in TensorBoard of  
the direction  
# the values are taking during training. It will help fine-tune the  
hyperparameters when  
# things go wrong or training does not go fast enough.  
def variable_summaries(var):  
with tf.name_scope('summaries'):  
mean = tf.reduce_mean(var) #the mean of the elements of a tensor  
tf.summary.scalar('mean', mean)  
with tf.name_scope('stddev'):  
stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))#var-mean will be  
displayed  
tf.summary.scalar('stddev', stddev) # calculates the standard deviation of  
the input  
tf.summary.scalar('max', tf.reduce_max(var)) #calculates the maximum of the  
elements in the tensor  
tf.summary.scalar('min', tf.reduce_min(var)) #calculates the minimum of the  
elements in the tensor  
tf.summary.histogram('histogram', var) # summarizes var for the histogram  
tab on TensorBoard
```

With this number of quality controls, exploring TensorBoard will provide deep insights into the neural network.

Displaying the details of the activation function through the preactivation process

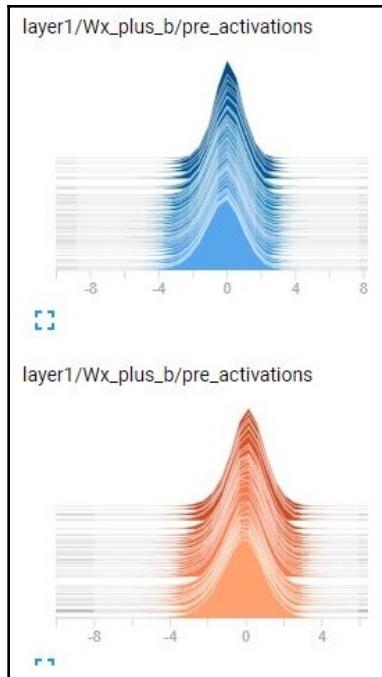
We often focus on the activation function in a layer. However, when a system fails and we have to debug it, having a peek at the preactivation function sometimes provides effective clues on what is going wrong. To visualize the preactivation output shown in the following graph, click on **Wx_plus_b** in layer 1, which shows that many events are influencing the training process:



Several notable operations occur with **wx_plus_b**:

- The weights and biases are connected to the training process, which is continuously modifying them; this happens even before the **MatMul** operation. Several tensors are sent to the training node and **Matmul**.
- The **Matmul** operation carries out the multiplications (Wx). But its output sends information to the training node and produces an output for the **add** node.
- The **add** node itself sends its output to the activation function and also to the preactivation data to the histogram summary so that we can analyze the behavior of this hidden layer.

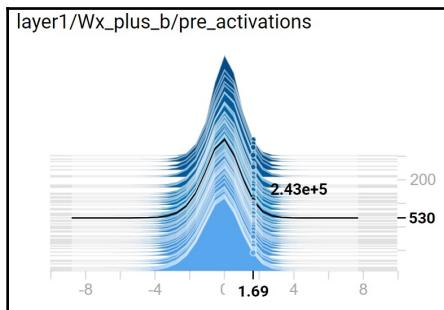
Now when we click on the **HISTOGRAMS** section on TensorBoard, we can visualize the evolution of weight and bias training in the following diagrams:



Evolution of weight and bias training

In this model, the preactivation diagrams show a smooth Gaussian-like bell (peak in the middle and similar slopes on each side). When building a model, everything seems fine. But if the curves become erratic and training the model fails, these histograms will provide critical information.

You can zoom in and check the values if necessary as shown in the following screenshot:



Evolution (Zoom in)

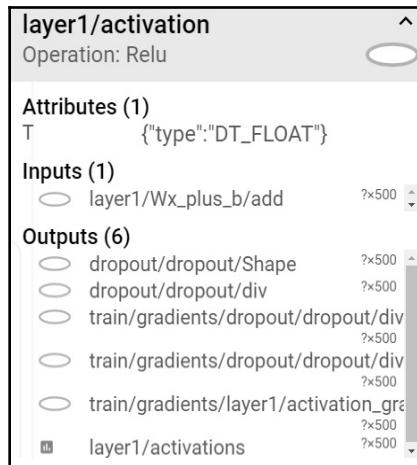
TensorBoard makes hidden layer intermediate values visible, which enhances our mental representation of a model.

In a real-life project, if the activation node goes wrong, checking the preactivation status of the layer eliminates a possible cause of design errors.

The activation function of Layer 1

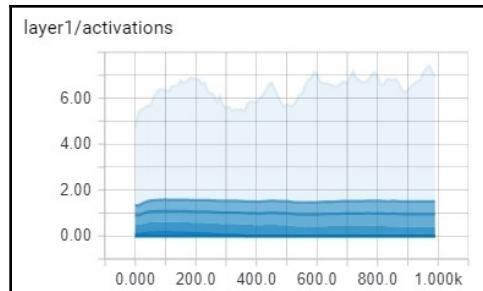
The activation function of this model is ReLU function (see Chapter 9, *Getting Your Neurons to Work*).

When we peek into its structure, once again we see that it is not just an activation function in a sequential process. Its output goes to the dropout and training nodes with complex intermediate calculations as shown in the following description of the activation function:



Activation function description

TensorBoard provides us with yet another visual representation to control the distribution of preactivation and, in this case, activations. The **DISTRIBUTIONS** section enables us to detect anomalies in the activation phase once we have verified that the preactivation functions produce acceptable results.



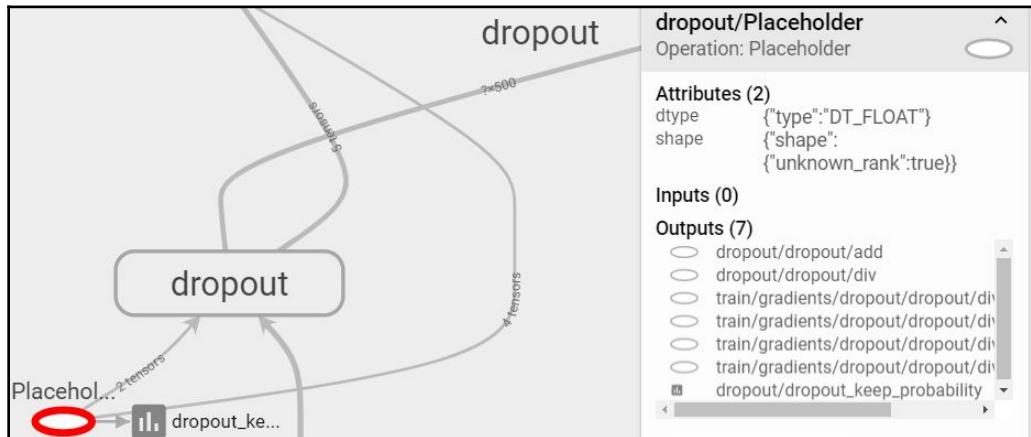
Visual representation complex node and edge structure

The visual representation of the complex node and edge structure of a TensorFlow data flow graph come in handy when design errors slow a project down.

Dropout and Layer 2

Dropouts and the designing of a layer were explained previously.

However, let us focus on some features of the dropout node: dropout keep probability and the outputs in the following diagram.

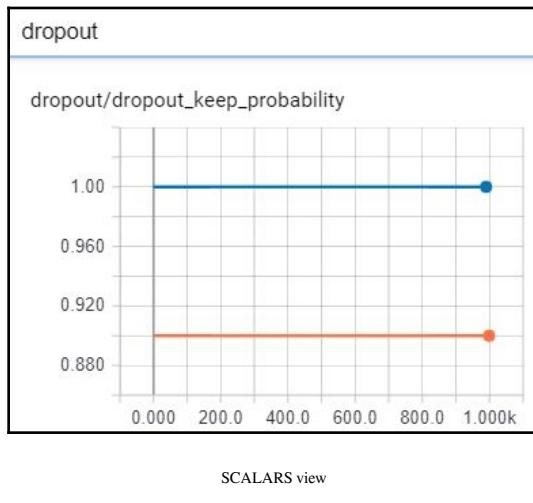


Dropout node

A dropout keep probability plays an overfitting reduction role. The probability leads to dropping neurons during the training process. The scaling is done automatically. Furthermore, this dropout feature is turned on and off during testing:

```
with tf.name_scope('dropout'):
    keep_prob = tf.placeholder(tf.float32)
    tf.summary.scalar('dropout_keep_probability', keep_prob)
    dropped = tf.nn.dropout(hidden1, keep_prob)
```

TensorBoard receives the information through the `summary.scalar` and can be viewed in the following **SCALARS** section:

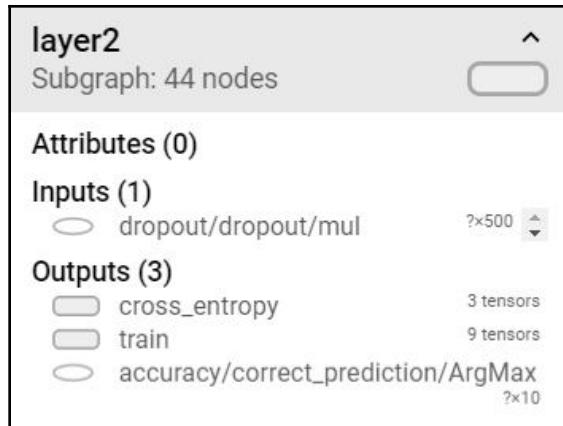


Layer 2

The graph structure of layer 2 is the same as that of layer 1.

However, there are notable output differences in the connections (edges) to other nodes:

- Layer 1 sends an output to the drop node and the layer 2 node
- Layer 2 now sends its output to the training node, the accuracy node, and the cross-entropy node as shown in TensorBoard



Layer 2 description

Measuring the precision of prediction of a network through accuracy values

The accuracy node contains two parts: correct prediction and accuracy. The output value is not sent to the training node but the correct predictions (see next section, *Correct prediction*) part of it is used by cross-entropy. Accuracy measures the performance of the system. You can stop training by limiting the number of epochs hyperparameter if you estimate that the accuracy is sufficient to achieve your goals. Or you can use accuracy to fine-tune the training model to increase its rate faster by modifying the hyperparameters or the design of your model.

Correct prediction

The following code implements `correct_prediction`, which uses an `argmax` function.

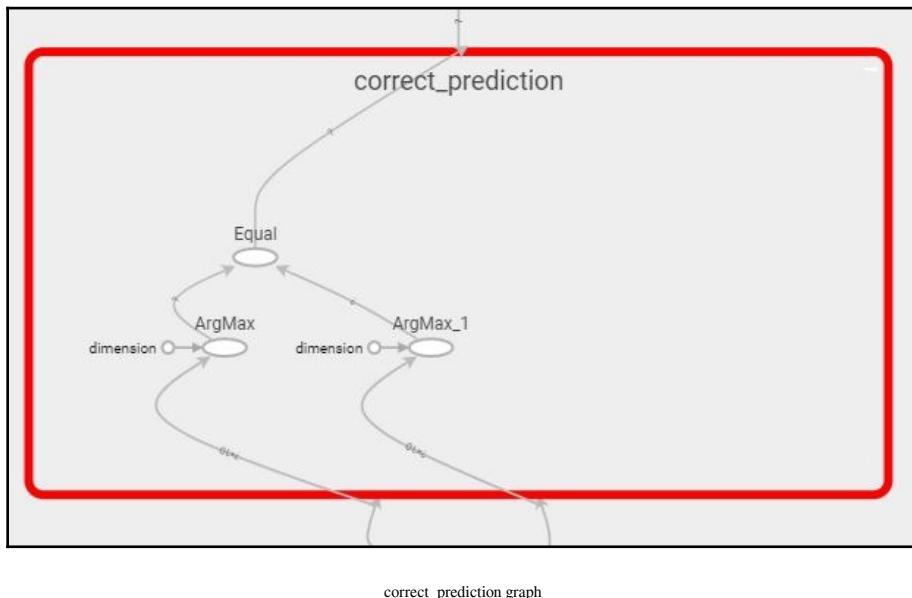
```
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
```

`tf.argmax` will output the index of the largest value of the input tensors. This simplifies the task considerably by avoiding the need to manage scores of values. When things go wrong—and they often do in real-life projects—we can go back to the preceding nodes and debug the data flow or improve the design of the data flow graph structure.



When accuracy fails to meet expectations, TensorBoard provides us with a two-dimensional debugging approach: fine-tune the data flow graph structure itself and also explore the data flow values that the node operations produce and send to other nodes through the edges.

The following graph of the `correct_prediction` node has two inputs. The first one is the `y-input`, which comes straight from the input node. The second input is the output of layer 2, which has processed the `x-input` input that came from its preceding nodes and is communicating with the core training and cross-entropy nodes as well.



correct_prediction graph

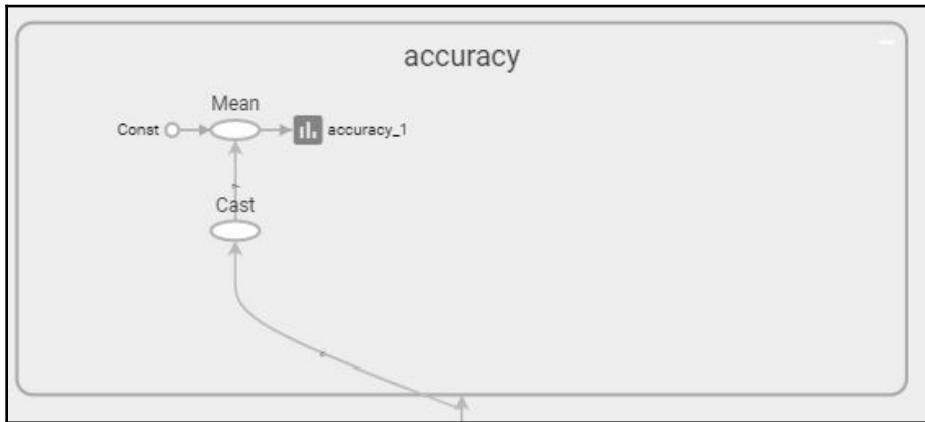
`argmax` is applied separately to both inputs and sent to the accuracy node, and correct predictions are used by the cross-entropy node.

accuracy

The `accuracy` uses the output of the correct prediction to compute the mean of the elements of the tensor in the following code:

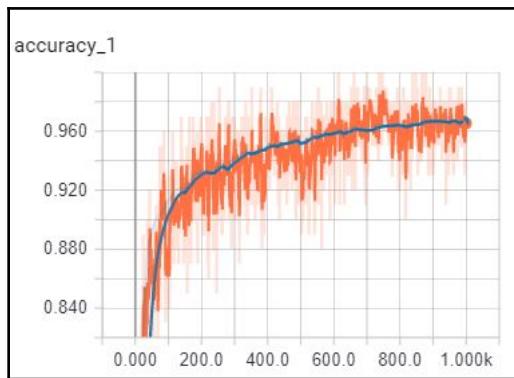
```
with tf.name_scope('accuracy'):  
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))  
    tf.summary.scalar('accuracy', accuracy)
```

`tf.reduce_mean` calculates the mean in a standard way. For example, the mean of 2 and 4 is 3. Reducing the values of the input gives a good picture of how the model is doing during training without overloading the designer with too many values to analyze, as shown in the following diagram:



Accuracy scalar

The accuracy scalar is sent to `accuracy`, which can be viewed in the following screenshot of the **SCALARS** section.



SCALARS section

You will notice that the accuracy values go up and up until we almost reach a **1** top score value.

Take another good look. This rarely happens in real-life projects.



Accuracy fails to increase in many real-life projects without hard work and analysis.

Bad accuracy will tell you to go back and do the following:

- Check the data flow—initial datasets, initial hyperparameters, and any value that you can check and change in the data flow
- Check the data flow graph structure to find flaws and improve them

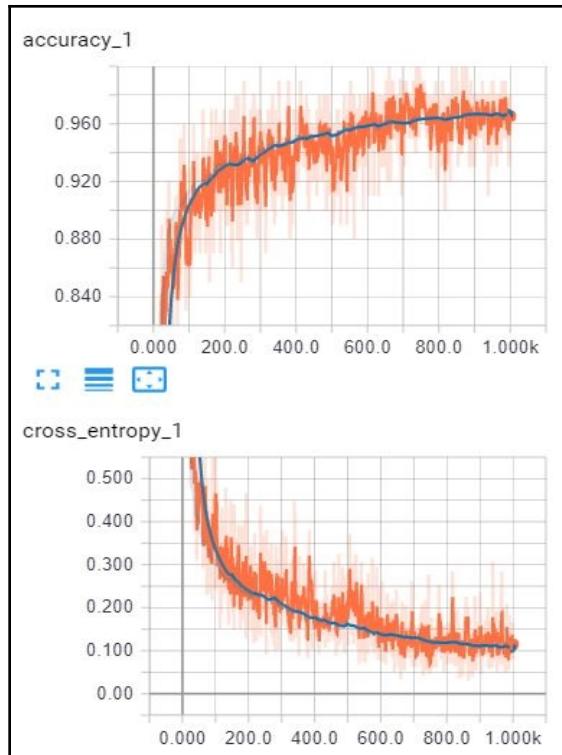
Then start the training process again.

Cross-entropy

Cross-entropy was described in [Chapter 9, Getting Your Neurons to Work](#). The first input comes from layer 2, the result of the preceding nodes computing `x-input`. The second input, `y-input`, comes directly from the input node, as implemented in the following code:

```
with tf.name_scope('total'):  
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(  
        logits=layer2, labels=y))  
    tf.summary.scalar('cross_entropy', cross_entropy)
```

TensorFlow measures its progression as shown in the `cross_entropy_1` and `accuracy_1` charts that follow:



cross_entropy_1 and accuracy_1 charts

Note how nicely the value of cross-entropy goes down a nice gradient (slope) as the loss function as the training progresses. This curve shows that the errors are being reduced. Then note how nicely the accuracy curve goes up as the errors diminish.

Think about it. In real-life projects, the cross-entropy completely slows down at one point while accuracy becomes erratic. The effect of this will be limited because cross-entropy takes the correct predictions into account.

However, more often than not, hard work needs to be put in when facing our implementations.

Training

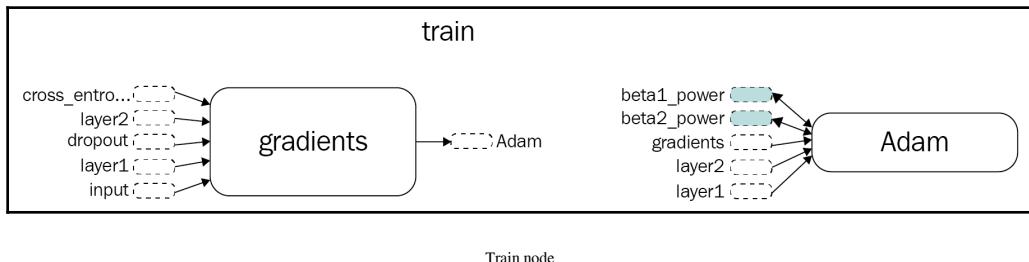
As in Chapter 9, *Getting Your Neurons to Work*, an algorithm with adaptive learning rates trains the model. The Adam algorithm using a cross-entropy loss function just like Chapter 9, *Getting Your Neurons to Work*, as shown here:

```
with tf.name_scope('train'):  
    train_step = tf.train.AdamOptimizer(FLAGS.learning_rate).minimize(  
        cross_entropy)
```



The similarity between the Keras model in Chapter 9, *Getting Your Neurons to Work*, and this TensorFlow model does not come as a surprise since Keras uses TensorFlow as a backend.

The train node has several inputs from other nodes for its Adam optimizer—cross-entropy, layer2, dropout, layer1, and x-input, as shown in the following graph:



At this point, TensorFlow provides all the necessary information to build and debug the model.



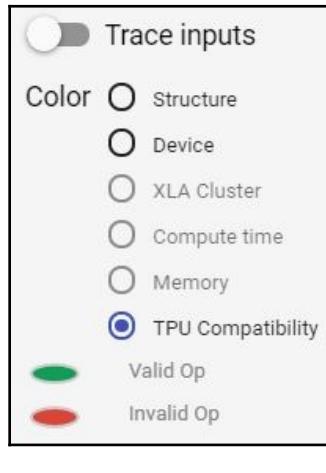
Drill down into all the nodes in this model on the **GRAPHS** dashboard. Explore all of the information about all the other sections: **SCALARS**, **IMAGES**, **DISTRIBUTIONS**, and **HISTOGRAMS** provided by this unit testing program. The more you know when things go wrong, the faster you will correct them.

Optimizing speed with Google's Tensor Processing Unit

Exploring a unit-testing model represents an effort on our part. Now, imagine having to build a model that is 10 times this size. Training the model to find out where and why things went wrong may prove time consuming.

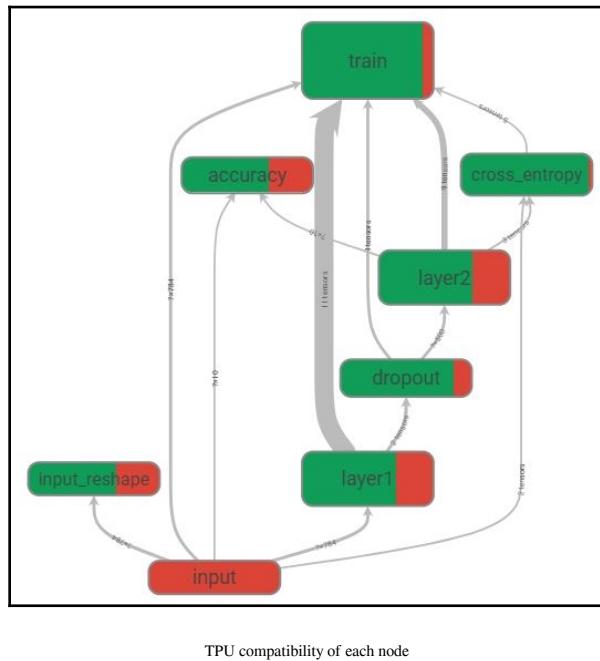
Google has come up with an innovation: a **Tensor Processing Unit (TPU)**. Google Search, Google Translate, and Google Photos use a TPU to speed up computations.

Making that available to us on its data centers will make a difference. TensorBoard already has a key function you might want to explore while building a complex and time-consuming model. This function is the TPU Compatibility option on TensorBoard (see the following screenshot).



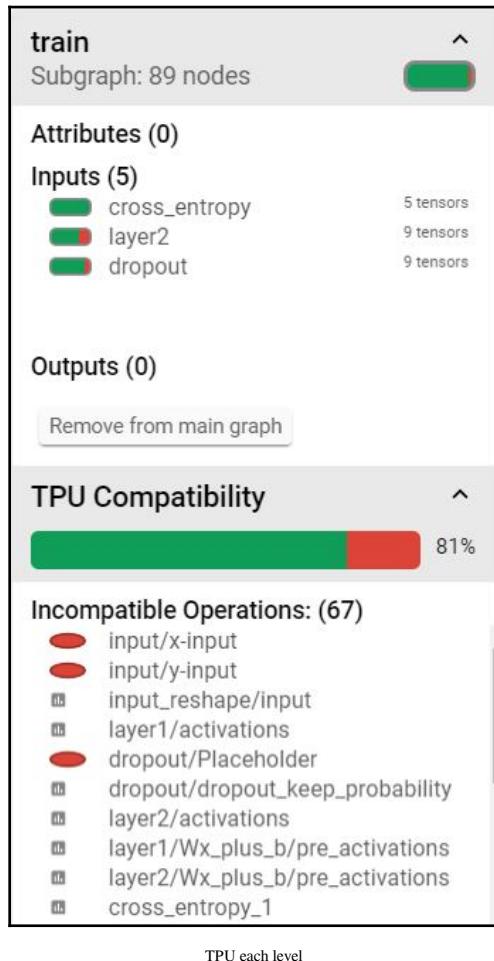
Click on **TPU Compatibility** (this chapter used the **Structure** option until now).

The following graph will now display the levels of TPU compatibility of each node (green + red proportions):



This means that, in future when you are designing a complex model, you should keep an eye on TPU compatibility if you are going to consume a lot of CPU/GPU.

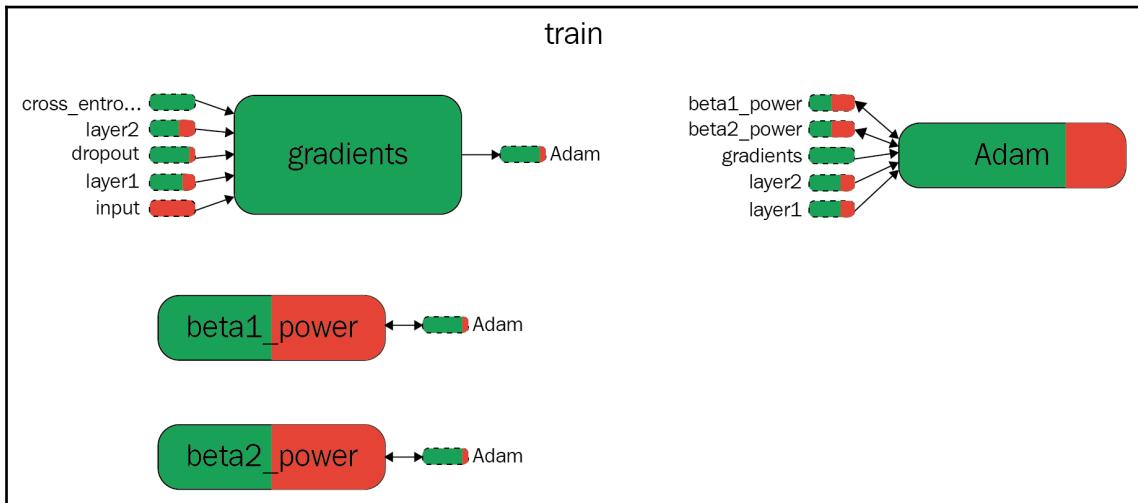
TensorBoard also provides the following list, for example, to display each level you are drilling down to:



TPU each level

Even if you do not have access to a TPU, the read nodes will signal where you have to be careful. You may want to optimize the model before even starting to train it.

One of the main nodes to keep an eye on is the training node that contains the Adam optimizer, as shown in the following screenshot:



As models grow larger and larger, optimization will become a key field. Teams will have to optimize data graph structures, data flows, and machine configuration, including network speeds (and balancing).

Summary

The TensorFlow model represented in TensorBoard contains several basic building blocks of ANNs, whether they are CNNs, RNNs such as LTSM networks, or other forms. All of these models rely on numerical computation in one form or the other: weights, biases, matmul operations, activation functions (logistic sigmoid, ReLU, or others), layers, optimizers, loss functions, and more. All of these tools belong to the field of applied mathematics.

This approach has been around for decades. However, as time goes by, linguists, philosophers, historians, and many more people fascinated by AI will bring new ideas, new programming languages, interfaces, and APIs. Focus on the core concepts, not the tools, and you will always be able to keep up to date and innovate.

Biomimicking humans will inevitably be extended to conceptual representation learning, the fascinating world of our video-clipped minds.

Complex projects require conceptual representation.

Chapter 11, Conceptual Representation Learning, will lead you into the uncharted territory of conceptual representation learning.

11

Conceptual Representation Learning

Artificial intelligence has just begun its long tryst with human intellect. Chapter 9, *Getting Your Neurons to Work*, showed the possibilities of a CNN built with Keras. Chapter 10, *Applying Biomimicking to Artificial Intelligence*, reproduced what humans use the most to represent a problem: visualization.

Understanding cutting-edge machine learning, and deep learning theory only marks the beginning of your adventure. The knowledge you have acquired should help you become an AI visionary. Take everything you see as opportunities and see how it can fit in your projects. Reach the limits and skydive beyond them.

This chapter focuses on decision-making and visual representations and explains the motivation leading to **conceptual representation learning** (CRL) and **meta models** (MM), which form **CRLMMs**. I developed this CRLMM method successfully for **automatic planning and scheduling** (APS) software for corporate projects. I also used this CRLMM method for cognitive NLP chatbots (see Chapter 13, *Cognitive NLP Chatbots*, and Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*).

To plan, humans need to visualize necessary information (events, locations, and so on) and more critical *dimensions such as concepts*. A human being thinks in *mental images*. When we think, mental images flow through minds with numbers, sounds, odors, and sensations, transforming our environment into fantastic multidimensional representations similar to video clips.

The following topics will be covered in this chapter:

- An approach to CRLMM in three steps: transfer learning, domain learning, and the motivation for using CRLMM. Over the years, I've successfully implemented CRL in C++, Java, and logic programming (Prolog) in various forms on corporate sites. In this chapter, I used Python to illustrate the approach with the Keras **Convolutional Neural Network (CNN)** built in Chapter 9, *Getting Your Neurons to Work*.
- Transfer learning using the Keras model trained in Chapter 9, *Getting Your Neurons to Work*, to generalize image recognition.
- Domain learning to extend image recognition trained in one field to another field.
- Possible use of GANs (Generative Adversarial Networks) to produce CRL datasets.
- The use (or not) of Autoencoders.

Technical requirements

Python 3.6x 64-bit from <https://www.python.org/>:

Packages and modules:

```
from keras.models import model_from_json
from keras.models import load_model
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.preprocessing.image import ImageDataGenerator
from keras.models import save_model
from keras import backend as K
from pprint import pprint

import Numpy as np
import matplotlib.image as mpimg
import scipy.ndimage.filters as filter
import matplotlib.pyplot as plt
```

Programs in GitHub Chapter11:

- EAD_CNN_MODEL.py
- CNN_STRATEGY_MODEL.py
- CNN_TDC_TRAINING.py
- CNN_TDC_STRATEGY.py

Check out the following video to see the code in action:

<https://goo.gl/e7p9Lg>

Generate profit with transfer learning

When it comes to reasoning and thinking in general, we use mental images with some words. Our thoughts contain concepts, on which we build solutions. The trained model of Chapter 9, *Getting Your Neurons to Work*, can now classify images of a certain type. In this section, the trained model will be loaded and then generalized through transfer learning to classify similar images.

The motivation of transfer learning

Transfer learning provides a cost-effective way of using trained models for other purposes within the same company, such as the food processing company described in chapter 9, *Getting Your Neurons to Work*. This chapter describes how this food processing company used the model for other similar purposes.

The company that succeeds in doing this will progressively generalize the use of the solution. By doing so, inductive abstraction will take place and lead to other AI projects, which proves gratifying to the management of a corporation and the teams providing the solutions.

Inductive thinking

Induction uses inferences to reach a conclusion. For example, a food processing conveyor belt with missing products will lead to packaging productivity problems. If insufficient amounts of products reach the packaging section, this will slow down the whole production process.

By observing similar problems in other areas of the company, inferences from managers will come up, such as, *If insufficient amounts of products flow through the process, production will slow down.*

Inductive abstraction

The project team in charge of improving efficiency in the company needs to find an *abstract representation* of the problem to implement a solution through organization or software. This book deals with the AI side of solving the problem. Organizational processes need to define how AI will fit in with several on-site meetings.

The problem AI needs to solve

In this particular case, each section of the factory has an optimal production rate defined per hour or per day, for example. The equation of an **optimal production rate (OPR)** per hour can be summed up as follows:

$$OPR : \min(p(s)) \leq opr \leq \max(p(s))$$

Where:

- p is the production rate of a given section (different production departments of a factory) s .
- $p(s)$ is the production rate of the section.
- $\min(p(s))$ is the historical minimum (trial and error over months of analysis). Under that level, the whole production process will slow down.
- $\max(p(s))$ is the historical maximum. Over that level, the whole production process will slow down as well.
- OPR is the optimal production rate.

The first time somebody sees this equation, it seems difficult to understand. The difficulty arises because one has to visualize the process, which is the goal of this chapter. Every warehouse, industry, and service uses production rates as a constraint to reach profitable levels.

Visualization requires representation at two levels:

- Ensuring that if a packaging department is not receiving enough products, it will have to slow down or even stop sometimes.
- Ensuring that if a packaging department receives too many products, it will not be able to package them. If the input is a conveyor belt with no intermediate storage (present-day trends), then it will have to be slowed down, slowing down or stopping the processes before that point.

In both cases, slowing down production leads to bad financial results and critical sales problems through late deliveries.

In both cases, an OPR gap is a problem. To solve it, another level of abstraction is required. First, let's separate the OPR equation into two parts:

$$\begin{aligned} OPR &>= \min(p(s)) \\ OPR &<= \max(p(s)) \end{aligned}$$

Now let's find a higher control level through variance variable v :

$$\begin{aligned} v_{min} &= |OPR - \min(p(s))| \\ v_{max} &= |OPR - \max(p(s))| \end{aligned}$$

v_{min} and v_{max} are the absolute values of the variance in both situations (not enough products to produce and too many to produce respectively).

The final representation is through a single control, detection, and learning rate (Greek gamma letter):

$$\Gamma = \max(v_{min}, v_{max})$$

This means that the variance between the optimal production rate of a given section of a company and its minimum speed (products per hour) will slow the following section down. If too few cakes (v_{min}), for example, are produced, then the cake packaging department will be waiting and stopping. If too many cakes are produced (v_{max}), then this section will have to slow down or stop. Both variances will create problems in a company that cannot manage intermediate storage easily, which is the case with the food processing industry.

With this single Γ parameter, Chapter 9, *Getting Your Neurons to Work*, Keras CNN, can start teaching a fundamental production concept: what a physical gap is.

The Γ gap concept

Teaching the CNN the gap concept will help it extend its thinking power to many fields:

- A gap in production, as explained before
- A gap in a traffic lane for a self-driving vehicle to move into
- Any incomplete, deficient area
- Any opening or window

Let's teach a CNN the Γ gap concept, **or simply Γ** . To achieve that goal, the CNN Keras model that was trained and saved in [Chapter 9, Getting Your Neurons to Work](#), now needs to be loaded and used. To grasp the implications of the Γ concept, imagine the cost of not producing enough customer orders or having piles of unfinished products everywhere. The financial transposition of the physical gap is a profit **variance** on set goals. We all know the pain those variances lead to.

Loading the Keras model after training

The technical goal is to load and use the trained CNN Keras model and then use the same model for other similar areas. The practical goal is to teach the CNN how to use the **Γ concept** to enhance the thinking abilities of the scheduling, chatbot, and other applications.

Loading the model has two main functions:

- Loading the model to compile and classify new images without training the model
- Displaying the parameters used layer by layer and displaying the weights reached during the learning and training phase

Loading the model to optimize training

A limited number of headers suffice to read a saved model with `READ_CNN_MODEL.py`, as implemented in the following lines:

```
from keras.models import model_from_json
from keras.models import load_model
import json
from pprint import pprint
```

```
#Direction containing the model file
directory='dataset/'
print("directory",directory)
```

The .json model saved is now loaded from its file as shown here:

```
#_____LOAD MODEL_____
json_file = open(directory+'model/model.json', 'r')
loaded_jsonf = json_file.read()
loaded_json=json.loads(loaded_jsonf)
json_file.close()
print("MODEL:")
pprint(loaded_json)
```

I used the pprint function in the following code instead of print to obtain a *pretty print* formatted display of the model:

```
print("MODEL:")
pprint(loaded_json)
```

Reading the model is not a formality. The output of the program will help you train the model. Each layer starts with a marker. In this case, class_name: indicates one of the layers added. For example, Conv2D in the following output provides vital information on the parameters and additional options of that layer:

```
MODEL:
{'backend': 'tensorflow',
'class_name': 'Sequential',
'config': [{'class_name': 'Conv2D',
'config': {'activation': 'relu',
'activity_regularizer': None,
'batch_input_shape': [None, 64, 64, 3],
'bias_constraint': None,
'bias_initializer': {'class_name': 'Zeros',
'config': {}},
'bias_regularizer': None,
'data_format': 'channels_last',
'dilation_rate': [1, 1],
'dtype': 'float32',
'filters': 32,
'kernel_constraint': None,
'kernel_initializer': {'class_name': 'VarianceScaling',
'config': {'distribution': 'uniform',
'mode': 'fan_avg',
'scale': 1.0,
'seed': None}},
'kernel_regularizer': None,
'kernel_size': [3, 3],
```

```
'name': 'conv2d_1',
'padding': 'valid',
'strides': [1, 1],
'trainable': True,
'use_bias': True}),
```

Each parameter contains very useful information. For example, 'padding': 'valid' means that padding has not been applied (no padding). In this model, the number and size of the kernels provide satisfactory results without padding and the shape decreases to the final status layer (classification) as shown here:

```
initial shape (570, 597, 4)
lay: 1 filters shape (568, 595, 3)
lay: 2 Pooling shape (113, 119, 3)
lay: 3 filters shape (111, 117, 3)
lay: 4 pooling shape (22, 23, 3)
lay: 5 flatten shape (1518,)
lay: 6 dense shape (128,)
lay: 7 dense shape (1,)
```

However, suppose you want to control the output shape of a layer so that the spatial dimensions do not decrease faster than necessary. One reason could be that the next layer will explore the edge of the image and that we need to explore them with kernels that fit in the shape.

In that case, a padding of size 1 can be added with 0 values as shown in the following matrix:

0	0	0	0	0	0
0	1	3	24	4	0
0	3	7	8	5	0
0	6	4	5	4	0
0	5	4	3	1	0
0	0	0	0	0	0

A padding of size 2 would add two rows and columns around the initial shape.

With that in mind, fine-tuning your training model by adding as many options as necessary will improve the quality of the results. The weights can be viewed by extracting them from the saved model file layer by layer, as shown in the following code snippet:

```
#_____ LOAD WEIGHTS_____  
  
loaded_model=model_from_json.loaded_jsonf)  
loaded_model.load_weights(directory+"model/model.h5")  
print("WEIGHTS")  
for layer in loaded_model.layers:  
    weights = layer.get_weights() # list of numpy arrays  
    print(weights)
```

Analyzing the weights used by the program will provide useful information about the way the optimizing process was carried out by the program. Sometimes, a program will get stuck, and the weights might seem off track. After all, a CNN can contain imperfections like any other program.

A look at the following output, for example, can help understand where the system went wrong:

```
WEIGHTS  
[array([[-0.1005416 , -0.08582606, -0.058176 , 0.11453936, 0.04511052,  
0.05389464, 0.03316241, 0.10681037, 0.08743957, 0.0007824 ,  
0.09682989, 0.01926496, -0.00474238, -0.05053157, -0.07657856,  
-0.10017087, -0.08425587, 0.06652132, 0.04018607, 0.13323469,  
0.12131914, -0.11700463, -0.09595016, -0.08730403, -0.06408932,  
-0.0613227 , 0.13427697, -0.08226932, 0.09776273, 0.00540088,  
-0.02313543, -0.03464791],
```



Much has been written about how to build CNNs and little about how to maintain them. This program provides one of the many tools to repair problems.

Loading the model to use it

Loading the model with `CNN_CONCEPT_STRATEGY.py` to use it requires more headers. I played it safe and loaded whatever would be needed to save the time otherwise spent in crashing and looking for imports as shown here:

```
from keras.preprocessing.image import load_img  
from keras.preprocessing.image import img_to_array  
import matplotlib.pyplot as plt  
import keras
```

```
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
from keras.models import model_from_json
from keras.models import load_model
import numpy as np
import cv2
from PIL import Image
```

Loading the .json file and weights is done by using the same code as in the READ_CNN_MODEL.py described previously. Once you load it, compile the model with the `model.compile` function, as follows:

```
# __compile loaded model
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

The model used for this case study and the image identification function has been implemented in two parts. First, we're loading and resizing the image with the following function, for example:

```
def identify(target_image):
    filename = target_image
    original = load_img(filename, target_size=(64, 64))
    #print('PIL image size',original.size)
    plt.imshow(original)
    plt.show()
    numpy_image = img_to_array(original)
    arrayresized = cv2.resize(numpy_image, (64,64))
    #print('Resized',arrayresized)
    inputarray = arrayresized[np.newaxis,...] # extra dimension to fit model
```

Keras expects another dimension in the input array to predict, so one is added to fit the model. In this case study, one image at a time needs to be identified. These little details are time-savers in the development phase.

I added the following two prediction methods and returned one:

```
#____PREDICTION_____
prediction1 = loaded_model.predict_proba(inputarray)
prediction2 = loaded_model.predict(inputarray)
print("image",target_image,"predict_proba:",prediction1,"predict:",predicti
on2)
return prediction1
```

Two prediction methods are there because, basically, every component needs to be checked in a CNN during a project's implementation phase to choose the best and fastest ones. To test prediction2, just change the return instruction.

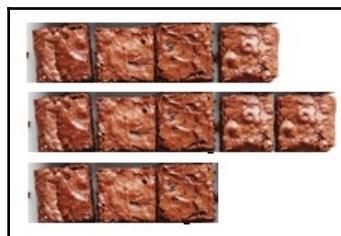


Once a CNN is running, it can prove difficult to find out what went wrong. Checking the output of each layer and component while building the network saves fine-tuning time once the full-blown model produces thousands of results.

The following example detects product **Γ gaps** on a conveyor belt in a food processing factory. The program loads the first image stored in the `classify` directory to predict its value. The program describes the prediction:

```
MS1='productive'  
MS2='gap'  
  
s=identify(directory+'classify/img1.jpg')  
if (int(s)==0):  
    print('Classified in class A')  
    print(MS1)
```

The program displays (optional) the shaped image, such as the following frame, which shows that the conveyor belt has a sufficient number of products at that point:

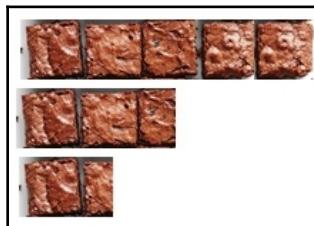


Output (shaped image)

The program then makes and displays its prediction 0, meaning no real gap has been found on the conveyor belt on this production line:

```
directory dataset/  
Strategy model loaded from training repository.  
image dataset/classify/img1.jpg predict_proba: [[ 0.]] predict: [[ 0.]]  
Classified in class A  
productive  
Seeking...
```

Seeking... means it is going to analyze the second image in the classify direction. It loads, displays, and predicts its value as shown in the following frame:



Output (shaped image)

The prediction (*value = 1*) correctly detected gaps on the conveyor belt, as shown in the following output:

```
image dataset/classify/img2.jpg predict_proba: [[ 1.]] predict: [[ 1.]]
Classified in class B
gap
```

Now that the predictions of the CNN have been verified, the implementation strategy needs approval. A convolutional neural network contains marvels of applied mathematics. CNNs epitomize deep learning in themselves. A researcher could easily spend hundreds of hours studying them.

However, applied mathematics in the business world requires profitability. As such, the components of CNNs appear as ever-evolving concepts. Added kernels, activation functions, pooling, flattening, dense layers, and compiling and training methods act as a starting point for architectures, not as a finality.

Using transfer learning to be profitable or see a project stopped

At one point, a company will demand results or shelve the project. If a spreadsheet represents a faster and a sufficient solution, a deep learning project will face potential competition and rejection. Many engineers learning artificial intelligence will have to assume the role of standard SQL reporting experts before accessing real AI projects.

Transfer learning appears as a solution to the present cost of building and training a CNN program. Your model might just pay off that way. The idea is to get a basic AI model rolling profits in fast for your customer and management. Then you will have everybody's attention. To do that, you must define strategies.

Defining the strategy

If a deep learning CNN expert, for example, our friend Pert Ex, comes to a top manager saying that this CNN model can classify CIFAR-10 images of dogs, cats, cars, plants, and more, the answer will be, *So what? My 3-year-old child can too. In fact, so can my dog!*

The IT manager in that meeting might even blurt out something like, *We have all the decision tools we need right now, and our profits are increasing. Why would we invest in a CNN?*

The core problem of marketing AI to real-world companies is that it relies upon the belief in the necessity of a CNN in the first place. Spreadsheets, SQL queries, standard automation, and software do 99% of the job. Most of the time, it does not take a CNN to replace many jobs; just an automated spreadsheet, a query, or standard and straightforward software is enough. Jobs have been sliced into simple-enough parts for decades to replace humans with basic software.

Before presenting a CNN, Pert Ex has to find out how much the company can earn using it. Pert has found an area with a quick win to show the power of the system in a cost-effective way.

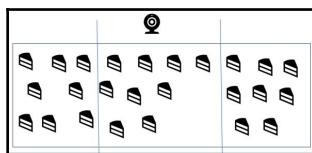


Understanding, designing, building, and running a CNN does not mean much regarding business. All the hard work we put in to understanding and running these complex programs will add up to nothing if we cannot prove that a solution will generate profit. Without profit, the implementation costs cannot be recovered, and nobody will listen to a presentation about even a fantastic program.

Pert Ex describes the example in the previous section applied to this food processing company.

Applying the model

At the food processing company, one of the packaging lines has a performance problem. Sometimes, randomly, some of the cakes are missing on the conveyor belt, as shown in the following frame:



Food processing company example

Pert explains that to start a cost-effective project, a cheap webcam could be installed over the conveyor belt. It'll take a random sample picture every 10 seconds and process it to find the holes as shown in the interval between the two blue lines. If a hole is detected, it means some cakes have not made it to the conveyor belt (production errors).



Pert estimates that a 2% to 5% productivity rate increase could be obtained by automatically sending a signal to the production robot when some cakes are missing.

The CEO decides to try this system out and approves a prototype test budget.

Making the model profitable by using it for another problem

The food processing experiment on the conveyor belt turns out to work well enough with dataset type d_1 and the CNN model M to encourage generalization to another dataset, d_2 , in the same company.

Transfer learning consists of going from $M(d_1)$ to $M(d_2)$ using the same CNN model M , with some limited, cost-effective additional training. Variations will appear, but they will be solved by shifting a few parameters and working on the input data following some basic dataset preparation rules:

- **Overfitting:** When the model fits the training data quickly with 100% accuracy, this may or not be a problem. In the case of classifying holes on the conveyor, belt overfitting might not prove critical. The shapes are always the same and the environment remains stable. However, in an unstable situation with all sorts of different images or products, then overfitting will limit the effectiveness of a system.
- **Underfitting:** If the accuracy drops down to low levels such as 20%, then the CNN or any other model will not work. The datasets and parameters need optimizing. Maybe the number of samples needs to be increased for $M(d_2)$, or reduced, or split into different groups.
- **Regularization:** Regularization, in general, involves the process of finding how to fix the generalization problem of $M(d_2)$, not the training errors of $M(d_2)$. Maybe an activation function needs some improvements, or the way the weights have been implemented require attention.

There is no limit to the number of methods you can apply to find a solution, just like standard software program improvement.

Where transfer learning ends and domain learning begins

Transfer learning can be used for similar types of objects, or images in this case as explained. The more similar images you can train within a company with the same model, the more the **return on investment (ROI)** it will produce and the more this company will ask you for more AI innovations.

Domain learning takes a model such as the one described in Chapter 9, *Getting your Neurons to Work*, and that describe in this chapter to generalize it. The generalization process will lead us to domain learning.

Domain learning

This section on domain learning builds a bridge between classic transfer learning, as described previously, and another use of domain learning I found profitable in corporate projects: teaching a machine a concept (CRLMM). The chapter focuses on teaching a machine to learn to recognize a gap in situations other than at the food processing company.

How to use the programs

You can read the chapter first to grasp the concepts, or play with the programs first. In any case, `CNN_TDC_STRATEGY.py` loads trained models (you do not have to train them again for this chapter) and `CNN_CONCEPT_STRATEGY.py` trains the models.

The trained models used in this section

This section uses `CNN_TDC_STRATEGY.py` to apply the trained models to the target concept images. `READ_CNN_MODEL.py` (as shown previously) was converted into `CNN_TDC_STRATEGY.py` by adding variable directory paths (for the `model.h5` files and images) and classification messages, as shown in the following code:

```
#loads,traffic,food processing
A=['dataset_0/','dataset_traffic/','dataset/']
MS1=['loaded','jammed','productive']
MS2=['unloaded','change','gap']
```

```
# _____LOAD MODEL_____
json_file = open(directory+'model/model.json', 'r')
....
s=identify(directory+'classify/img1.jpg')
```

Each subdirectory contains four subdirectories:

- **classify**: Contains the images to classify
- **model**: The trained `model.h5` used to classify the images
- **test_set**: The test set of conceptual images
- **training_set**: The training set of conceptual images

Chapter 9, *Getting Your Neurons to Work*, describes the whole process.

The training model program

For this chapter, you do not need to train the models; they were already trained using `CNN_TDC_TRAINING.py`. This program is an extension of the training program in Chapter 9, *Getting Your Neurons to Work*. The directory paths have become variables to access the subdirectories described previously. The paths can be called, as shown in the following code:

```
A=['dataset_0/','dataset_traffic/','dataset/']
scenario=3 #reference to A
directory=A[scenario] #transfer learning parameter (choice of images)
print("directory",directory)
```

You do not need to use this program for this chapter. The models were trained and automatically stored in their respective subdirectories on the virtual machine delivered with the book.



For this chapter, focus on understanding the concepts. You can read the chapter without running the programs, open them without running them, or run them, whatever makes you comfortable. The main goal is to grasp the concepts to prepare for the subsequent chapters.

GAP – loaded or unloaded

When planning in general, in a factory for example, a key concept of a gap is unloaded, loaded.



Set the scenario to 0 in `CNN_TDC_STRATEGY.py` to run this conceptual model: `scenario=0 #reference to A,MS1,MS2.`

The gap concept has just become a polysemy word concept (polysemy means different meanings; see Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations Small to Large Companies*).

In the cake situation, the Γ gap was negative in its $g1$ subset of meaning and concepts applied to a CNN, relating it to negative images $n+g1$:

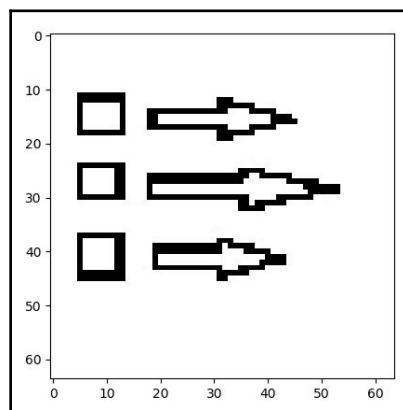
$$ng1=\{\text{missing, not enough, slowing production down...bad}\}$$

The full-of-products image was positive, $p+g2$:

$$pg2=\{\text{good production flow, no gap}\}$$

In this example, the CNN is learning how to distinguish an abstract representation, not an image like for the cakes. Another subset of Γ (gap conceptual dataset) is loaded/unloaded.

The following abstract image is loaded. The squares represent some kind of production machine and the arrows represent the load-in time. This means that the x axis represents time and the y axis represents machine production resources.



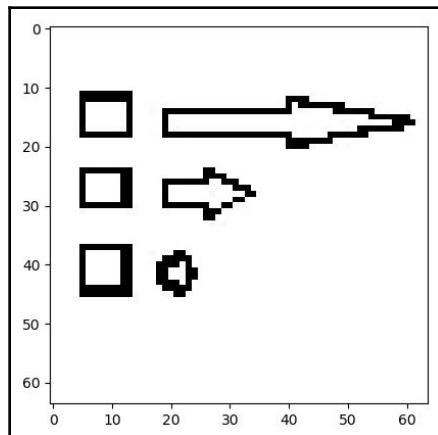
Abstract image

The CNN model runs and produces the following result:

```
directory dataset_0/  
Strategy model loaded from training repository.  
image dataset_0/classify/img1.jpg predict_proba: [[ 0.]] predict: [[ 0.]]  
Classified in class A  
loaded  
Seeking...
```

The CNN recognizes this as loaded. The task goes beyond classifying. The system needs to recognize this to make a decision.

Another image produces a different result. In this case, an unloaded gap appears in the following screenshot:



Abstract image

And the CNN has a different output, as shown here:

```
Seeking...  
image dataset_0/classify/img2.jpg predict_proba: [[ 1.]] predict: [[ 1.]]  
Classified in class  
unloaded
```

The gap concept Γ has added two other subsets, g_3 and g_4 , to its dataset. We now have:

$$\Gamma = \{g_1, g_2, g_3, g_4, \dots, g_n\}$$

The four $g1$ to $g4$ subsets of Γ are:

$$ng1=\{\text{missing, not enough, slowing production down...bad}\}$$

$$pg2=pg2=\{\text{good production flow, no gap}\}$$

$$g3=\{\text{loaded}\}$$

$$g4=\{\text{unloaded}\}$$

The remaining problem will take some time to solve. $g4$ (gap) can sometimes represent an opportunity for a machine that does not have a good workload to be open to more production. In some cases, $g4$ becomes $pg4(p = \text{positive})$. In other cases, it will become $ng4(n = \text{negative})$ if production rates go down.

As in Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies*, the context had become strategic when a bus (coach) broke down and the translation came out as a trainer (coach) and not as a vehicle. The problem in these situations relies on the fact that additional input information is required. More words will not help. CRLMM will bring additional concepts into the model.

GAP – jammed or open lanes

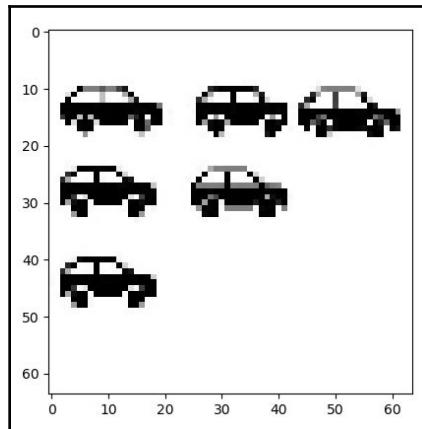
A self-driving car needs to recognize whether it is in a traffic jam or not. Also, the self-driving car has to know how to change lanes when it detects enough space (a gap) to do that.

This produces two new subsets:

$$g5=\{\text{traffic jam, heavy traffic...too much traffic}\}$$

$$g6=\{\text{open lane, light traffic...normal traffic}\}$$

The model now detects g5 (traffic jam), as shown in the following screenshot:

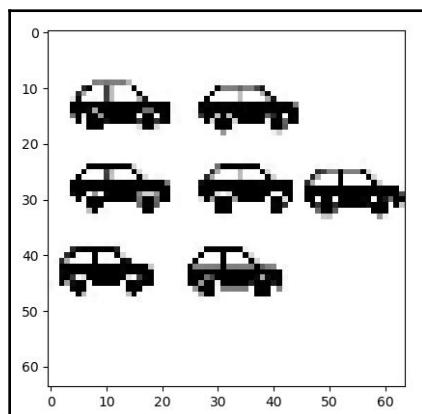


Traffic jam example

The following output appears correctly:

```
directory dataset_traffic/  
Strategy model loaded from training repository.  
image dataset_traffic/classify/img1.jpg predict_proba: [[ 0.]] predict: [[  
0.]]  
Classified in class A  
jammed
```

g6 comes out right as well, as shown in this screenshot:



Traffic jam example

A potential lane change has become possible, as detected by the following code:

```
Seeking...
image dataset_traffic/classify/img2.jpg predict_proba: [[ 1.]] predict: [[
1.]]
Classified in class B
change
```

The gap dataset

At this point, the Γ (gap conceptual dataset) has begun to learn several subsets:

$$\Gamma = \{g1, g2, g3, g4, g5, g6\}$$

In which:

$$ng1=\{\text{missing, not enough, slowing production down...bad}\}$$

$$pg2=pg2=\{\text{good production flow, no gap}\}$$

$$g2=\{\text{loaded}\}$$

$$g3=\{\text{unloaded}\}$$

$$pg4=\{\text{traffic jam, heavy traffic...too much traffic}\} ng5=\{\text{open lane, light traffic...normal traffic}\}$$

Notice that $g2$ and $g3$ do not have labels yet. The food processing context provided the label. Concept detection requires a context, which CRLMMs will provide.

Generalizing the Γ (gap conceptual dataset)

The generalization of Γ (gap conceptual dataset) will provide a conceptual tool for meta-models.



Γ (gap conceptual dataset) refers to negative, positive, or undetermined space between two elements (objects, locations, or products on a production line).

Γ (gamma) also refers to a gap in time: too long, not long enough, too short, or not short enough.

Γ represents the distance between two locations: too far or too close.

Γ represents a misunderstanding or an understanding between two parties: a divergence of opinions or a convergence.

All of these examples refer to gaps in space and time viewed as space.

Generative adversarial networks

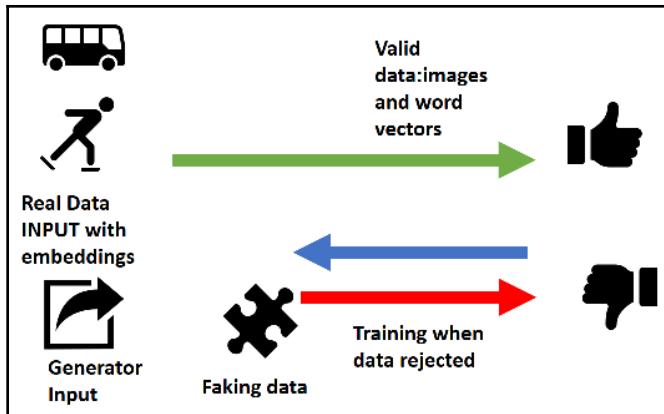
Using GANs to generate concept-word images can boost the production CRL datasets and produce millions of concept-word images. No single solution will suffice. CNNs can do the job; GANs can also.

Suppose the Γ **gap** concept needs to be generalized to thousands of situations. Millions of images with words could be analyzed and automatically stored in a Γ **dataset**, with tens of thousands of image-word concepts of gaps.

A GAN could be built in a few lines with the following components:

- Input data for the generator
- The generator
- Real data input
- A discriminator (detects fake data from real data)
- An optimizer (training)

The network starts by inputting real data and fake data (with the generator), as shown in the following diagram:



The discriminator function accepts or rejects the generated data. If the data is rejected, then we are back to known territory (see Chapter 9, *Getting Your Neurons to Work*) with cross-entropy to measure the loss and an Adam optimizer, for example.

The system will train until it distinguishes right images from fake images like other networks.



What makes GANs interesting is the fact that the generator data has no direct connection with the real data.

It has to go through the discriminator and be trained if it's wrong. Furthermore, the generated data can be labeled with accurate or inaccurate labels although the real data is correctly labeled.

Generating conceptual representations

A GAN can be used to generate the large datasets required for CRLMMs. The real data could be a set of images with the correct word vectors. The generator could then produce millions of images and labels that the discriminator would work on. The accurately generated datasets could be stored in a CRL repository.

A GAN can be implemented in a few lines of code in TensorFlow, for example. The *Further reading* section at the end of the chapter provides links that detail this simple but effective type of network.

The use of autoencoders

The use of autoencoders must be explored although not recommended for the generation of a conceptual learning dataset.

Data compression has become a part of our daily life through MP3 algorithms, for example. Autoencoding in machine learning is a data compression algorithm.

It takes the original data, compresses it, and can then reconstruct it:

- Original input
- The **encoder** compresses the data (compressed representation)
- The **decoder** reconstructs the data
- The reconstructed output is produced

They were used for CNNs at a time for data representations. But as shown in Chapter 9, *Getting Your Neurons to Work*, random weight initialization scenarios are as effective if not more reliable and flexible.

Two main limitations of autoencoders have restrained their use:

- Autoencoders need to be trained on **specific** data. Being data-specific makes transfer learning tedious.
- The decompressed output is not 100% reliable. They are **lossy**, which means that information is lost during the encoding-decoding phase, unlike standard lossless compression algorithms.

Autoencoders can be used in some cases of dimensionality reduction for data visualization and data denoising.

Beyond that, present-day random weight initialization methods are favored for CNN applications.

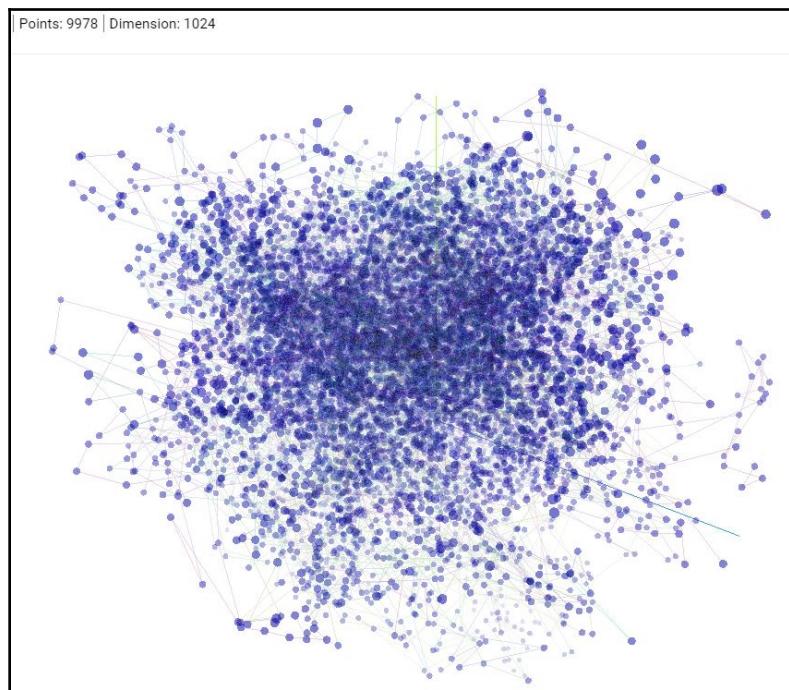
The motivation of conceptual representation learning meta-models

A CRLMM converts images into concepts. These abstract concepts will then be embedded in vectors that become logits for a softmax function and in turn be converted into parameters for complex artificial intelligence programs.

At one point, in some artificial intelligence projects, dimension reduction does not produce good results. When scheduling maintenance on airplanes, rockets, and satellite launcher, for example, thousands of features enter the system without leaving any out. A single missing screw in the wrong place can create a disaster.

The curse of dimensionality

The features for a given project can reach large numbers. The following example contains 1,024 dimensions:



In Chapter 9, *Getting Your Neurons to Work*, and Chapter 10, *Applying Biomimicking to Artificial Intelligence*, kernels and pooling reduced the number of dimensions to extract key features. With those key features in tensors, arrays and/or vectors, classification and prediction processes are possible. This process produces effective results in many areas.

The limit of this approach occurs when the result does not reach expectations, such as in some of the cases we will discover in the following chapters. In those cases, CRLMMs will increase the productivity of the solution providing a useful abstract model.

When a solution requires a large number of unreduced dimensions, kernels, pooling, and other dimension reduction methods cannot be applied. CRLMMs will provide a *pair of glasses* to the system.

The blessing of dimensionality

In some projects, when the model reaches limits that comprise a project, dimensionality is a blessing. Let's focus on three examples described in the next chapters: scheduling, chatbots, and self-driving cars.

Scheduling and blockchains

An artificial neural network (ANN) system built on dimension reduction might sometimes be seriously underfitted. As long as it was trained with ready-to-use datasets (MNIST, CIFAR, and others), the system worked fine. It even ran perfectly on small datasets.

Then all of a sudden, on the first week in the food processing industry, for example, it mistakes a positive gap (an available resource) for a negative gap (not enough production).

For example, since a gap meant missing products for several days, the AI system wrongly predicted a negative gap when it was not supposed to. In this circumstance, this was a deliberate choice of the management to slow production down due to resource shortages. Dimensionality enhancements will solve the problem.

Humans reduce the number of possibilities of a problem with logic and inbuilt decision trees. Machines can beat humans in some forms of computation with no logic at all using random memoryless algorithms (MDP; see Chapter 1, *Become an Adaptive Thinker*).

However, present-day deep learning networks, however innovative they are, reduce the dimensions of representation, layer by layer. They flatten them out, drop out information and more to reach a result.



The future of AI will no doubt rely on increasing the number of dimensions way beyond human limits and performing powerful calculations without present-day levels of reduction. Today's state-of-the-art is tomorrow's past. Blow the number of dimensions up to indefinite numbers when it is necessary and possible!

Chatbots

Most NLP algorithms use probabilities to predict the next words. Or they use preset sentences to answer questions *without analyzing the real meaning*.

This means building systems that guess, sometimes effectively, sometimes with errors. In any case, they do not *think*. Many applications produce good results with statistical approaches. But for some projects, customizing a chatbot comes in handy.

For example, imagine that somebody is speaking to you in a language you do not know at all. You have the inbuilt ability of present NLP algorithms to predict representations through statistics on your smartphone.

Although you do not understand a single word, you produce words, statistically, that could fit the situation. For isolated words and simple sentences, you are doing well.

Then all of a sudden, the person frowns. You are totally off track, like many NLP systems today. The person is astonished by your ability to speak and has just said in French, for example, "tu as la patate!"

You understand what Google Translate tells you: "*You have the potato!*" You have a big question mark on your face. The other person frowns. Oops!

You just spoke in body language: your question mark, the person's frown. Body language constitutes a major part of real communication between humans. Body language needs to be compensated for with empathy as we will explore in the chatbot chapters.

In the meantime, in French colloquial or slang language *patate* means *pep* (lots of energy) in English.

Classifying documents, finding keywords, and doing first-level translations (see Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies*) is one thing; building a smart chatbot beyond language statistics when necessary is another. A CRLMM solution will provide a deeper interaction between a system and humans such as described in the chatbot chapters.

Self-driving cars

Self-driving cars will progressively drive everywhere for everything. In the meantime, accidents caused by self-driving cars grab the headlines. Let's imagine some solutions to help them avoid accidents and improve maintenance problems with small but effective CRLMM enhancements.



Humans increase dimensionality to solve problems, find new ideas, clear misunderstandings up, and adapt.

The CRLMMs built as add-ons on the awesome tools we all have access to, give us the opportunity to push the limits of AI further every day.

Summary

In this chapter, the **Convolution Neural Network (CNN)** architecture built in [Chapter 9, Getting Your Neurons to Work](#), was loaded to classify physical gaps in a food processing company.

Then the trained models were applied to transfer learning by identifying similar types of images. Some of these images represented concepts that lead the trained CNN to identify **concept gaps**.

concept gaps were applied to different fields using the CNN as a training and classification tool in domain learning.

concept gaps have two main properties: negative n-gaps and positive p-gaps. To distinguish one from the other, a CRLMM provides a useful add-on. In the food processing company, installing a webcam on the right food processing conveyor belt provided a context for the system to decide whether the gap was positive or negative.

12

Optimizing Blockchains with AI

IBM was founded in 1911, making it the most experienced company today in its field. Google, Amazon, and Microsoft also offer history-making machine learning platforms. However, IBM offers machine learning platforms with its valuable 100+ years of experience.

Some of IBM's 100+ years in the computer and software market were bumpy. Some terrible decisions cost the company a lot of problems across the world. IBM learned from those mistakes and now offers robust solutions:

- IBM Hyperledger for blockchain solutions (this chapter)
- IBM Watson for artificial intelligence applications, including chatbots (see Chapter 13, *Cognitive NLP Chatbots*, and Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*)

This chapter offers an overview of what blockchains mean for companies around the world, introduces naive Bayes, and explains where it could fit in a CRLMM.

Mining cryptocurrency represents one side of the use of blockchains. IBM advocates using blockchains for corporate transactions and brings 100+ years of experience into the field.

The following topics will be covered in this chapter:

- Using blockchains to mine bitcoins
- Using blockchains for business transactions
- How the blocks of a blockchain provide a unique dataset for artificial intelligence
- Applying artificial intelligence to the blocks of a blockchain to predict and suggest transactions
- Naive Bayes
- How to use naive Bayes on blocks of a blockchain to predict further transactions and blocks

Technical requirements

Python 3.6x 64-bit from <https://www.python.org/>.

Package and modules:

```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import GaussianNB
```

Programs from GitHub Chapter14:

naive_bayes_blockchains.py

Check out the following video to see the code in action:

<https://goo.gl/kE3yNc>

Blockchain technology background

Blockchain technology will transform transactions in every field.

For 1,000+ years, transactions have been mostly local bookkeeping systems. For the past 100 years, even though the computer age changed the way information was managed, things did not change that much. Each company continued to keep its transactions to itself, only sharing some information through tedious systems.

Blockchain makes a transaction **block** visible to the global network it has been generated in.

The fundamental concepts to keep in mind are **sharing** and **privacy control**. The two ideas seem to create a cognitive dissonance, something impossible to solve. Yet it has been solved, and it will change the world.

When a block (a transaction of any kind) is generated, it is shared with the whole network. Permissions to read the information within that block remain manageable and thus private if the regulator of that block wants the information to stay private.

Whether the goal is to mine bitcoins through blocks or use blocks for transactions, artificial intelligence will enhance this innovation shortly.

Mining bitcoins

Mining a bitcoin means creating a mathematical block for a valid transaction and adding this block to the chain, the blockchain:

$$\text{Blockchain} = \{\text{block}_1, \text{block}_2, \text{the block just added...}, \text{block}_n\}$$

The blockchain cannot go back in time. It is like a time-dating feature in life. On minute m , you do something, on minute $m+1$ something else, on minute $m+n$ something else, and so on. You cannot travel back in time. **What is done is done.**

When a block is added to the bitcoin chain, there is no way of undoing the transaction.

The global network of bitcoin mining consists of **nodes**. With the appropriate software, you leave a port open, allocate around 150+ GB of disk space, and generate new blocks. The nodes communicate with each other, and the information is relayed to the other nodes around the whole network.

For a node to be a miner, it must solve complex mathematical puzzles that are part of the bitcoin program.

To solve the puzzle, the software must find a number that fits in a specific range when combined with the data in the block being generated. The number is passed through a hash function.

You can call the number a **nonce** and it's used only once. For example, an integer between 0 and 4,294,967,296 for a bitcoin must be generated.

The process is random. The software generates a number, passes it through the hash function, and sends it out to the network. The first **miner** who produces a number in the expected range informs the whole network that that particular block has been generated. The rest of the network stops working on that block and moves on to another one.

The reward for the miner is naturally paid out in bitcoins. It represents a lot of money, but it's hard to get, considering the competition in the network and the cost required (CPU, electricity, disk space, and time) to produce correct results.

Using cryptocurrency

Be very careful with cryptocurrency. The concept sounds fascinating, but the result remains currency. Currency can be volatile, and you can lose your life's savings in less than an hour if a crash occurs.



Golden rule: If you cannot resist investing in cryptocurrencies, do not invest more than you can afford to lose.

That being said, to use cryptocurrency, first set up a wallet to store your bitcoins, for example. The wallet can be online, through a provider, or even offline.

Once that is done, you will be able to purchase bitcoins as you wish in hard cash, or using credit cards, debit cards, and transfers.

Remember, you are buying these currencies like any other currency with all the potential, but also all the risks, involved.

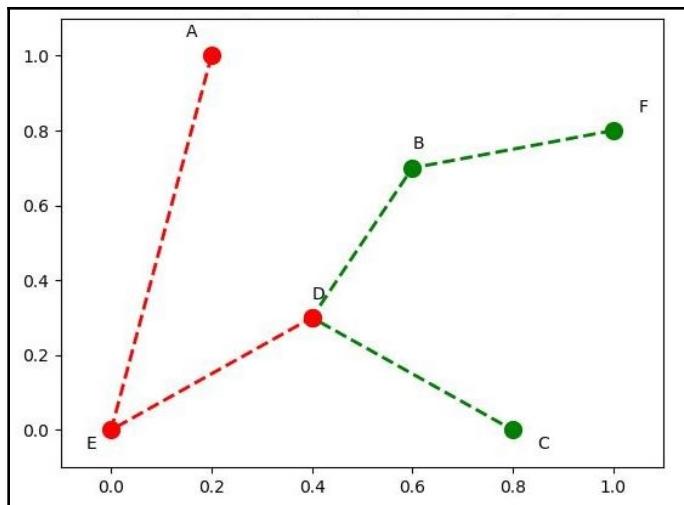
Using blockchains

IBM blockchain, based on Hyperledger Fabric, provides a way for companies around the world to share a blockchain transaction network without worrying about mining or using cryptocurrencies.

The system is based on the Linux Foundation project. Hyperledger is an open source collaborative project hosted by the Linux Foundation.

At this level, Hyperledger uses blockchains to guarantee secure transactions without trying to optimize the cryptocurrency aspect. The software generates blocks in a blockchain network shared by all the parties involved, but they do not have to purchase cryptocurrencies in the currency sense—only in the technological sense.

The following graph, used in previous chapters to illustrate the Markov Decision Process, is applied to a blockchain.



Markov Decision Process graph

Each vertex represents a company that takes part in an IBM Hyperledger network set up for those six companies, as described in the following table:

Company	Activity	Weight in a CRLMM
A buys and sells clothing and also other products in this network.	Provide goods to the network but keep the stock levels down	Stock levels
B buys and sells fabric and also other products in this network.	Provide goods to the network but keep the stock levels down	Stock levels
C buys and sells buttons and also other products in this network.	Provide goods to the network but keep the stock levels down	Stock levels
D buys and sells printed fabric and also other products in this network.	Provide goods to the network but keep the stock levels down	Stock levels
E buys and sells accessories (belts, bracelets) and also other products in this network.	Provide goods to the network but keep the stock levels down	Stock levels
F buys and sells packaging boxes and also other products in this network.	Provide goods to the network but keep the stock levels down	Stock levels

With millions of commercial transactions per year with a huge amount of transportation (truck, train, boat, air), it is impossible to manage this network effectively in the 21st century without a solution like IBM Hyperledger.



With IBM Hyperledger, the companies have **one** online transaction ledger with smart contracts (online) and real-time tracking.

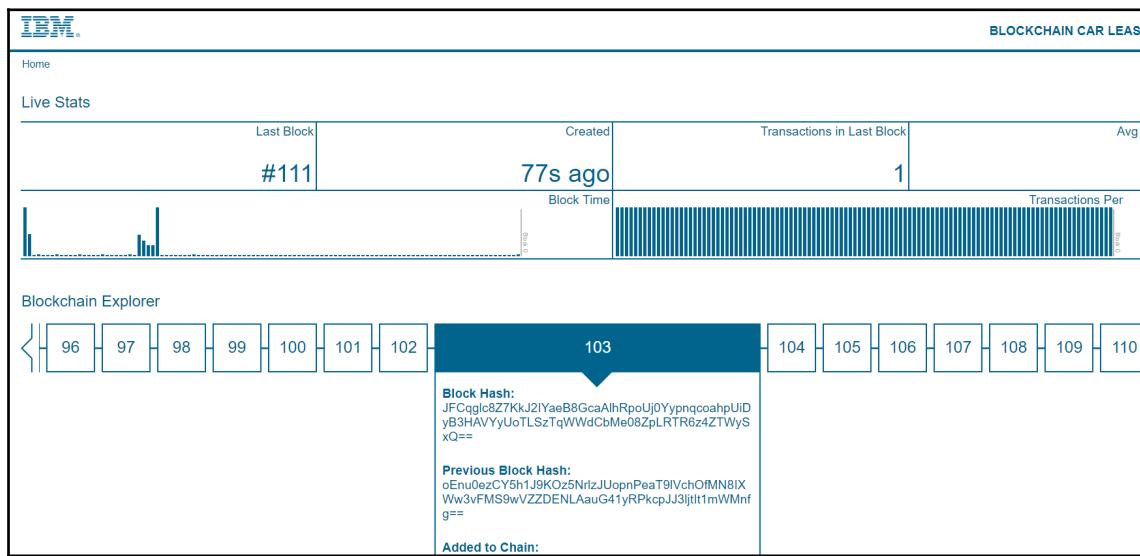
The transactions are secure; they can be private or public among the members of the network and they provide real-time optimization information for an artificial intelligence solution.

Using blockchains in the A-F network

IBM Hyperledger provides artificial intelligence developers with a unique advantage over any other dataset—a 100% reliable dataset updated in real time.

Creating a block

A block is formed with the method described for mining a bitcoin, except that this time currency is not a goal. The goal is a secure transaction with a smart contract when necessary. The following screenshot is a standard IBM interface that can be customized:



Standard IBM interface

You can see the individual and unique blocks that make up a blockchain. Each block in IBM Hyperledger has a unique number. In this example, it is 111 with a zoom on block 103.

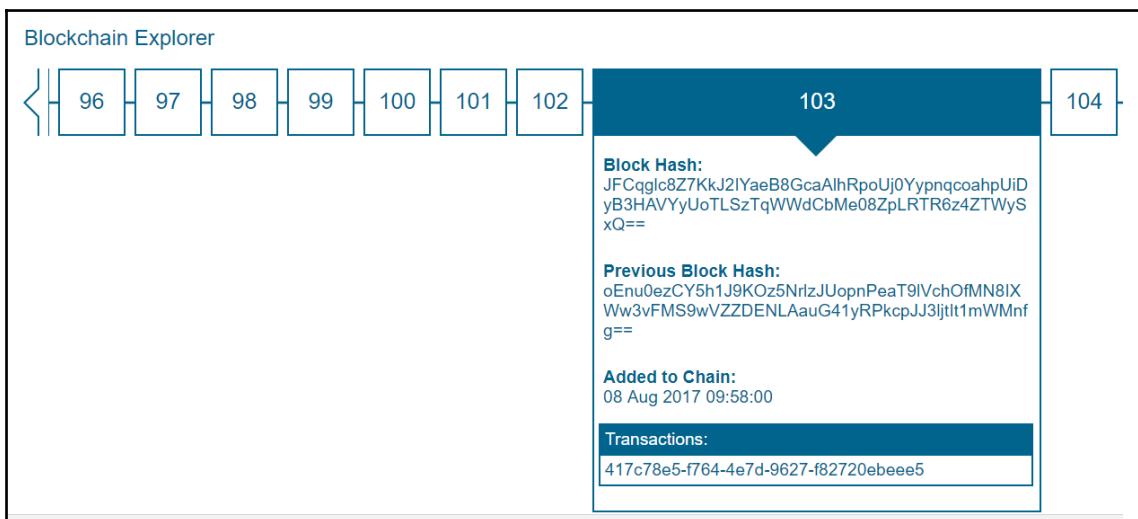
A block in the preceding A to F example can be the purchase of a product X with a contract. The next block can be the transportation of that product to another location and any transaction within the A-F blockchain network.

The information attached to that block is in the Hyperledger repository: company name, address, phone number, transaction description, and any other type of data required by the network of companies. Each block can be viewed by all or some depending on the permission properties attached to it.

Exploring the blocks

Exploring the blocks provides an artificial intelligence program with a gold mine: a limitless, real-life, and 100% reliable dataset.

The interesting part for AI optimization is the block information, as described in the following screenshot. The present block was added to the chain along with the previous block and the transaction code.



Notice that the block hash (see the preceding mining section) of a given block is linked to the previous block hash and possesses a unique transaction code.

The CRLMM will use some of the critical information as **features** to optimize warehouse storage and product availability in a real-time process.

To implement the CRLMM, an additional component is required: a naive Bayes learning function. The naive Bayes learning function will learn the previous block to predict and output the next blocks that should be inserted in the blockchain.

Using naive Bayes in a blockchain process

Naive Bayes is based on Bayes' theorem. Bayes' theorem applies conditional probability, defined as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- $P(A|B)$ is a **posterior probability**, the probability of A after having observed some events (B). It is also a **conditional probability**: the likelihood of A happening given B has already happened.
- $P(B|A)$ the probability of B given the prior observations A . It is also a conditional probability: the likelihood of B happening given A has already happened.
- $P(A)$ is the probability of A prior to the observations.
- $P(B)$ is the probability of the predictions.

Naive Bayes, although based on Bayes' theorem, assumes that the features in a class are **independent** of each other. In many cases, this makes predictions more practical to implement. The statistical presence of features, related or not, will produce a prediction. As long as the prediction remains sufficiently efficient, naive Bayes provides a good solution.

A naive Bayes example

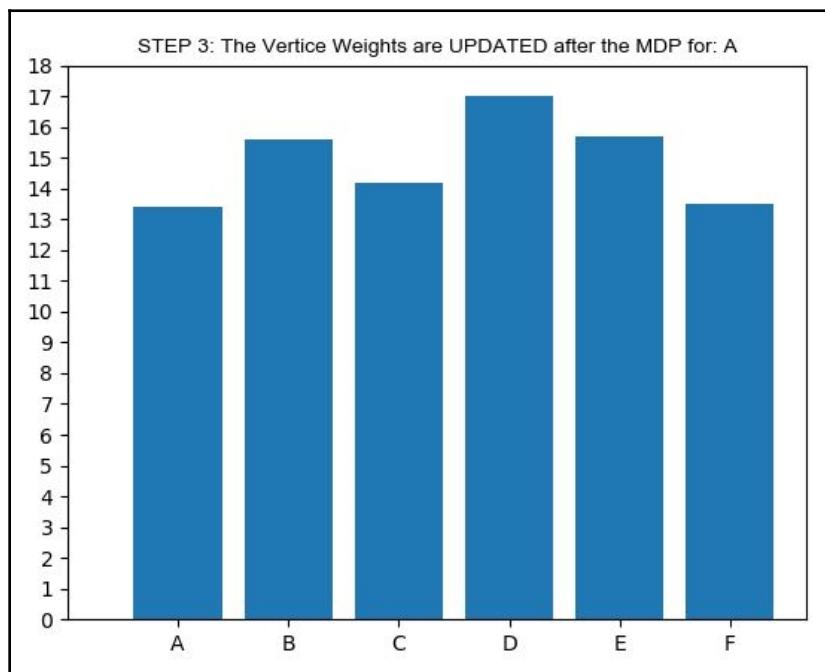
The load of a sewing station in an apparel industry is expressed in quantities, in **stock keep units (SKUs)**. An SKU, for example, can be product P: a pair of jeans of a given size.

Once the garment has been produced, it goes into storage. At that point, a block in a blockchain can represent that transaction with two useful features for a machine learning algorithm:

- The day the garment was stored
- The total quantity of that SKU-garment now in storage

Since the blockchain contains the storage blocks of all A, B, C, D, E, and F locations that are part of the network, a machine learning program can access the data and make predictions.

The goal is to spread the stored quantities of the given product evenly over the six locations, as represented in the following histogram:



This screenshot shows the storage level of product P distributed over six locations. Each location in this blockchain network is a **hub**. A hub in **supply chain management (SCM)** is often an intermediate storage warehouse. For example, to cover the area of these six locations, the same product will be stored at each location. This way, local trucks can come and pick the goods for delivery.

The goal is to have an available product P at point A when needed. The delivery time from A to a location point a_1 (a store or home) will only take a few hours. If A did not store P, then the finer consumer would have to wait for the product to travel from C to A, for example.

If the blockchain network is well organized, one location can specialize in producing product P (best production costs) and evenly distribute the quantity stored in the six locations, including itself.

Reusing a model in domain learning will generate profit and lower costs. The more an AI solution can be applied to transfer, domain, and conceptual learning, the more powerful it will become. It will learn more and solve an endless number of problems.

The blockchain anticipation novelty

In former days, all the warehouses at those six locations (A to F) had to ensure the minimum storage level for each location.

In a world of real-time producing and selling, distributors need to predict **demand**. The system needs to be **demand driven**. Naive Bayes can solve that problem.

It will take the first two features into account:

- **DAY**: The day the garment was stored
- **STOCK**: The total quantity of that SKU-garment now in storage

Then it will add a novelty—the number of blocks related to product P.

A high number of blocks at a given date means that this product was in demand in general (production, distribution). The more blocks there are, the more transactions there are. Also, if the storage levels (STOCK feature) are diminishing, this is an indicator; it means storage levels must be replenished. The DAY feature time-stamps the history of the product.

The block feature is named **BLOCK**. Since the blockchain is shared by all, a machine learning program can access reliable global data in seconds. The dataset reliability provided by blockchains constitutes a motivation in itself to optimize blocks.

The goal

This CRLMM is the same one as in Chapter 11, *Conceptual Representation Learning*. The difference is that instead of analyzing a conveyor belt to make a decision, the system will analyze the blocks of a blockchain.

The program will take the DAY, STOCK, and BLOCKS (number of) features for a given product P and produce a result. The result predicts whether this product P will be in demand or not. If the answer is yes or 1, the demand for this product requires anticipation.

Step 1 the dataset

The dataset contains raw data from prior events in a sequence, which makes it perfect for prediction algorithms. This constitutes a unique opportunity to see the data of all companies without having to build a database. The raw dataset will look like the following list:

BLOCKS	STATUS
Blocks	No
Some_blocks	Yes
No_Blocks	Yes
Blocks	Yes
Blocks	Yes
Some_blocks	Yes
Blocks	Yes
No_Blocks	No
Blocks	Yes
No_Blocks	Yes

Raw dataset

This dataset contains the following:

- **Blocks** of product P present in the blockchain on day x scanning the blockchain back by 30 days. **No** means no significant amounts of blocks has been found. **Yes** means a significant number of blocks have been found. If blocks have been found, this means that there is a demand for this product somewhere along the blockchain.

- **Some_blocks** means that blocks have been found, but they are too sparse to be taken into account without overfitting the prediction. However, yes will contribute to the prediction as well as no.
- **No_blocks** means there is no demand at all, sparse or not (**Some_blocks**), numerous (blocks) or not. This means trouble for this product P.

The goal is to avoid predicting demand on sparse (**Some_blocks**) or absent (**No_blocks**) products. This example is trying to predict a potential Yes for numerous blocks for this product P. Only if **Yes** is predicted can the system trigger the automatic demand process (see the *Implementation* section later in the chapter).

Step 2 frequency

Looking at the following frequency table provides additional information:

STATUS:	No		Yes		Yes	
	Some_blocks	No_Blocks	Blocks	Some_blocks	No_Blocks	Blocks
			1			
				1		
					1	
						1
						1
				1		
						1
			1			
						1
FREQUENCY	0	1	1	2	2	4

Frequency table

The **Yes** and **No** statuses of each feature (**Blocks**, **Some_blocks**, or **No_blocks**) for a given product P for a given period (past 30 days) have been grouped by frequency.

The sum is on the bottom line for each no feature and yes feature. For example, **Yes** and **No_Blocks** add up to 2.

Some additional information will prove useful for the final calculation:

- The total number of samples = 10
- The total number of yes samples = 8
- The total number of no samples = 2

Step 3 likelihood

Now that the **frequency** table has been calculated, the following **likelihood** table is produced using that data:

Weather	No	Yes		
Some block	0	2	2	0,2
No Blocks	1	2	3	0,3
Blocks	1	4	5	0,5
	2	8		
	0,2	0,8		

Likelihood table

The table contains the following statistics:

- $No = 2 = 20\% = 0.2$
- $Yes = 8 = 80\% = 0.8$
- $Some\ blocks = 2 = 20\% = 0.2$
- $No\ blocks = 3 = 30\% = 0.3$
- $Blocks = 5 = 50\% = 0.5$

Blocks represent an important proportion of the samples, which means that along with some blocks, the demand looks good.

Step 4 naive Bayes equation

The goal now is to represent each variable of the Bayes' theorem in a naive Bayes equation to obtain the probability of having **demand** for product P and trigger a purchase scenario for the blockchain network. Bayes' theorem can be expressed as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- $P(Yes|Blocks) = P(Blocks|Yes) * P(Yes)/P(Blocks)$
- $P(Yes) = 8/10 = 0.8$
- $P(Blocks) = 5/10 = 0.5$
- $P(Blocks|Yes) = 4/8 = 0.5$
- $P(Yes|Blocks) = (0.5 * 0.8)/0.5 = 0.8$

The demand looks acceptable. However, penalties must be added and other factors must be considered as well (transportation availability through other block exploration processes).

This example and method show the concept of the naive Bayes approach. However, scikit-learn has excellent scientific functions that make implementation easier.

Implementation

This section shows how to use an advanced version of naive Bayes and where to insert this component in the CRLMM described in Chapter 11, *Conceptual Representation Learning*.

Gaussian naive Bayes

In implementation mode, a dataset with raw data from the blockchain will be used without the feature interpretation function of naive Bayes in the following table:

DAY	STOCK	BLOCKS	DEMAND
10	1455	78	1
11	1666	67	1
12	1254	57	1
14	1563	45	1
15	1674	89	1
10	1465	89	1
12	1646	76	1
15	1746	87	2
12	1435	78	2

Each line represents a block:

- **DAY**: The day of the month scanned (dd/mm/yyyy can be used beyond a prototype)
- **STOCK**: The total inputs in a given location (A, B, or... F) found in the blocks and totaled on that day
- **BLOCKS**: The number of blocks containing product P for location A, for example

A high number of blocks in the BLOCK column and a low number of quantities in the STOCK column mean that the demand is high.

- **DEMAND = 1.** The proof of demand is a transaction block that contains a purchase in the past. These transaction blocks provide vital information.

A low number of blocks in the BLOCK column and a high number of quantities in the STOCK column mean that the demand is low.

- **DEMAND = 2.** Proof that no transaction was found.

However, in some cases, *DEMAND = 1* when the stock is high and the blocks are low. That's why strict correlation is not so useful where naive Bayes just analyzes the statistics and learns how to predict, ignoring the actual conditional probabilities.

The Python program

The Python `naive_bayes_blockchains.py` program uses a `sklearn` class. Consider the following snippet:

```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import GaussianNB
```

It reads the dataset into the data structure. The following code reads `data_BC.csv` into `df`:

```
#Reading the data
df = pd.read_csv('data_BC.csv')
print("Blocks of the Blockchain")
print (df.head())
```

It prints the top of the file in the following output:

```
Blocks of the Blockchain
DAY STOCK BLOCKS DEMAND
0 10 1455 78 1
1 11 1666 67 1
2 12 1254 57 1
3 14 1563 45 1
4 15 1674 89 1
```

It prepares the training set, using X to find and predict Y in the following code:

```
# Prepare the training set
X = df.loc[:, 'DAY':'BLOCKS']
Y = df.loc[:, 'DEMAND']
```

It chooses the class and trains the following clfG model:

```
#Choose the class
clfG = GaussianNB()
# Train the model
clfG.fit(X,Y)
```

The program then takes some blocks of the blockchain, makes predictions, and prints them using the following clfG.predict function:

```
# Predict with the model(return the class)
print("Blocks for the prediction of the A-F blockchain")
blocks=[[28,2345,12],
        [29,2034,50],
        [30,7789,4],
        [31,6789,4]]
print(blocks)
prediction = clfG.predict(blocks) I
for i in range(4):
    print("Block #",i+1," Gauss Naive Bayes
Prediction:",prediction[i])
```

The blocks are displayed and the following predictions are produced. 2 means no demand for the moment; 1 will trigger a purchase block:

```
Blocks for the prediction of the A-F blockchain
[[14, 1345, 12], [29, 2034, 50], [30, 7789, 4], [31, 6789, 4]]
Block # 1 Gauss Naive Bayes Prediction: 1
Block # 2 Gauss Naive Bayes Prediction: 2
Block # 3 Gauss Naive Bayes Prediction: 2
Block # 4 Gauss Naive Bayes Prediction: 2
```

This is a replenishment program. It will mimic the demand. When no demand is found, nothing happens; when a demand is found, it triggers a purchase block. Some chain stores know the number of garments purchased on a given day (or week or another unit) and automatically purchase that amount. Others have other purchasing rules. Finding business rules is part of the consulting aspect of a project.

Summary

The reliable sequence of blocks in a blockchain has opened the door to endless machine learning algorithms. Naive Bayes appears to be a practical way to start optimizing the blocks of a blockchain. It calculates correlations and makes predictions by learning the independent features of a dataset, whether the relationship is conditional or not.

This freestyle prediction approach fits the open-minded spirit of blockchains that propagate by the millions today with limitless resources.

IBM Hyperledger takes blockchain's "Frontierland" development to another level with the Linux Foundation project. IBM also offers a cloud platform and services.

IBM, Microsoft, Amazon, and Google provide cloud platforms with an arsenal of disruptive machine learning algorithms. This provides a smooth approach to your market or department, along with the ability to set up a blockchain prototype online in a short time. With this approach, you can enter some real prototype data in the model, export the data, or use an API to read the block sequences. Then you will be able to apply machine learning algorithms to these reliable datasets.

The limit is our imaginations.

The next chapter will first take us into another IBM innovation, IBM Watson, applied to chatbots, and then beyond... into cognitive **natural language processing (NLP)**.

13

Cognitive NLP Chatbots

IBM Watson on IBM Cloud provides a set of tools to build a cognitive NLP chatbot. IBM proposes a well-thought process to design a chatbot. Good architecture remains the key to a chatbot's success.

To make a chatbot meet user demands, IBM Watson has many services libraries (speech to text, text to speech, and much more).

The whole process can be customized with one's programs. This makes IBM Cloud an excellent place to start creating chatbots. The TextBlob project offers another set of NLP modules that provide some interesting ways of improving a chatbot.

This chapter starts by setting up an IBM Watson chatbot following IBM's standard procedure and shows how to use IBM Watson to add one's cognitive chatbot functions. A cognitive service in Python will be described to solve a case study. The problem to address is that of a tourist on a bus that just broke down. He needs to communicate with a police chatbot to get help. The solution uses a CNN, Google Translate, **conceptual representation learning (CRL)** cognitive techniques, and sentiment analysis.

The following topics will be covered in this chapter:

- IBM Watson's chatbot solution
- Defining intents and entities
- Defining a dialog flow
- Adding scripts to the dialog flow
- Customizing scripts for a chatbot
- Using sentiment analysis to improve some dialogs
- Using **conceptual representation learning meta models (CRLMM)** to enhance some dialogs

Technical requirements

Python 3.6x 64-bit from <https://www.python.org/>.

Package and modules:

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import matplotlib.pyplot as plt
import keras
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
from keras.models import model_from_json
from keras.models import load_model
import numpy as np
import scipy.misc
from PIL import Image
import time
from textblob import TextBlob
```

Program: GitHub Chapter15/CRL-MM-Chabot.py.

Check out the following video to see the code in action:

<https://goo.gl/R7qHk6>

IBM Watson

IBM's Watson conversation service has many development tools for creating a chatbot with compatible standard services (translations, digitized voice, voice recognition, and so on). You can set up your own basic account in a few clicks.

To create a chatbot, you need three major objects: **intents**, **entities**, and a **dialog** flow. To complete the dialog flow, there are additional features: conditions, context, and expressions for example. Then services can be added such as digitized voice, speech recognition, and more.

This section describes how to build an interview chatbot named **HobChat**, a diet bot that can interview users.

Intents

An **intent** represents the initial subject of the conversation that the chatbot is initiating. The user input needs to be **classified** just as images need to be in a CNN, for example.

Let's say HobChat needs two types of information from the user:

#whatkindof and #food

You'll notice a fundamental concept here. We are getting information from a person and not the other way around. In many chatbots, the user asks a question to obtain an answer. HobChat works the other way around. It's extracting information through interrogation techniques.

Chatbots have different points of view:

- Providing information for a user: tourism, online shopping, and other services
- Obtaining information from a user: medical applications, dietitians, sports trainers, and marketing
- Providing and gathering information

In this case, Hobchat's **intent** is to learn about *#whatkindof #food* the user likes. HobChat is prepared for the user to either answer or ask a question. To do that, we set up a list of variations of *#whatkindof* in the system in the following intents interface:

The screenshot shows the Watson Conversation interface under the 'Build' section. The 'Intents' tab is active. An intent named '#whatkindof' is selected. Below the intent name, there is a 'User example' field with the placeholder 'Add a user example...'. Underneath the field, three examples are listed: 'what kind of', 'what type of', and 'what sort of', each followed by a small circular delete icon.

Intents interface

The intent name `#whatkindof` will be detected if a similar phrase comes up:

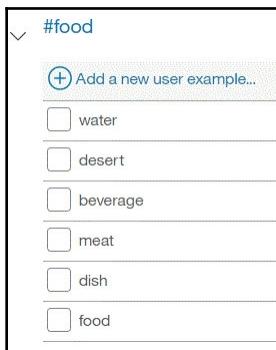
`#whatkindof={what kindof, what type of, what sort ...n}`

The number of sentences depends on the final use of the application. One similar phrase could suffice in some cases; in others, hundreds are required.

In this example, we can see that if HobChat asks *What kind of food do you like?* and if a user answers *What type of what?*, then the chatbot will apply a function f to the utterance.

$f(\text{utterance})$ will provide the intent. $f(\text{What kind of what?})$, for example, would be parsed and classified in the `#whatkindof` intent.

The same has to be done with `#food`, for example, as shown in the following screenshot:



Food intent

The `#food` intent subset is now ready to use:

$\text{food} = \{\text{water, dessert, beverage, meat, dish, food}\}$

Now that some intents have been created, it's time to test these subsets.

Testing the subsets

IBM Watson provides a dialog tool to test the chatbot throughout the creation process. The dialog tool can be set up intuitively in a few steps, as described in the following dialog section.

The test dialog will run an initial dialog in the following dialog testing interface:



Dialog testing interface

We can see that HobChat was activated to start a conversation called **food interrogation**. HobChat classifies the chatbot subject in `#food` (blue in the screenshot) and starts the interview by asking a sort of generic text blob question **What kind of food do you like to eat?**.

This interview technology will also be beneficial for medical applications such as doctors asking patients questions to classify types of pathologies.

The user answers, "What?"

"What?" is processed as $f(\text{utterance} = f(\text{what})) = \#whatkindof$.

To pursue this conversation, the chatbot needs **entities**.

Entities

Now, we need to clarify Hobchat's `#food` intent. What exactly are we talking about? What kind of food? To clarify our intent, we need entities that will describe our types of food in the following interface:

@foodtypes				
+ Add a new value				
<input type="checkbox"/>	salad	iceberg lettuce	green stuff	iceberg lettuce
<input type="checkbox"/>	meat	T-Bone	hamburger, chee...	
<input type="checkbox"/>	french fries	potatoes	chips	greasy potatoes

Food intent interface

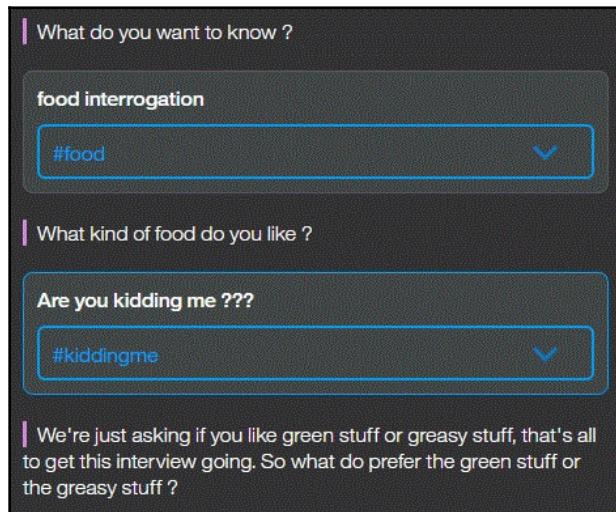
Our food type entity is `#foodtypes={salad, meat, french fries}`. We also need as many synonyms for each member of the `#foodtype` group :

`salad={iceberg lettuce, green stuff, iceberg, lettuce}`

`meat = {T-Bone, hamburger, cheeseburger}`

`french fries={potatoes, chips, greasy potatoes, greasy}`

Now we can clarify our dialog with the food type entity in the following interface:



Food type entity interface

We've gone through the first part. Hobchat has asked the user, a person who visibly does not like such questions, *What kind of food do you like?*

A surprise entity was added so that *Hobchat* can detect surprise (emotion) in the answer:

The user answers: *Are you kidding me?*

Real-life conversations involve emotional answers a chatbot must deal with.

The chatbot offers a calm response: *We're just asking if you like green stuff or greasy stuff, that's all to get this interview going. So what do you prefer, the green stuff or the greasy stuff?*

To find out what's going to happen next, we have to set up the dialog flow.

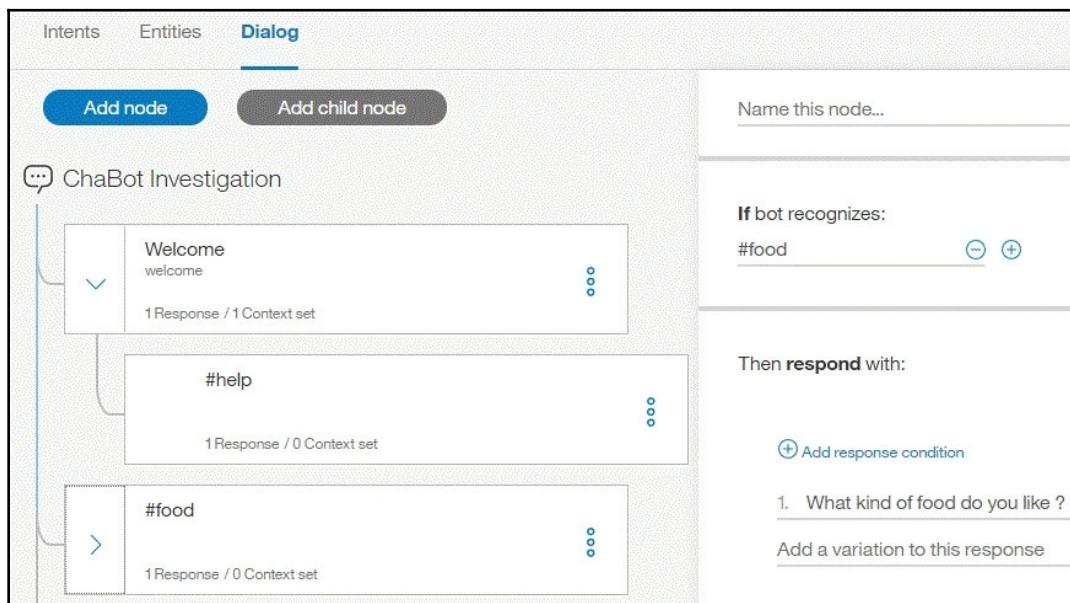
Dialog flow

IBM Watson has an interface to configure the dialog flows shown previously.

To teach your chatbot how to conduct an interview using IBM Watson, or any other solution, requires a lot of cognitive science work to represent the expertise of a given field. Cognitive science involves linguistics, anthropology, psychology, philosophy, analyzing intelligence, and more.

The next section shows how to use services to implement a cognitive approach to solving a problem encountered by a chatbot.

First, a basic dialog flow must be configured. We are ready to create dialog scenarios for Hobchat. An IBM Watson conversation provides an efficient, ready-to-use platform, as shown in the following screenshot:



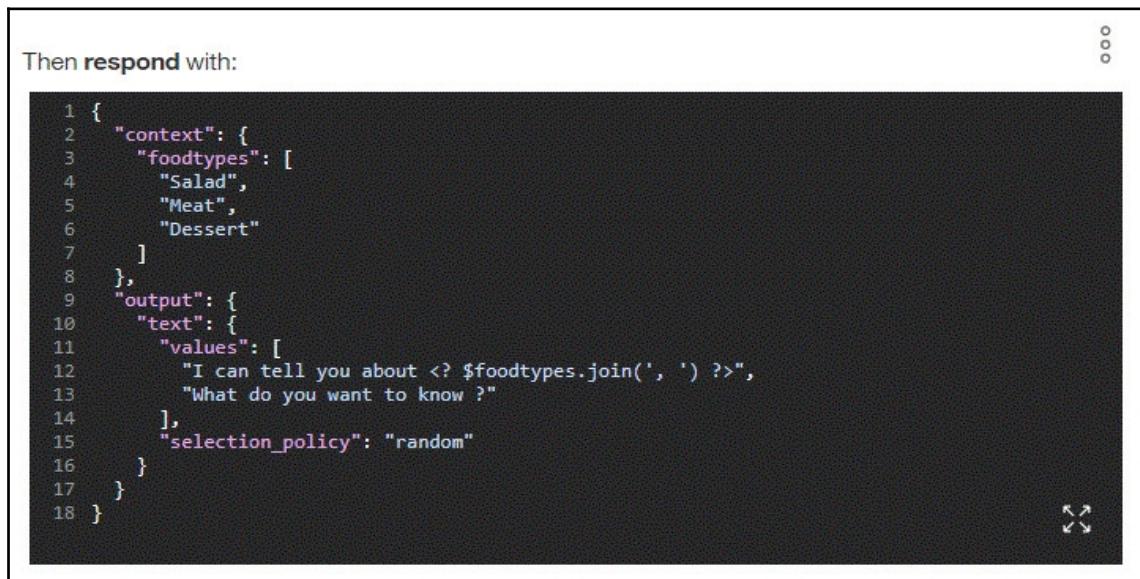
IBM Watson conversation flow chart

As you can see, it's a complex type of flowchart. You set up the dialog using your intents and entities, with all the possible linguistic variations you can provide. Then you can create an unlimited number of nodes in a combinatory tree.

You can add conditions, expressions, context, and also script as much as you want.

Scripting and building up the model

In the preceding dialog flow graph, you can enter scripts. For example, at *then respond with* nodes, you can write your script as shown in the following script-editing interface:



The screenshot shows a dark-themed script-editing interface with a text area containing JSON-like code. The text area has a scroll bar on the right. At the top left, there is a label "Then respond with:" followed by a colon. The code itself is as follows:

```
1 {  
2   "context": {  
3     "foodtypes": [  
4       "Salad",  
5       "Meat",  
6       "Dessert"  
7     ]  
8   },  
9   "output": {  
10    "text": {  
11      "values": [  
12        "I can tell you about <? $foodtypes.join(', ') ?>",  
13        "What do you want to know ?"  
14      ],  
15      "selection_policy": "random"  
16    }  
17  }  
18 }
```

Script-editing interface

In the preceding context script, on the top left, you can see *then respond with*. That means you can script your subdialogs within the dialog flow. You can add a random function such as `selection_policy: random` so that the system changes phrases every time.



Your intents and entities can also be enriched by importing data. I recommend using big data map and retrieving techniques to boost the power of the dialog. The dialog will come close to human dialogs.

IBM Watson entity features offer a vast number of entity feature add-ons, as the following screenshot shows:

The screenshot shows the IBM Watson Conversation interface under the ChatBot / Build section. The Entities tab is selected. Under the System entities section, five entities are listed with their descriptions and status toggles:

- @sys-person BETA**: Extracts names from user input. Status: on
- @sys-location**: Extracts place names (country, state/province, city, town, etc.) from user input. Status: on
- @sys-time**: Extracts time mentions (at 10). Status: on
- @sys-date**: Extracts date mentions (Friday). Status: on
- @sys-currency**: Extracts currency values from user examples including the amount and the unit. Status: on

IBM Watson entity

The @Sys-person extracts names, the @sys-location system extracts place names (country, state/province, city, town, and so on), @sys-time extracts the time, @sys-date extracts the date, and @sys-currency extracts the currency and other features.

The quality of a chatbot will depend on:

- Not adding too many features, but just those required in some cases
- Adding all the necessary features when necessary, no matter how much time it takes

Adding services to a chatbot

IBM's Watson Conversation Service allows you to add all the services you want to your dialog flow, as shown in the following list:

- AlchemyLanguage
- AlchemyVision
- AlchemyData News
- Authorization
- Concept Insights
- Conversation
- Dialog
- Discovery
- Document Conversion
- Language Translator
- Natural Language Classifier
- Personality Insights
- Relationship Extraction
- Retrieve and Rank
- Speech to Text
- Text to Speech
- Tone Analyzer

Watson Conversation Services

IBM Watson provides a robust framework and a cloud service to build a solid, reliable chatbot.

With IBM Watson services, your imagination will be your only limit.



You can also implement a service of your own, such as a cognitive service.

A cognitive chatbot service

You now have a chatbot running on IBM Watson or another platform. The architecture of an efficient chatbot requires restricting the perimeter of the functions to a minimum and getting it to work as quickly as possible.

With a solid foundation, the chatbot can now expand its limitless potential in the areas required. Services such as these can be added to a personal assistant, for example.

The case study

A bus of English-speaking tourists is traveling through France. Somewhere in the countryside, on a small road, the bus breaks down. A tourist will use a personal assistant with inbuilt functions to communicate with a police chatbot.

Nobody speaks French. One of the tourists has a tablet and an internet connection. It also has a personal assistant on it. It could be Google Assistant, Siri, Bixby, Alexa, Cortana, or any other.

For our purpose, let's assume it's Google Assistant and say we built a backend with Google Translate, with the API described in Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies*.

The tourist types or says as described in that chapter: The coach broke down. Google Translate produces a sentence in French, confusing "coach" with a sports coach and not translating it as "bus." As explained before, Google will no doubt correct this error, but there are still thousands more to correct.

This is not a problem but a limitation that we, as a worldwide community of artificial intelligence, must expand by sharing ideas.

A cognitive dataset

A cognitive NLP chatbot must provide cognition. Humans think in more than words. Mostly humans combine words and images.

Furthermore, for a human, a chat generates mental images, sounds, odors, feelings, sensations, and much more.

The scenario is expanded to images for this case study:

- The tourist will chat with the police chatbot using the inbuilt functions of a personal assistant.
- The police bot will answer that it does not understand the statement. The tourist's personal assistant will trigger off a deeper conversation level.
- They will find a way to understand each other using images to solve the problem.

In this section, CRL-MM-Chabot .py will be described in a chronological way, following the sequence of the dialog.

The program starts with an explanation of the scenario and shows where this new cognitive function could be inserted in `Google_Translate_Customized.py` (see Chapter 8, *Revolutions Designed for Some Corporations and Disruptive Innovations for Small to Large Companies*).

Cognitive natural language processing

Conceptual representation learning meta-model (CRLMM) means describing concepts in words plus images, beyond using mathematical numerical vectors.

If you run `CRL-MM-Chabot.py`, the output comes as a detailed explanation of what the program is doing while it is running, as shown in the following code block. You can also view the video that comes with this chapter to get a feel of the program before running and viewing the following output:

```
In chapter 8, the Google_Translate_Customized.py
program found an error in Google Translate.
Google Translate confused a sports coach with a vehicle-coach.
In the near future, Google will probably correct this example
but here are thousands of other confusions.
On line 48 of Google_Translate_Customized.py, a deeper_translate function
was added to correct such errors.
This was only possible because Google provides
the necessary tools for the AI community to help make progress.
The translation was corrected so that the word bus replaced coach
and Google translated the sentence correctly
This program extends polysemy deeper understanding to concepts:
words and images

This model can be added to
deep_translate in Google_Translate_Customized at line 114 or
deployed in a cognitive chabot as follows...
Press ENTER to continue
```

The tourist now enters a sentence in the personal assistant (text or voice) that Google Translate does not understand (but it will in the near future), as displayed in the following snippet:

```
PERSON IN A BUS THAT BROKE DOWN TEXTING THE POLICE
A MESSAGE WITH A CHABOT TRANSLATOR:
My sports coach broke down(the sentence is in French but came out that
way)
```

The police bot answers and a sentiment analysis function detects a negative feeling, as shown in the following code snippet:

```
POLICE RECEIVING THE SMARTPHONE MESSAGE:  
I dont understand what you mean.  
The sentiment analysis estimation is: -0.3125
```

The following sentiment analysis function comes from TextBlob:

```
ext = "I dont understand what you mean."  
obj = TextBlob(text)  
sentiment = obj.sentiment.polarity  
print (text, "\n", " The sentiment analysis estimation is: ", sentiment)
```

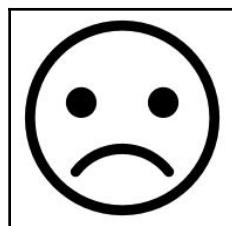
The sentiment has been analyzed through the `polarity` function. Negative values mean negative feelings, positive feelings come out as positive values, and 0 is neutral. In this case study, 0 will be OK.

The sentiment estimation for the policeman's sentence is -0.3, which is not good at all.

Our dialog flow (see the explanation in the first section of the chapter) does not wish to add confusions to the present confusion. Since the sentiment is negative, instead of adding more words that might or might not be correctly translated, an image is displayed. In this example, a frown is displayed, as implemented in the following code:

```
if(sentiment<0):  
    print("IMAGE DISPLAYED: FROWN OR OTHER POLYSEMY")  
    imgc=directory+'cchat/notok.jpg'  
    imgcc = load_img(imgc, target_size=(64, 64))  
    plt.imshow(imgcc)  
    plt.show(block=False)  
    time.sleep(5)  
    plt.close()
```

The frown will appear and disappear automatically on the tablet or smartphone. The following image is displayed by the program:



Output image

A /cchat (conceptual chat) directory was added to this CRLMM model. The CRLMM is beginning to use words + images to form concepts to express itself.

Words are sometimes confusing, especially in foreign-language environments on global social networks or personal assistants. Images will help clarify the conversation.



You can add a whole library of emoticons that will be activated by a sentiment analysis function such as TextBlob.

Activating an image + word cognitive chat

To resolve their misunderstanding the tourist and the police bot both have an inbuilt CRLMM function in their personal assistants and cloud chatbot platform.

On the policy side, a polysemy (several meanings to a word) function has automatically been triggered by the sentiment analysis value. The Polysemy function is activated and begins to clarify the situation, as shown in the following code:

```
This is sports coach.  
Polysemy Activated  
POLICE: Do you mean this coach?  
Enter yes or no(show an icon also to avoid language):
```

The program has already loaded its trained image model and compiled it, and the following prediction function is ready:

```
#____LOAD MODEL_____  
json_file = open(directory+'model/model.json', 'r')  
loaded_model_json = json_file.read()  
....  
loaded_model=model_from_json(loaded_model_json)  
#_____ load weights into new model  
loaded_model.load_weights(directory+"model/model.h5")  
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop',  
metrics=['accuracy'])
```

The images were trained by the same program described in the previous chapters. The training has been orientated towards defining images conceptually more than classifying them. The prediction function is also the same as described in earlier chapters.



Chapter 14, *Improve the Emotional Intelligence Deficiencies of Chatbots*, will detail this conceptual training approach for image-word concepts.

The police bot displays what was understood by its system. Using more words will add more confusion. This is why an image provides a clear message. The program will display the following image:



Output image providing clear message

The system displays the following message:

POLICE: Do you mean this coach?
Enter yes or no (show an icon also to avoid language) :

I recommend adding a yes or no icon.

The user will answer: **no**.

The conceptual (images + words) polysemy has been triggered by the mistake the system made. Another image is displayed as follows, with a question:



Output image with question

The tourist may or may not understand the policeman's question. The question may or may not be correctly translated. But providing a yes or no answer to an image simplifies the communication problem.

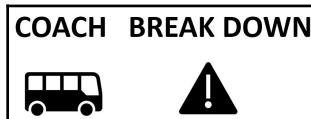
The user answers yes (or clicks on a yes icon). The tourist + personal assistant bot and the police bot are now beginning to understand each other.

Solving the problem

The chatbot has escalated into image + word chat mode. Confusing words have been replaced by an image and a yes/no clarified communication function. The system now thinks it understands. However, the bot has to make sure that it understands as shown in the following screenshot of the program's dialog:

```
POLICE:I understand. Your bus broke down.  
Bus+ brock down  
Is this what you mean?  
Enter yes or no:
```

It displays the following image of the situation (concept + words):



Output image (concept + words)

The tourist will recognize the bus and a sort of problem sign even if the words remain unclear.

The user answers yes. The following snippet runs a sentiment analysis and displays an image:

```
if(answer=="yes"):  
    print("THE SYSTEM WRITES A SENTENCE:", "\n")  
    text = "Image to text: Yes that is correct"  
    obj = TextBlob(text)  
    sentiment = obj.sentiment.polarity  
    print (text, " : ",sentiment)  
    if(sentiment>=0):  
        imgc=directory+'cchat/ok.jpg'
```

The system produces a scenario sentence (see IBM Watson's dialog flow), the `sentiment.polarity` produces a value and displays the following image.



Output image

The police bot and the tourist now understand each other.

The police bot produces an image and a text in case the message is confusing. The following snippet shows the police reply with an image to clarify the message:

```
print("\n", "POLICE : OK. We are already on our way.", "\n", "The bus  
emergency sensor sent us a message 5 minutes ago.", "\n", "Thanks for  
confirming", "\n")  
print("\n", "POLICE : We will be there at 15:30")  
imgc=directory+'cchat/ETA.jpg'  
imgcc = load_img(imgc, target_size=(128, 200))
```

The image confirms or replaces the text. The police bot informs the tourist that the estimated time of arrival is 15:30.

The police bot had received a sensor message from the bus. The bus had an inbuilt alerting system. But the emphatic cognitive chatbot was glad to help, just like a real-life police officer.

Implementation

Implementing a cognitive chatbot using CRLMM, as for any chatbot, needs to follow some well-organized steps:

- Identify the intents and entities
- Create the dialog scenarios
- Dialog controls—all that the user clicks and answers
- Make an emoticon images repository
- Make a conceptual images repository
- Train and test the model with all sorts of scenarios; especially look out for the ones that fail to improve the system

Summary

IBM Watson provides a complete set of tools on IBM Cloud to build a chatbot, add services to it, and customize it with your own cognitive programs.

A good chatbot fits the need. Thinking the architecture through before starting development avoids underfitting (not enough functionality) or overfitting (functions that will not be used) of the model.

Determining the right intent (what is expected of the chatbot), determining the entities to describe the intent (the subsets of phrases and words), and creating a proper dialog will take quite some time.

If necessary, adding services and specially customized machine learning functions will enhance the quality of the system. The use of CRLMM introduces concepts (images + words) into chatbots, bringing them closer to how humans communicate.

NLP parses and learns words. CRL adds a higher dimension to those words with images. These images clarify interpretation and mistranslation problems. To close the gap between machines and humans, AI must bring emotional intelligence into chatbot dialogs.

The next chapter explores emotional intelligence through a Boltzmann machine (recently used by Netflix) and principal component analysis applied to a CRLMM's image + word (concept) approach.

14

Improve the Emotional Intelligence Deficiencies of Chatbots

Recent disruptive AI technology provides all the tools to eliminate emotional deficiencies in artificial intelligence applications. **Conceptual representation learning meta-models (CRLMM)** can convert mathematical outputs into somewhat human mind models. Classical machine learning and deep learning programs mostly produce mathematical outputs. In previous chapters, CRLMM models produced enhanced conceptual representations, closing the gap between machine and human thinking.

The goal is clearly to build a mind, not an output, through several machine learning techniques: an RBM, sentiment analysis, CRLMM, RNN, LSTM, Word2Vec, and PCA. For each step, an application is available on a Windows or Linux platform, along with a video.

RBM^s were presented in a Netflix competition for prediction purposes, namely to predict user choices. The ultimate goal of this model is not to predict what a person will say or do but to make an approximate mental clone of a human mind—to build thinking and feeling machines.

Building a mind requires teamwork between several AI models. This chapter describes a profiler solution that will build a profile of a person named **X**.

Starting to build a mind is the beginning of what will be one of the major adventures of the 21st century.

The following topics will be covered in this chapter:

- **Restricted Boltzmann Machine (RBM)**
- CRLMM
- Sentiment analysis
- **Recurrent neural networks (RNN) and Long Short-Term Memory (LSTM)**
RNNs
- Word2Vec (word to vector) embedding
- **Principal Component Analysis (PCA)**
- Jacobian matrix

Technical requirements

Python 3.6x 64-bit from <https://www.python.org/>.

Package and modules:

```
import numpy
from __future__ import print_function
import numpy as np
from textblob import TextBlob
from PIL import Image
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import matplotlib.pyplot as plt
import time
import scipy.misc
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import keras
from keras.preprocessing.image import ImageDataGenerator
from keras.models import model_from_json

import tensorflow as tf
from tensorflow.contrib import rnn
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import collections
import math
import os
import random
from tempfile import gettempdir
import zipfile
```

```
import numpy as np
from six.moves import urllib
from six.moves import xrange # pylint: disable=redefined-builtin
from tensorflow.contrib.tensorboard.plugins import projector
import tensorflow as tf
```

Programs needed:

- RBM.py
- CNN_CONCEPT_STRATEGY.py
- LSTM.py with its Tensorboard_reader.py
- RNN.py
- Embedding.py and its Tensorboard_reader.py

Check out the following video to see the code in action:

<https://goo.gl/SKPUZA>

Building a mind

The following method represents a starting point to build a mind in a machine.

It is based on a cognitive NLP chatbot I successfully delivered many years ago. It was designed for language teaching and would teach a student, but more importantly, it would **learn** who that student was by **memorizing** the chats and storing words, images, and sounds. The bot would know more about the student after each session and ask questions about past conversations.

A random memory sentence was produced among many, such as:

Oh, I remember you. You like.....

Then the profile dataset of the person was used to add:

Oh, I remember you. You like + **football**

The voice was digitized (and a bit bumpy because of intonation problems), but it scared one of the first users enough to make the student stand up and go look to see if there was somebody behind the computer playing a prank.

I had to build everything from scratch, which was excruciating but incredibly exhilarating. This project was fun and very profitable for a while, but that was 25+ years ago! There was no internet and no need to replace people. So that was it. I got some media attention but had to move on.

I then used the CRLMM model I had been describing during the previous chapters to introduce deep human concepts in an advanced planning and scheduling system; this is not meant to be a real-time planning tool.

This proves that building a mind is possible and will now happen. Maybe you can take everything in this book and become the architect of a thinking machine, and sell it to a corporation for deep decision-making purposes or as an advanced cognitive NLP chatbot.

Outer space, way above the sky, is the limit and might lead you to build the technology for a smart rocket on its road to Mars. It's at the touch of your fingertips.

How to read this chapter

The goal is to show how to build a mind using several artificial intelligence techniques. Each technique will be briefly explained and then applied. At this point in the book, you know that all the network models encountered in machine learning contain inputs, activation functions, and outputs going through layers (visible and hidden). The flow is trained with an optimizer (loss function) and weights (with biases). There are many combinations of these models.

Each section contains a Python program you can run, explore, and view through a video. The video shows how a mind thinks in a 3D projection of thoughts and concepts. Image embedding was used to represent concepts (conceptual representation learning). Then PCA was applied.

The target scenario is for us to use machine learning tools and bots to gather data on a person named X.

The profiling scenario

The goal is to learn who X is and build a mind for X through a CRLMM approach added to other techniques.



Building a machine mind requires using all the machine learning techniques we can put together. It would take several months to produce a thinking machine for decision-making or chatbot purposes.

This x step scenario will provide enough data to start building a mind:

- An RBM that will not try to predict what a user will do but will profile person X based on their Netflix ratings in an advanced approach
- A sentiment analysis function to gather information through X's social networks
- A CRLMM function to gather images and begin to fit them with the words as X would do
- Using an RNN to generate sentences X could say in a data augmentation approach
- Applying word embedding (word2vec) to the text to gather information on how X perceives words, not their general statistical meaning
- Representing some of X's professional environment words with word2vec in TensorFlow projector with PCA, to obtain a personal mental representation of how X sees the world (not the statistical representation)



The goal is not to classify X or predict what X would do, but to understand X by representing X's mind.

Restricted Boltzmann Machines

RBM^s are random and undirected graph models generally built with a visible and a hidden layer. They were used in a Netflix competition to predict future user behaviors. The goal here is **not** to predict what X will do but **find out who X is** and store the data in X's profile-structured mind-dataset.

The input data represents the features to be trained to learn about person X. Each column represents a feature of X's potential personality and tastes. The following code (and this section) is in RBM.py:

```
F = ["love", "happiness", "family", "horizons", "action", "violence"]
```

Each of the following lines is a movie X watched containing those six features; for them, X gave a five-star rating:

```
r = RBM(num_visible = 6, num_hidden = 2)
training_data = np.array([[1,1,0,0,1,1],
                         [1,1,0,1,1,0],
                         [1,1,1,0,0,1],
                         [1,1,0,1,1,0],
                         [1,1,0,0,1,0],
                         [1,1,1,0,1,0]])
```

The goal of this RBM is to find out more about X's personality through the output of the RBM.

The connections between visible and hidden units

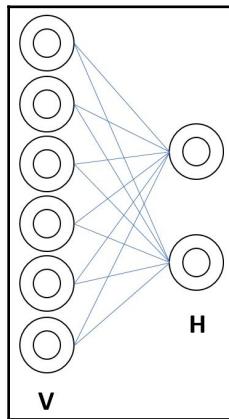
The RBM model used contains two layers:

- A visible layer
- A hidden layer

Many types of RBMs exist, but generally, they contain these properties:

- There is no connection between the visible units
- There is no connection between the hidden units

- The visible and hidden layers are connected by a weight matrix and a bias vector, as shown in the following diagram:



The connections between visible and hidden units

The network contains six visible and two hidden units, producing a weight matrix of 2×6 values. Note that there are no connections between the visible or hidden units. The following Python code initializes the model with the units:

```
self.num_hidden = num_hidden
self.num_visible = num_visible
```

The weight matrix is initialized with random weights between the following:

- $-\sqrt{6. / (\text{num_hidden} + \text{visible})}$
- $+\sqrt{6. / (\text{num_hidden} + \text{visible})}$

It is implemented with the following random states and values:

```
np_rng = np.random.RandomState(1234)
self.weights = np.asarray(np_rng.uniform(
    low=-0.1 * np.sqrt(6. / (num_hidden + num_visible)),
    high=0.1 * np.sqrt(6. / (num_hidden + num_visible)),
    size=(num_visible, num_hidden)))
```

The bias will now be inserted in the first row and the first column in the following code:

```
self.weights = np.insert(self.weights, 0, 0, axis = 0)
self.weights = np.insert(self.weights, 0, 0, axis = 1)
```

The goal of this model will be to observe the behavior of the weights. Observing the weights will determine how to interpret the result in this model based on calculations between the visible and hidden units. The following output displays the weight values:

```
[ [ 0.91393138 -0.06594172 -1.1465728 ]
[ 3.01088157 1.71400554 0.57620638]
[ 2.9878015 1.73764972 0.58420333]
[ 0.96733669 0.09742497 -3.26198615]
[-1.09339128 -1.21252634 2.19432393]
[ 0.19740106 0.30175338 2.59991769]
[ 0.99232358 -0.04781768 -3.00195143] ]
```

The first row and column are the biases. Only the weights will be analyzed for the *profiling* functions. The weights and biases are now in place.

Energy-based models

An RBM is an **energy-based model (EBM)**. An EBM uses probabilistic distributions defined by an energy function.

For a given x , $E(x)$ or $E(v,h)$ is one way of expressing the energy function of an RBM in a simplified, conceptual line. It is:

$$E(v, h) = -bv - hWv$$

This **energy function drives** the model and will be used to direct weight and bias optimization:

- The v (visible) and h (hidden) units are connected by W (the weights)
- b is an offset of the units that acts as a bias

Then the energy will be measured with a probabilistic function p :

$$p(x) = \frac{e^{-E(x)}}{Z}$$

Z is a **partition function** for making sure that the sum of the probabilities of each x input does not exceed 1:

$$\sum_x p(x) = 1$$

The partition function is the sum of all the individual probabilities of each x :

$$Z = \sum_x e^{-E(x)}$$

With each $p(x)$ divided by Z (the sum of all the probabilities), we will reach 1. The whole system is based on the energy between the visible and hidden units until the loss is optimized, as shown in the following RBM.py code:

```
error = np.sum((data - neg_visible_probs) ** 2)
if self.debug_print:
    print("Epoch %s: error is %s" % (epoch, error))
```

Gibbs random sampling

An RBM increases in efficiency when random variable sampling is applied during the training process. Gibbs sampling generates a Markov chain of samples that takes the nearby samples into account as well.

Random sampling makes the system more objective. Hidden units remain latent (this is a two-layer system), which confers its dream property.

Running the epochs and analyzing the results

Once the RBM has optimized the weight-bias matrix for n epochs, the matrix will provide the following information for the profiler system of person X:

```
[[ 0.91393138 -0.06594172 -1.1465728 ]
 [ 3.01088157 1.71400554 0.57620638]
 [ 2.9878015 1.73764972 0.58420333]
 [ 0.96733669 0.09742497 -3.26198615]
 [-1.09339128 -1.21252634 2.19432393]
 [ 0.19740106 0.30175338 2.59991769]
 [ 0.99232358 -0.04781768 -3.00195143]]
```

When RBM.py reaches this point, it asks you if you want to profile X, as shown in the following output:

```
Press ENTER if you agree to start learning about X
```



I strongly recommend that only bots access this type of data and that the provider protects the data as much as possible. For users to accept these models, they must first trust them.

If you accept, it will train the input and display the features added to X's profile.

The weights of the features have been trained for person X. The first line is the bias and examines column 2 and 3. The following six lines are the weights of X's features:

```
Weights:  
[[ 0.913269 -0.06843517 -1.13654324]  
[ 3.00969897 1.70999493 0.58441134]  
[ 2.98644016 1.73355337 0.59234319]  
[ 0.953465 0.08329804 -3.26016158]  
[-1.10051951 -1.2227973 2.21361701]  
[ 0.20618461 0.30940653 2.59980058]  
[ 0.98040128 -0.06023325 -3.00127746] ]
```

The weights (in bold) are lines 2 to 6 and columns 2 to 3. The first line and first column are the biases.

The weight matrix will provide a profile of X by summing the weight lines of the feature, as shown in the following code:

```
for w in range(7):  
    if(w>0):  
        W=print(F[w-1],":",r.weights[w,1]+r.weights[w,2])
```

The features are now **labeled**, as displayed in this output:

```
love : 2.25265339223  
happiness : 2.28398311347  
family : -3.16621250031  
horizons : 0.946830830963  
action : 2.88757989766  
violence : -3.05188501936
```



A value >0 is positive, close to 0 slightly positive.
A value <0 is negative, close to 0 slightly negative.

We can see that beyond standard movie classifications, X likes horizons somewhat, does not like violence, and likes action. X finds happiness and love important but not family at this point.

The RBM has provided a personal profile of X—not a prediction but getting ready for a suggestion through a chatbot or just building X's machine mind-dataset.

RBM.py will now begin sentiment analysis.

Sentiment analysis

RMB.py leaves the realm of RBMs and enters a tone analyzer phase using sentiment analysis to parse X's social networks.

The following code triggers the bot to ask you once again whether you agree (the mind-dataset-building bot) to parse X's social networks:

```
Press ENTER if you agree to learn more about X.  
The AI program will now enter social networks  
to scan X's social media profile.  
  
First words(Tweets, posts, messages on Facebook and much more) will be  
analyzed then images  
  
Press ENTER if you agree to scan X's social networking...
```

Parsing the datasets

For this to work, a bot has read through X's data. A bot can use big data map and retrieve functions that Google, Amazon, Facebook, LinkedIn, and other social networks have been developing.

The sentiment analysis function is a TextBlob module. Entering the text and `text+.sentiment` will return the following sentiment analysis:

```
myview=TextBlob("I hate movie 1. It was too violent ")  
print(myview,":","\n",myview.sentiment,"\n")
```

The output will contain two values:

- **Polarity**



A value >0 is positive, close to 0 slightly positive.
A value <0 is negative, close to 0 slightly negative.

- **Subjectivity:** The higher the value, the more X's subjectivity influences the result

For mind-dataset building purposes, a high level of subjectivity provides inside information on X's feelings, which increases the quality of the dataset.

At the same time, the total dialog is loaded in a TextBlob for each sentence analyzed as shown in the following code:

```
myview=TextBlob("I like autumn. It reminds me of some sad music")
print(myview,":","\n",myview.sentiment,"\n")
dialog=dialog+myview
```

myview contains the sentence to be analyzed. dialog contains the complete dialog. At the end of the analysis, dialog is parsed for key noun phrases, as shown in this code:

```
#Parse noun phrases
print("Parse noun phrases to find potential key words:")
print(dialog.noun_phrases)
```

The result will produce sentiment analysis for each sentence and the noun phrases at the end. The output provides useful information to add to X's mind-dataset. The following sentences were scanned on X's social networks:

```
Press ENTER to Continue
A value $>0$  is positive, close to 0 slightly positive
A value $<0$  is negative, close to 0 slightly negative

I hate movie 1. It was too violent :
Sentiment(polarity=-0.8, subjectivity=0.95)

I like autumn. It reminds me of some sad music :
Sentiment(polarity=-0.5, subjectivity=1.0)

The love story was cool too. A bit mushy but cool :
Sentiment(polarity=0.3999999999999997, subjectivity=0.6333333333333333)

I would like to get out of here and see other horizons :
Sentiment(polarity=-0.125, subjectivity=0.375)
```

Parse noun phrases to find potential key words:
['sad music', 'bit mushy']

RBM.py now asks you, the profiler bot, whether you want to complete X's profile with images gathered on X's social network pages with the following authorization:

Press ENTER if you agree to complete X's profiling dataset with some images

Conceptual representation learning meta-models

The CRLMM phase is there to find images that constitute a mental representation for X.



Humans mostly think in words+images and feel with words+images. Sounds, sensations, and odors are critical as well. That parameter can also be explored.

Profiling with images

Humans communicate widely through images, symbols, and pictures. They also have a large number of mental image datasets—images that have been transformed and linked to words.

The goal of the bot's conversation now is to confirm the profile of X and suggest a movie to X, just for fun. An image was stored by X, and the name of a movie comes up called *Lost?* with a dark picture. It is supposed to appeal to the emotions of somebody who yearns for happiness and love and is curious to see whether something will be found.

The following picture illustrates the darkness of being lost, with a forest as a symbol and light through the trees (hope?):



Darkness of being lost picture

Now the streaming website chatbot will detect that a viewer has been scrolling for over 5 minutes without making a choice, for example. It would like to make a suggestion.

The bot starts with a random choice of sentences in the following structured class and adds keywords:

```
print("Hi, I am your personal assistant to find a movie you like")
print("I found a movie named Lost. The hero has to fight(action) through
life")
print("after losing a loved one (love) and is searching for
happiness(happiness) again...")..
Do you want to watch the movie?
```

X is curious about this movie. Will something be found? Happiness? Something else? X answers, *I sure do.*

The program manages the following dialog sequence. If you use the program, enter *I sure do!* when asked if you want to be person X:

```
mytext=input("Do you want to watch the movie? ")
#enter I sure do!
myview=TextBlob(mytext)
print(myview,":",myview.sentiment)
```

It picks up `I sure do` and obtains the following sentiment analysis, which it will now add to X's mind-dataset:

```
I sure do : Sentiment(polarity=0.5, subjectivity=0.888888888888888)
```

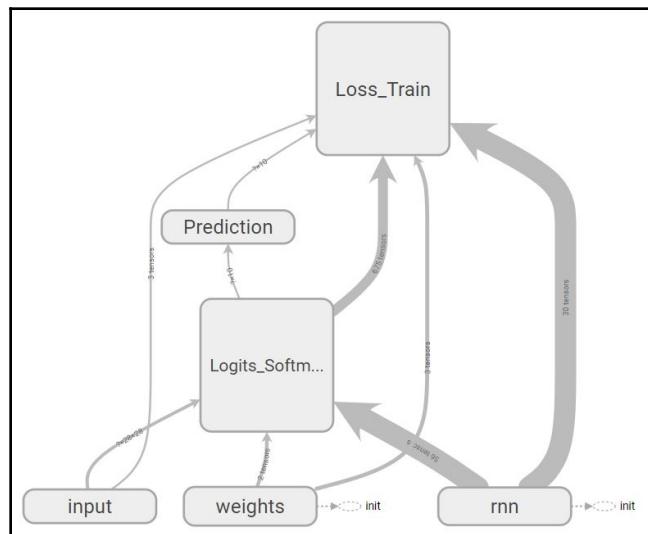
Note that the goal here was first to profile X and build a mind-dataset of X's features, not movie categories. Then only suggest a movie, not based on its category, but on more intrinsic features.

The next phase will be to increase the size of the mind-dataset with an RNN.

RNN for data augmentation

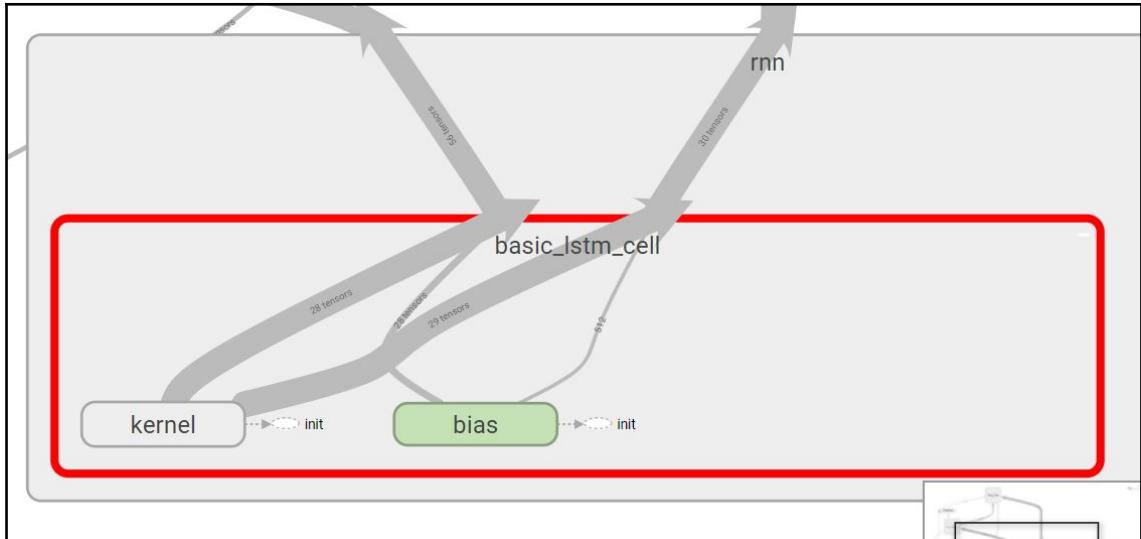
An RNN models sequences, in this case, words. It analyzes anything in a sequence, including images. To speed the mind-dataset process up, data augmentation can be applied here exactly as it is to images in other models. In this case, an RNN will be applied to words.

A first look at its graph data flow structure shows that an RNN is a neural network like the others previously explored. The following graphs were obtained by first running `LSTM.py` and then `Tensorboard_reader.py`:



Data flow structure

The y inputs (test data) go to the loss function (`Loss_train`). The x inputs (training data) will be transformed through weights and biases into logits with a softmax function. A zoom into the RNN area of the graph shows the following `basic_lstm_cell`:



basic_lstm cell—RNN area of the graph

What makes an RNN special is to be found in the LSTM cell.

RNNs and LSTMs

An LSTM is an RNN that best shows how RNNs work.

An RNN contains functions that take the output of a layer and feed it back to the input in sequences simulating time. This feedback process takes information in a sequence. For example:

*The->movie->was->**interesting**->but->*I*->*didn't*->*like*->*It**

An RNN will unroll a stack of words into a sequence and parse a window of words to the right and to the left. For example, in this sentence, an RNN can start with **interesting** (bold) and then read the words on the right and left (italics). These are some of the hyperparameters of the RNN.

This sequence aspect opens the door to sequence prediction. Instead of recognizing a whole pattern of data at the same time, it is recognizing the sequence of data, as in this example.

A network with no RNN will recognize the following vector as a week, a pattern just like any other one:

Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday

An RNN will explore the same data in a sequence by **unrolling streams of data**.

Monday->Tuesday-Wednesday->Thursday->Friday->Saturday-Sunday

The main difference lies in the fact that once trained, the network will predict the following: if Wednesday is the input, Thursday could be one of the outputs. This is shown in the next section.

RNN, LSTM, and vanishing gradients

To simulate sequences and memory, an RNN and an LSTM will use backpropagation algorithms.

An RNN often has problems of gradients when calculating them over deeper and deeper layers in the network. Sometimes, it vanishes (too close to 0) due to the sequence property, just like us when a memory sequence becomes too long.

The backpropagation (just like us in a sequence) becomes less efficient. There are many backpropagation algorithms such as vanilla backpropagation, which is commonly used. This algorithm performs efficient backpropagation because it updates the weights after every training pattern.

One way to force the gradient not to vanish is to use a ReLU activation function, $f(x)=\max(0,x)$, forcing values on the model so that it will not get stuck.

Another way is to use an LSTM cell containing a forget gate between the input and the output cells, a bit like us when we get stuck in a memory sequence and we say "whatever" and move on.



To explore this type of network deeper, `LSTM.py` runs on a MNIST example. `Tensorboard_reader.py` will display the data flow graph structure.

The LSTM cell will act as a memory gate with 0 and 1 values, for example. This cell will forget some information to have a fresh view of the information it has unrolled into a sequence.

The key idea of an RNN is that it unrolls information into sequences.

Prediction as data augmentation

`RNN.py`, a Vanilla RNN model, shows how person X's profile can be enhanced in three steps.

Step1 – providing an input file

`input.txt` contains texts picked up in X's dialogues with the chatbot, social networks, messaging, and more, as shown in the following output:

```
"The action movie was boring. The comedy was better. Next time, I'll watch  
an action comedy with funny scenes...."
```

The ultimate goal is to automatically collect the largest file possible with thousands of words to have a representative dataset.

Step 2 – running an RNN

The RNN, based on a vanilla algorithm, starts by reading the stream of characters. It first opens `input.txt`, the reference text, as follows:

```
# data I/O  
data = open('input.txt', 'r').read() # should be simple plain text file  
chars = list(set(data))  
data_size, vocab_size = len(data), len(chars)  
print ('data has %d characters, %d unique.' % (data_size, vocab_size))  
char_to_ix = { ch:i for i,ch in enumerate(chars) }  
ix_to_char = { i:ch for i,ch in enumerate(chars) }
```

The RNN has hidden layers connected to the visible input and a learning rate, `learning_rate`, as shown in the following snippet:

```
# graph structure
hidden_size = 100 # hidden layer size
learning_rate = 1e-1
```

Then the specific parameter of an RNN is added—the length of the sequence, to unroll:

```
seq_length = 25 # length of units to unroll
```

Next, the weights are initialized with random functions. The program then runs forward passes to parse the elements of the sequence and backward to compute gradients.

Step 3 – producing data augmentation

The program will now generate text that can be added to X's text profile dataset. Data augmentation by text generation will:

- Help understand X's profile better
- Be used in chatbot conversations when this technology improves

This approach takes time and resources, and the results are not always efficient. However, research and development should continue by combining text generators with concept representation learning and more.

An intermediate result at iteration 60200 provided the following output:

```
iteration 60200, loss: 0.145638
-----
er autumn because the leaves are brown, yellow, orange, red and all sorts
of colors. Summer is too hot, winhes comeny lard, yellories rore. I like
some but not too much. Love is really something great
```

I like some but not too much contains some interesting emotional information: some but not too much.



To explore this type of network deeper, consult `rnn.py` and the video of this chapter.

Word embedding

The mind-dataset of X has begun to pile up a lot of data. It's time to introduce a word to vector model to determine how X sees words.

Building a mind-dataset is **not** like a dictionary that provides a standard definition for everybody. In this project, we want to target person X's personality.

This approach is about profiling X to understand how X perceives a word. For example, a dictionary might depict the season spring as something flowery. But X does not like spring as we have seen. X prefers autumn and its mild sadness.

So in X's dictionary, the definition of autumn is closer to satisfying than summer. This type of bot takes personal views into account to build a personal profiled mind-dataset. On the contrary, a standard dictionary would not suggest a preference between two seasons.

Word2vec will be used for customized phrases.



The Word2vec model

A **vector space model (VSM)** takes words and transforms them into a vector space. In this continuous space, similar words are represented in points near each other. This vectorial mapping process is called **embedding**.

Word2vec takes raw text, finds the words nearest to each other, and maps them into vectors. The system will take words, build them into a dictionary, and then input indexes.

A **skip-gram model** is applied to the sequences with a **skip window**. In that skip window, a number of words are taken into consideration on the left and the right. Then the model **skips**, and it will reuse and input several times to produce a label as implemented in `embedding.py`. The embedding size, skip window, and the number of skips are initialized and described in the following snippet:

```
embedding_size = 128 # Dimension of the embedding vector.  
skip_window = 1 # How many words to consider left and right.  
num_skips = 2 # How many times to re-use an input to generate a label.
```

Once the whole process is over, the model will have produced a number of vector features of the dataset with groups of nearest words to each other.

For our profile, Embedding.py analyzes a text written by X on artificial intelligence. These are texts written after reading this book. We will know more about X in a workplace.



The following paragraphs describe the key functions of Embedding.py.
For more info, view the complete source code on GitHub.

Embedding.py starts by reading a file called concept.txt, which is in a ZIP file and contains a text that X wrote after reading this book, as shown in the following code:

```
def read_data(filename):
    """Extract the first file enclosed in a zip file as a list of concepts."""
    with zipfile.ZipFile(filename) as f:
        data = tf.compat.as_str(f.read(f.namelist()[0])).split()
    return data
```

The words are stored as labels in the label filename labels.tsv, as shown in the following code snippet:

```
file = open('log/labels.tsv', 'w+')
for i in range(maxvoc+1):
    print(i,vocabulary[i])
    file.write(str(i) + '\t' + vocabulary[i] + '\n')
file.close()
```

A dataset is built in real time in the following code:

```
def build_dataset(words, n_words):
    """Process raw inputs into a dataset."""
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(n_words - 1))
    dictionary = dict()
    ...
    UNK represents unknown words that are generally simply skipped.
```

Then, training batches are generated by the following generate_batch function:

```
def generate_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1 # [ skip_window target skip_window ]
    buffer = collections.deque(maxlen=span)
```

```

if data_index + span > len(data):
    data_index = 0
    buffer.extend(data[data_index:data_index + span])
    data_index += span
    for i in range(batch_size // num_skips):
        ...

```

The skip-gram model has been prepared. A TensorFlow data flow graph will manage the operations. First, come the following inputs:

```

embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
                                           embedding_size], -1.0, 1.0))
embed = tf.nn.embedding_lookup(embeddings, train_inputs)

```

Then the following weights and biases along with the loss function are implemented:

```

# Construct the variables for the NCE loss
nce_weights = tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
                        stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
loss = tf.reduce_mean(
    tf.nn.nce_loss(weights=nce_weights,
                   biases=nce_biases,
                   labels=train_labels,
                   inputs=embed,
                   num_sampled=num_sampled,
                   num_classes=vocabulary_size))

```

The word2vec count model will take the words and build a count matrix, thus converting words into vectors so that the deep learning algorithm can process the data.

Given the `concept.txt` data file, a part of that matrix could look something like the following table:

	Transfer	Learning	Model	Number
Transfer		15		
Learning			10	5
Model				
Number			1	

The words have been transformed into numbers (vectors) so that a deep learning program can process them and find the words nearest to each other.

The output represents the features of the word with properties expressed in numbers, which in turn can be viewed as words through the stored labels (`labels.tsv`).

The output of the `concept.txt` input file shows the nearest words to each keyword. For example, for `such`, the system displays the following output:

```
Nearest to such: "class, model, way, Adding, been, and, face, business,
```

It does not look so exciting at first glance. However, if more text is added and deeper calculations are allowed, it will make progress.

To obtain a 3D view of the result with PCA (see next section), the information for TensorBoard projector is initialized, as shown in the following snippet:

```
print('Initialized')
config = projector.ProjectorConfig()
summary_writer = tf.summary.FileWriter("log/")
embedding = config.embeddings.add()
```

TensorFlow Projector is now ready to be launched.

Principal component analysis

General dictionary or encyclopedia representations of words, images, and concepts satisfy many needs. The probabilistic machine learning approach is applied very efficiently to marketing by Facebook, Amazon, Google, Microsoft, IBM, and many other corporations.

Probabilistic machine learning training remains efficient when targeting apparels, food, books, music, travel, cars, and other market consumer segments.

However, humans are not just consumers; they are human beings. When they contact websites or call centers, standard answers or stereotyped emotional tone analysis approaches can depend on one's nerves. When humans are in contact with doctors, lawyers, and other professional services, a touch of humanity is necessary if major personal crises occur.

The PCA phase is there to build a mental representation of X's profile either to communicate with X or use X's mind as a powerful, *mind-full* chatbot or decision maker.

The input data for this PCA is the output of `Embedding.py` described in the previous section. TensorFlow stored the model, data and labels in its `log/` directory.

TensorBoard projector has an inbuilt PCA loader.

Intuitive explanation

PCA takes data and represents it at a higher level.

For example, imagine you are in your bedroom. You have some books, magazines, and music (maybe on your smartphone) around the room. If you consider your room as a 3D Cartesian coordinate system, the objects in your room are all in specific x , y , z coordinates.

For experimentation purposes, take your favorite objects and put them on your bed. Put the objects you like the most near each other and your second choices a bit further away. If you imagine your bed as a 2D Cartesian space, you have just made your objects change dimensions.

They are not in their usual place anymore; they are on your bed and at specific coordinates depending on your tastes.

That is the philosophy of PCA. If the number of data points in the dataset is very large, the PCA of a mental dataset of one person will always be different from the PCA representation of another person like DNA.



That is what CRLMM is about as applied to a person's mental representation. Each person is different, and each person deserves a customized chatbot or bot treatment.

Mathematical explanation

The main steps for calculating PCA are important for understanding how to go from the intuitive approach to how TensorBoard projector represents datasets using PCA.

Variance

Step 1: Calculate the mean of the array `data1`.

You can check this with the `math.py`, as shown in the following function:

```
data1 = [1, 2, 3, 4]
M1=statistics.mean(data1)
print("Mean data1",M1)
```

The answer is 2.5. The mean is not the median (the middle value of an array).

Step 2: Calculate the mean of array `data2`.

The mean calculation is done with the following standard function:

```
data2 = [1, 2, 3, 5]
M2=statistics.mean(data2)
print("Mean data2",M2)
```

The answer is:

$$\bar{X} = 2.75$$

The bar above the X signifies that it is a mean.

Step 3: Calculate the variance using the following equation:

$$var = \frac{\sum_{x=1}^{x=n} (X - \bar{X})^2}{n}$$

Now NumPy will calculate the variance with the absolute value of each x minus the mean, sum them up, and divide the sum by n as shown in the following snippet.

```
#var = mean(abs(x - x.mean())**2).
print("Variance 1", np.var(data1))
print("Variance 2", np.var(data2))
```

Some variances are calculated with $n-1$ depending on the population of the dataset.

The result of the program for variances is as displayed in the following output:

```
Mean data1 2.5
Mean data2 2.75
Variance 1 1.25
Variance 2 2.1875
```

We can already see that `data2` varies a lot more than `data1`. Do they fit together? Are their variances close or not? Do they vary in the same way?

Covariance

Our goal in this section is to find out whether two words, for example, will often be found together or close to each other, taking the output of the embedding program into account.

Covariance will tell us whether these datasets vary together or not. The equation follows the same philosophy as variance, but now both variances are joined to see whether they belong together or not:

$$\text{cov}(X, Y) = \frac{\sum_{x=1}^{x=n} (X - \bar{X})(Y - \bar{Y})}{n}$$

As with the variance, the denominator can be $n-1$ depending on your model. Also, in this equation, the numerator is expanded to visualize the co-part of covariance, as implemented in the following array in `math.py`:

```
x=np.array([[1, 2, 3, 4],  
           [1, 2, 3, 5]])  
print(np.cov(x))
```

NumPy's output is a covariance matrix **a**:

```
[[1.666666672.16666667]  
 [2.166666672.91666667]]
```

If you increase some of the values of the dataset, it will increase the value of the parts of the matrix. If you decrease some of the values of the dataset, the elements of the covariance matrix will decrease.

Looking at some of the elements of the matrix increase or decrease that way takes time and observation. What if we could find one value or two values that would give us that information?

Eigenvalues and eigenvectors

To make sense out of the covariance of the covariance matrix, the eigenvector will point to the direction in which the covariances are going. The eigenvalues will express the magnitude or importance of a given feature.

To sum it up, an eigenvector will provide the direction and the eigenvalue the importance for the covariance matrix \mathbf{a} . With those results, we will be able to represent the PCA with TensorBoard projector in a multi-dimensional space.

Let \mathbf{w} be an eigenvalue(s) of \mathbf{a} . An eigenvalue(s) must satisfy the following equation:

$$\text{dot}(\mathbf{a}, \mathbf{v}) = \mathbf{w} * \mathbf{v}$$

There must exist a vector \mathbf{v} for which $\text{dot}(\mathbf{a}, \mathbf{v})$ is the same as $\mathbf{w}^* \mathbf{v}$:

NumPy will do the math through the following function:

```
from numpy import linalg as LA
print ("eigenvalues", np.linalg.eigvals(a))
```

The eigenvalues are (in ascending order) displayed in the following output:

```
Eigenvalues [ 0.03665681  4.54667652]
```

Now we need the eigenvectors to see in which direction these values should be applied. NumPy provides a function to calculate both the eigenvalues and eigenvectors together. That is because eigenvectors are calculated using the eigenvalues of a matrix as shown in this code snippet:

```
from numpy import linalg as LA
w, v = LA.eigh(a)
print("eigenvalue(s)",w)
print("eigenvector(s)",v)
```

The output of the program is:

```
eigenvalue(s) [ 0.03665681  4.54667652]
eigenvector(s) [[-0.79911221  0.6011819 ]
 [ 0.6011819  0.79911221]]
```

Eigenvalues come in a 1D array with the eigenvalues of \mathbf{a} .

Eigenvectors come in a 2D square array with the corresponding (to each eigenvalue) in columns.

Creating the feature vector

The remaining step is to sort the eigenvalues from the highest to the lowest value. The highest eigenvalue will provide the principal component (most important). The eigenvector that goes with it will be its feature vector. You can choose to ignore the lowest values or features. In the dataset, there will be hundreds and often thousands of features to represent. Now we have the feature vector:

$$\text{Feature Vector} = \text{FV} = \{\text{eigenvector}_1, \text{eigenvector}_2, \dots, \text{n}\}$$

n means that there could be many more features to transform into a PCA feature vector.

Deriving the dataset

The last step is to transpose the feature vector and original dataset and multiply the row feature vector by row data:

$$\text{Data that will be displayed} = \text{row of feature vector} * \text{row of data}$$

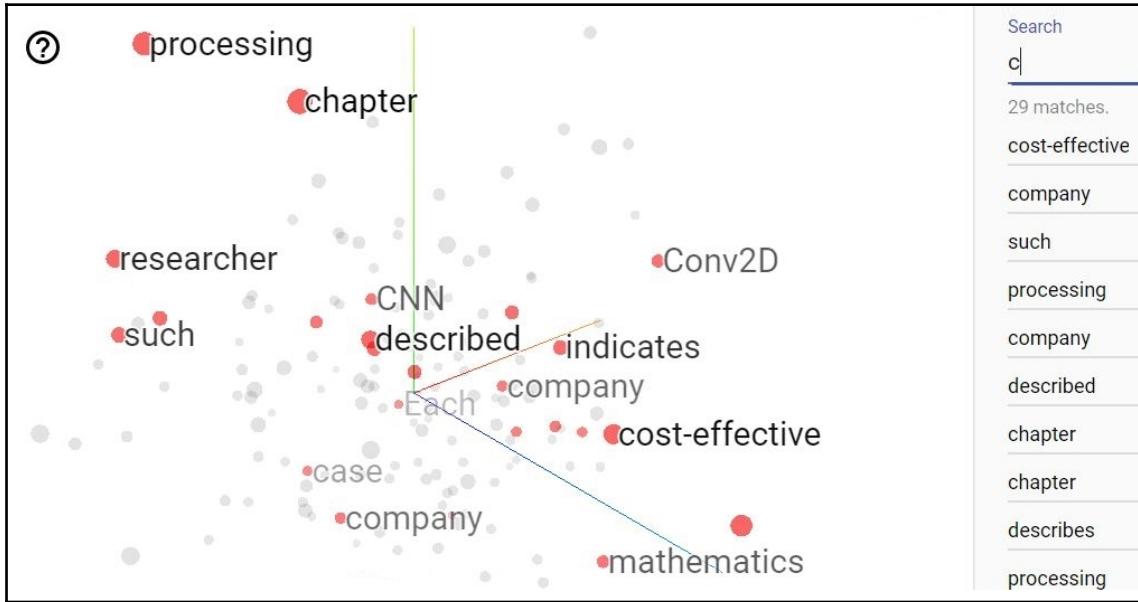
Summing it up

The highest value of eigenvalues is the principal component. The eigenvector will determine in which direction the data points will be oriented when multiplied by that vector.

TensorBoard Projector

TensorBoard projector loads the metadata from the output of the `Embedding.py` located in the `log` directory. First, run `Embedding.py`. Then run `Tensorboard.py`. For this example, click on **Load data** and load the `labels.tsv` file from the `log` directory generated by `Embedding.py`.

Next, TensorBoard runs a PCA calculation (the option on the left in the TensorBoard window) as explained previously. It then displays a 3D view, as shown in the following screenshot:



You can now select words on the right-hand side of the window, move the projection around in 3D, zoom, and explore the workplace mind of X.

Using Jacobian matrices

Using a matrix of partial derivatives is also possible. It shows the local trend of the data being explored. A Jacobian matrix contains partial derivatives.

A Jacobian matrix that contains the partial derivatives of X's mind functions produced between two periods of time can prove useful when minute trends need to be analyzed. The philosophy remains the same as for measuring covariances. The goal is to see the trends in the variables of X's mind.

Summary

Efficient machine learning programs produce good results but do not build a mind. To produce complex AI solutions, no single technique (MDP, CNN, RNN, or any other) will suffice. An efficient artificial intelligence solution to a complex problem will necessarily be an innovative combination of various techniques. This chapter showed how to start building one using an RBM, a tone analyzer (sentiment analysis), a CRLMM, an RNN, a word2vec program, and PCA.

X's mind-dataset has been enriched with the beginning of an actual memory with word and feeling associations as image and word associations. This brings X's mind-dataset profile beyond dictionary definitions, predictive statistics, and linear dimensional personality definitions.

Artificial intelligence will go beyond the present trend of mathematical probabilistic approaches into the realms of our minds.

Great progress has been made by major corporations distributing machine learning cloud platforms (Google, Amazon, IBM, Microsoft, and many more). CRLMM will become a necessity to solve complex problems that mathematical probabilities cannot, such as empathetic chatbots or deep thinking machines.

15

Building Deep Learning Environments

Welcome to the applied AI deep-learning team, and to our first project—*Building a Common Deep Learning Environment!* We're excited about the projects we've assembled in this book. The foundation of a common working environment will help us work together and learn very cool and powerful **deep learning (DL)** technologies, such as **computer vision (CV)** and **natural language processing (NLP)**, that you will be able to use in your professional career as a data scientist.

The following topics will be covered in this chapter:

- Components in building a common DL environment
- Setting up a local DL environment
- Setting up a DL environment in the cloud
- Using the cloud for deployment for DL applications
- Automating the setup process to reduce errors and get started quickly

Building a common DL environment

Our main goal to achieve by the end of this chapter is to standardize the toolsets to work together and achieve consistently accurate results.

In the process of building applications using DL algorithms that can also scale for production, it's very important to have the right kind of setup, whether local or on the cloud, to make things work end to end. So, in this chapter, we will learn how to set up a DL environment that we will be using to run all the experiments and finally take the AI models into production.



First, we will discuss the major components required to code, build, and deploy the DL models, then various ways to do this, and finally, look at a few code snippets that will help to automate the whole process.

The following is the list of required components that we need to build DL applications:

- Ubuntu 16.04 or greater
- Anaconda Package
- Python 2.x/3.x
- TensorFlow/Keras DL packages
- CUDA for GPU support
- Gunicorn for deployment at scale

Get focused and into the code!

We'll start by setting up your local DL environment. Much of the work that you'll do can be done on local machines. But with large datasets and complex model architectures, processing time slows down dramatically. This is why we are also setting up a DL environment in the cloud, because the processing time for these complex and repetitive calculations just becomes too long to be able to efficiently get things done otherwise.

We will work straight through the preceding list, and by the end (and with the help of a bit of automated script), you'll have everything set up!

DL environment setup locally

Throughout this book, we will be using Ubuntu OS to run all the experiments, because there is great community support for Linux and mostly any DL application can be set up easily on Linux. For any assistance on installation and setup related to Ubuntu, please refer to the tutorials at <https://tutorials.ubuntu.com/>. On top of that, this book will use the Anaconda package with Python 2.7+ to write our code, train, and test. Anaconda comes with a huge list of pre-installed Python packages, such as `numpy`, `pandas`, `sklearn`, and so on, which are commonly used in all kinds of data science projects.



Why do we need Anaconda? Can't we use Vanilla Python?

Anaconda is a generic bundle that contains iPython Notebook, editor, and lots of Python libraries preinstalled, which saves a lot of time on setting up everything. With Anaconda, we can quickly get started on solving the data science problem, instead of configuring the environment. But, yes, you can use the default Python—it's totally the reader's choice, and we will learn at the end of this chapter how to configure `python env` using script.

Downloading and installing Anaconda

Anaconda is a very popular data science platform for people using Python to build machine learning and DL models, and deployable applications. The Anaconda marketing team put it best on their *What is Anaconda?* page, available at <https://www.anaconda.com/what-is-anaconda/>. To install Anaconda, perform the following steps:

1. Click **Anaconda** on the menu, then click **Downloads** to go to the download page at <https://www.anaconda.com/download/#linux>
2. Choose the download suitable for your platform (Linux, OS X, or Windows):
 1. Choose Python 3.6 version*
 2. Choose the Graphical Installer
3. Follow the instructions on the wizard, and in 10 to 20 minutes, your Anaconda environment (Python) setup will be ready

Once the installation process is completed, you can use following command to check the Python version on your Terminal:

```
python -v
```

You should see the following output:

```
Python 3.6 :: Anaconda, Inc.
```

If the command does not work, or returns an error, please check the documentation for help for your platform.

Installing DL libraries

Now, let's install the Python libraries used for DL, specifically, TensorFlow and Keras.



What is TensorFlow?

TensorFlow is a Python library developed and maintained by Google. You can implement many powerful machine learning and DL architectures in custom models and applications using TensorFlow. To find out more, visit <https://www.tensorflow.org/>.

Install the TensorFlow DL library (for all OS except Windows) by typing the following command:

```
conda install -c conda-forge tensorflow
```

Alternatively, you may choose to install using pip and a specific version of TensorFlow for your platform, using the following command:

```
pip install tensorflow==1.6
```

You can find the installation instructions for TensorFlow at https://www.tensorflow.org/get_started/os_setup#anaconda_installation.

Now we will install keras using the following command:

```
pip install keras
```

To validate the environment and the version of the packages, let's write the following script, which will print the version numbers of each library:

```
# Import the tensorflow library
import tensorflow
# Import the keras library
import keras

print('tensorflow: %s' % tensorflow.__version__)
print('keras: %s' % keras.__version__)
```

Save the script as `dl_versions.py`. Run the script by typing the following command:

```
python dl_version.py
```

You should see the following output:

```
tensorflow: 1.6.0
Using TensorFlow backend.
keras: 2.1.5
```

Voila! Now our Python development environment is ready for us to write some awesome DL applications in our local.

Setting up a DL environment in the cloud

All the steps we performed up to now remain the same for the cloud as well, but there are a few additional modules required to configure the cloud virtual machines to make your DL applications servable and scalable. So, before setting up your server, follow the instructions from the preceding section.

To deploy your DL applications in the cloud, you will need a server good enough to train your models and serve at the same time. With huge development in the sphere of DL, the need for cloud servers to practice and deploy projects has increased drastically, and so have the options on the market. The following is a list of some of the best options on offer:

- Paperspace (<https://www.paperspace.com/>)
- FloydHub (<https://www.floydhub.com>)
- Amazon Web Services (<https://aws.amazon.com/>)
- Google Cloud Platform (<https://cloud.google.com/>)
- DigitalOcean (<https://cloud.digitalocean.com/>)

All of these options have their own pro and cons, and the final choice totally depends on your use case and preferences, so feel free to explore more. In this book, we will build and deploy our models mostly on **Google Compute Engine** (GCE), which is a part of **Google Cloud Platform** (GCP). Follow the steps mentioned in this chapter to spin up a VM server and get started.



Google has released an internal notebook platform, **Google Colab** (<https://colab.research.google.com/>), which is pre-installed with all the DL packages and other Python libraries. You can write all of your ML/DL applications on the Google Cloud, leveraging free GPUs for 10 hours.

Cloud platforms for deployment

The main idea behind this book is to empower you to build and deploy DL applications. In this section, we will discuss some critical components required to make your applications accessible to millions of users.

The best way to make your application accessible is to expose it as a web service, using REST or SOAP APIs. To do so, we have many Python web frameworks to choose from, such as `web.py`, Flask, Bottle, and many more. These frameworks allow us to easily build web services and deploy them.

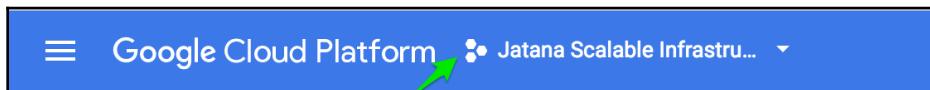
Prerequisites

You should have a Google Cloud (<https://cloud.google.com/>) account. Google is promoting the usage of its platform right now, and is giving away \$300 dollars of credit and 12 months as a free tier user.

Setting up the GCP

Follow these instructions to set up your GCP:

1. **Creating a new project:** Click on the three dots, as shown in the following screenshot, and then click on the + sign to create a new project:



2. **Spinning a VM instance:** Click on the three lines on the upper-left corner of the screen, select the **compute** option, and click on **Compute Engine**. Now choose **Create new instance**. Name the VM instance, and select your zone as **us-west2b**. Choose the **machine type** size.

Choose your boot disk as **Ubuntu 16.04 LTS**. In firewall options, choose both the **http** and **https** option (it's important to make it accessible from the outer world). To opt for GPU options, you can click on **customize** button, and find the GPU options. You can choose between two NVIDIA GPUs. Check both **Allow HTTP traffic** and **Allow HTTPS traffic**.

Now click on **Create**. Boom! your new VM is getting ready.

3. **Modify the firewall settings:** Now click on the **Firewall rules** setting under **Networking**. Under Protocols and Ports, we need to select the port that we will use to export our APIs. We have chosen `tcp:8080` as our port number. Now click on the **Save** button. This will assign a rule in the firewall of your VM to access the applications from the external world.
4. **Boot your VM:** Now start your VM instance. When you see the green tick, click on **SSH**—this will open a command window, and you are now inside the VM. You can also use `gcloud cli` to log in and access your VMs.
5. Then follow the same steps as we performed to set up the local environment, or read further to learn how to create an automation script that will perform all the setup automatically.

Now we need a web framework to write our DL applications as web services—again, there are lots of options, but to make it simple, we will be using a combination of `web.py` and Gunicorn.



If you want to know which web framework to choose based on memory consumption, CPU utilization, and so on, you can have a look at the comprehensive list of benchmarks at <http://klen.github.io/py-frameworks-bench>.

Let's install them using following commands:

```
pip install web.py  
pip install gunicorn
```

Now we are ready to deploy our DL solution as a web service, and scale it to production level.

Automating the setup process

Installing of Python packages and DL libraries can be a tedious process, requiring lots of time and repetitive effort. So, to ease the job, we will create a bash script that can be used to install everything using a single command.

The following is a list of components that will get installed and configured:

- Java 8
- Bazel for building
- Python and associated dependencies
- TensorFlow
- Keras
- Git
- Unzip
- Dependencies for all of the aforementioned services (see the script for exact details)

You can simply download the automation script to your server or locally, execute it, and you're done. Here are the steps to follow:

1. Save the script to your home directory, by cloning the code from the repository:

```
git clone  
https://github.com/PacktPublishing/Python-Deep-Learning-Projects
```

2. Once you have the copy of the complete repository, move to the Chapter01 folder, which will contain a script file named `setupDeepLearning.sh`. This is the script that we will execute to start the setup process, but, before execution, we will have to make it executable using the `chmod` command:

```
cd Python-Deep-Learning-Projects/Chapter01/  
chmod +x setupDeepLearning.sh
```

3. Once this is done, we are ready to execute it as follows:

```
./setupDeepLearning.sh
```

Follow any instructions that appear (basically, say `yes` to everything and accept Java's license). It should take about 10 to 15 minutes to install everything. Once it has finished, you will see the list of Python packages being installed, as shown in the following screenshot:

```
Ready to run TensorFlow!
***Listing modules***
absl-py (0.1.13)
astor (0.6.2)
backports.weakref (1.0.post1)
bleach (1.5.0)
boto (2.38.0)
chardet (2.3.0)
crcmod (1.7)
enum34 (1.1.6)
funcsigs (1.0.2)
futures (3.2.0)
gast (0.2.0)
google-compute-engine (2.7.5)
grpcio (1.10.1)
gunicorn (19.7.1)
html5lib (0.9999999)
Markdown (2.6.11)
mock (2.0.0)
numpy (1.14.2)
pandas (0.22.0)
pbr (4.0.1)
Pillow (5.1.0)
pip (8.1.1)
protobuf (3.5.2.post1)
python-dateutil (2.7.2)
pytz (2018.4)
requests (2.9.1)
scikit-learn (0.19.1)
scipy (1.0.1)
setuptools (20.7.0)
six (1.10.0)
sklearn (0.0)
tensorboard (1.7.0)
tensorflow (1.1.0) ←
termcolor (1.1.0)
urllib3 (1.13.1)
web.py (0.39)
Werkzeug (0.14.1)
wheel (0.29.0)
```

Listed packages with TensorFlow and other Python dependencies

There are a couple of other options, too, such as getting Docker images from TensorFlow and other DL packages, which can set up fully functional DL machines for large-scale and production-ready environments. You can find out more about Docker at <https://www.docker.com/what-docker>. Also, for a quick-start guide, follow the instructions on this repository for an all-in-one Docker image for DL at <https://github.com/floydhub/dl-docker>.

Summary

In this chapter, we worked to get the team set up in a common environment with a standardized toolset. We are looking to deploy our project applications by utilizing Gunicorn and CUDA. Those projects will rely on highly advanced and effective DL libraries, such as TensorFlow and Keras running in Python 2.x/3.x. We'll write our code using the resources in the Anaconda package, and all of this will be running on Ubuntu 16.04 or greater.

Now we are all set to perform experiments and deploy our DL models in production!

16

Training NN for Prediction Using Regression

Welcome to our first proper project in Python deep learning! What we'll be doing today is building a classifier to solve the problem of identifying specific handwriting samples from a dataset of images. We've been asked (in this hypothetical use case) to do this by a restaurant chain that has the need to accurately classify handwritten numbers into digits. What they have their customers do is write their phone numbers in a simple iPad application. At the time when they can be seated, the guest will get a text prompting them to come and see the restaurant's host. We need to accurately classify the handwritten numbers, so that the output from the app will be accurately predicted labels for the digits of a phone number. This can then be sent to their (hypothetical) auto dialer service for text messages, and the notice gets to the right hungry customer!



Define success: A good practice is to define the criteria for success at the beginning of a project. What metric should we use for this project? Let's use a global accuracy test as a percentage to measure our performance in this project.

The data science approach to the problem of classification can be configured in a number of ways. In fact, later in this book, we'll look at how to increase accuracy in image classification with convolutional neural networks.



Transfer learning: This means pretraining a deep learning model on a different (but quite similar) dataset to speed up the rate of learning and accuracy on another (often smaller) dataset. In this project and our hypothetical use case, the pretraining of our deep learning **multi-layer perceptron (MLP)** on the MNIST dataset would enable the deployment of a production system of handwriting classification, without having a huge period of time where we were collecting data samples in a live but non-functional system. Python deep learning projects are cool!

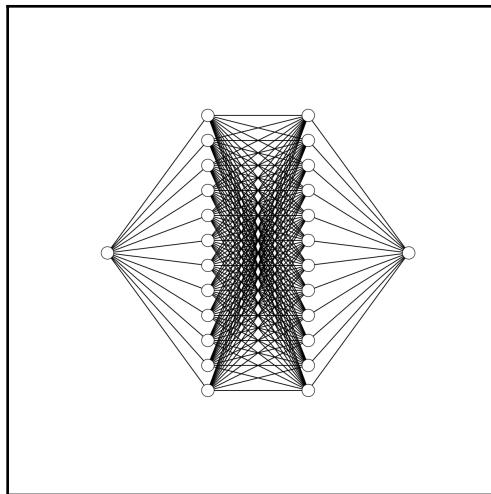
Let's start with the baseline deep neural network model architecture. We will get our intuition and skills firmly established, and this will prepare us for learning more complex architectures to solve a wider variety of problems as we go progress through the projects in this book.

What we'll learn in this chapter includes the following:

- What is an MLP?
- Exploring a common open source handwriting dataset—the MNIST dataset
- Building our intuition and preparations for model architecture
- Coding the model and defining hyperparameters
- Building the training loop
- Testing the model

Building a regression model for prediction using an MLP deep neural network

In any real job working in an AI team, one of the primary goals will be to build regression models that can make predictions in non-linear datasets. Because of the complexity of the real world and the data that you'll be working with, simple linear regression models won't provide the predictive power you're seeking. That is why, in this chapter, we will discuss how to build world-class prediction models using MLP. More information can be found at <http://www.deeplearningbook.org/contents/mlp.html>, and an example of the MLP architecture is shown here:



An MLP with two hidden layers

We will implement a neural network with a simple architecture of only two layers, using TensorFlow, that will perform regression on the MNIST dataset (<http://yann.lecun.com/exdb/mnist/>) that we will provide. We can (and will) go deeper in architecture in later projects! We assume that you are already familiar with backpropagation (if not, please read article on backpropagation by Michal Nielsen at <http://neuralnetworksanddeeplearning.com/chap2.html>). We'll not spend much time on how TensorFlow works, but you can refer to the official tutorial, available at https://www.tensorflow.org/versions/r0.10/get_started/basic_usage.html, if you are interested in looking under the hood of that technology.

Exploring the MNIST dataset

Before we jump into building our awesome neural network, let's first have a look at the famous MNIST dataset. So let's visualize the MNIST dataset in this section.

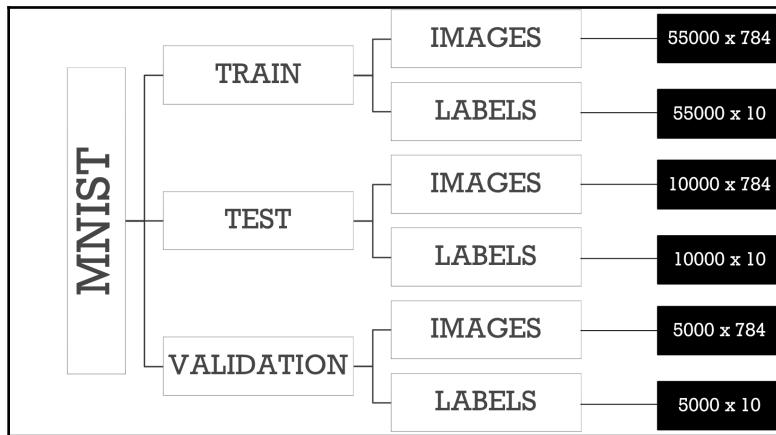


Words of wisdom: You must know your data and how it has been preprocessed, in order to know why the models you build perform the way they do. This section reviews the significant work that has been done in preparation on the dataset, to make our current job of building the MLP easier. Always remember: data science begins with DATA!

Let's start therefore by downloading the data, using the following commands:

```
from tensorflow.examples.tutorials.mnist import input_data  
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

If we examine the `mnist` variable content, we can see that it is structured in a specific format, with three major components—**TRAIN**, **TEST**, and **VALIDATION**. Each set has handwritten images and their respective labels. The images are stored in a flattened way as a single vector:



The format of the MNIST dataset

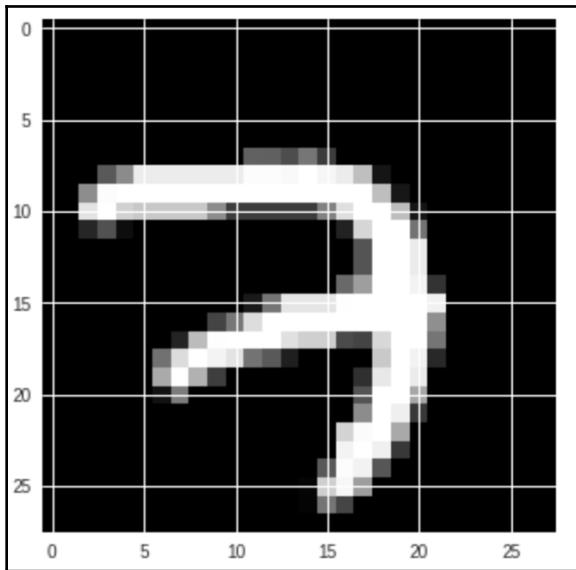
Let's extract one image from the dataset and plot it. Since the stored shape of a single image matrix is `[1, 784]`, we need to reshape these vectors into `[28, 28]` to visualize the original image:

```
sample_image = mnist.train.images[0].reshape([28,28])
```

Once we have the image matrix, we will use `matplotlib` to plot it, as follows:

```
import matplotlib.pyplot as plt  
plt.gray()  
plt.imshow(sample_image)
```

The output will be as follows:



A sample of the MNIST dataset

In the same vein as this image, there are a total of 55,000 similar images of handwritten digits [0-9]. The labels in the MNIST dataset are the true value of the digits present in the image. Our objective, then, is to train a model with this set of images and labels, so that it can predict the labels of any image provided from the MNIST dataset.



Be a deep learning explorer: If you are interested in playing around with the dataset, you can try the Colab Notebook, available at https://drive.google.com/file/d/1-GVlob72EyiJyQpk8EL2fg2mvzaEayJ_/view?usp=sharing.

Intuition and preparation

Let's build our intuition around this project. What we need to do is build a deep learning technology that accurately assigns class labels to an input image. We're using a deep neural network, known as an MLP, to do this. The core of this technology is the mathematics of regression. The specific calculus proofs are outside the scope of this book, but in this section, we provide a foundational basis for your understanding. We also outline the structure of the project, so that it's easy to understand the primary steps needed to create our desired results.

Defining regression

Our first task is to define the model that will perform regression on the provided MNIST dataset. So, we will create a TensorFlow model with two hidden layers as part of a fully connected neural network. You may also hear it referred to as MLP.

The model will perform the operation that will fit the following equation, where y is the label, x is the image, W is the weight that the model will learn, and b is the bias, which will also be learned by the model, following is the regression equation for the model:

$$y = \text{nonlinearity}(xW + b)$$

The regression equation for the model



Supervised learning: When you have data and accurate labels for the training set (that is, you know the answer), you are in a supervised deep learning paradigm. Model training is a mathematical process by which the features of the data are learned and associated with the proper labels, so that when a new data point (test data) is presented, the accurate output class label can be produced. In other words, when you present a new data point and do not have the label (that is, you don't know the answer), your model can produce it for you with a highly reliable class prediction.

Each iteration will try to generalize the values of weight and bias and reduce the error rate. Also, keep in mind that we need to ensure that the model is not overfitting, which may lead to wrong predictions for the unseen dataset. We'll show you how to code this and visualize the progress to aid in your intuition of model performance.

Defining the project structure

Let's structure our project as shown in the following pattern:

- `hy_param.py`: All the hyperparameters and other configurations are defined here
- `model.py`: The definition and architecture of the model are defined here
- `train.py`: The code to train the model is written here
- `inference.py`: The code to execute the trained model and make predictions is defined here
- `/runs`: This folder will store all of the checkpoints that get created during the training process

You can clone the code from the repository—the code for this can be found in the Chapter02 folder, available at <https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/tree/master/Module%2002>.

Let's code the implementation!

To code the implementation, we'll start by defining the hyperparameters, then we will define the model, followed by building and executing the training loop. We conclude by checking to see if our model is overfitting and build an inference code that loads the latest checkpoints and then makes predictions on the basis of learned parameters.

Defining hyperparameters

We will define all of the required hyperparameters in the `hy_param.py` file and then import it as a module in our other codes. This makes it easy in deployment, and is good practice to make your code as modular as possible. Let's look into the hyperparameter configurations that we have in our `hy_param.py` file:

```
#!/usr/bin/env python2

# Hyperparameters and all other kind of params

# Parameters
learning_rate = 0.01
num_steps = 100
batch_size = 128
display_step = 1

# Network Parameters
n_hidden_1 = 300 # 1st layer number of neurons
n_hidden_2 = 300 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

#Training Parameters
checkpoint_every = 100
checkpoint_dir = './runs/'
```

We will be using these values throughout our code, and they're totally configurable.



As a Python deep learning projects exploration opportunity, we invite you, our project teammate and reader, to try different values of learning rate and numbers of hidden layers to experiment and build better models!

Since the flat vectors of images shown previously are of a size of [1 x 786], the num_input=784 is fixed in this case. In addition, the class count in the MNIST dataset is 10. We have digits from 0-9, so obviously we have num_classes=10.

Model definition

First, we will load the Python modules; in this case, the TensorFlow package and the hyperparameters that we defined previously:

```
import tensorflow as tf  
import hy_param
```

Then, we define the placeholders that we will be using to input data into the model. `tf.placeholder` allows us to feed input data to the computational graph. We can define constraints with the shape of the placeholder to only accept a tensor of a certain shape. Note that it is common to provide `None` for the first dimension, which allows us to the size of the batch at runtime.



Master your craft: Batch size can often have a big impact on the performance of deep learning models. Explore different batch sizes in this project. What changes as a result? What's your intuition? Batch size is another tool in your data science toolkit!

We have also assigned names to the placeholders, so that we can use them later on while building our inference code:

```
X = tf.placeholder("float", [None, hy_param.num_input], name="input_x")  
Y = tf.placeholder("float", [None, hy_param.num_classes], name="input_y")
```

Now we will define variables that will hold values for weights and bias.

`tf.Variable` allows us to store and update tensors in our graph. To initialize our variables with random values from a normal distribution, we will use `tf.random_normal()` (more details can be found at https://www.tensorflow.org/api_docs/python/tf/random_normal). The important thing to notice here is the mapping variable size between layers:

```
weights = {  
    'h1': tf.Variable(tf.random_normal([hy_param.num_input,  
                                         hy_param.n_hidden_1])),
```

```

'h2': tf.Variable(tf.random_normal([hy_param.n_hidden_1,
hy_param.n_hidden_2])),
'out': tf.Variable(tf.random_normal([hy_param.n_hidden_2,
hy_param.num_classes]))
}
biases = {
'b1': tf.Variable(tf.random_normal([hy_param.n_hidden_1])),
'b2': tf.Variable(tf.random_normal([hy_param.n_hidden_2])),
'out': tf.Variable(tf.random_normal([hy_param.num_classes]))
}

```

Now, let's set up the operation that we defined in the equation earlier in this chapter. This is the logistic regression operation:

```

layer_1 = tf.add(tf.matmul(X, weights['h1']), biases['b1'])
layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
logits = tf.matmul(layer_2, weights['out']) + biases['out']

```

The logistic values are converted into the probabilistic values using `tf.nn.softmax()`. The softmax activation squashes the output values of each unit to a value between zero and one:

```
prediction = tf.nn.softmax(logits, name='prediction')
```

Next, let's use `tf.nn.softmax_cross_entropy_with_logits` to define our cost function. We will optimize our performance using the Adam Optimizer. Finally, we can use the built-in `minimize()` function to calculate the **stochastic gradient descent (SGD)** update rule for each parameter in our network:

```

loss_op =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=hy_param.learning_rate)
train_op = optimizer.minimize(loss_op)

```

Next, we make our prediction. These functions are needed to calculate and capture the accuracy values in a batch:

```

correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32),
, name='accuracy')

```

The complete code is as follows:

```

#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import tensorflow as tf

```

```
import hy_param

## Defining Placeholders which will be used as inputs for the model
X = tf.placeholder("float", [None, hy_param.num_input], name="input_x")
Y = tf.placeholder("float", [None, hy_param.num_classes], name="input_y")

# Defining variables for weights & bias
weights = {
    'h1': tf.Variable(tf.random_normal([hy_param.num_input,
                                         hy_param.n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([hy_param.n_hidden_1,
                                         hy_param.n_hidden_2])),
    'out': tf.Variable(tf.random_normal([hy_param.n_hidden_2,
                                         hy_param.num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([hy_param.n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([hy_param.n_hidden_2])),
    'out': tf.Variable(tf.random_normal([hy_param.num_classes]))
}

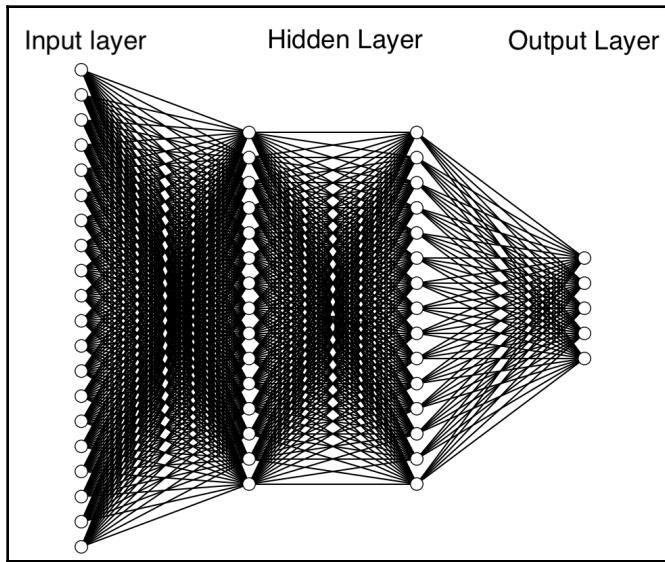
# Hidden fully connected layer 1 with 300 neurons
layer_1 = tf.add(tf.matmul(X, weights['h1']), biases['b1'])
# Hidden fully connected layer 2 with 300 neurons
layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
# Output fully connected layer with a neuron for each class
logits = tf.matmul(layer_2, weights['out']) + biases['out']

# Performing softmax operation
prediction = tf.nn.softmax(logits, name='prediction')

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=hy_param.learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32),
                           name='accuracy')
```

Hurray! The heavy lifting part of the code is done. We save the model code in the `model.py` file. So, up until now, we've defined the simple two-hidden-layer model architecture, with 300 neurons in each layer, which will try to learn the best weight distribution using the Adam Optimizer and predict the probability of ten classes. These layers are shown in the following diagram:



An illustration of the model that we created

Building the training loop

The next step is to utilize the model for training, and record the learned model parameters, which we will accomplish in `train.py`.

Let's start by importing the dependencies:

```
import tensorflow as tf
import hy_param

# MLP Model which we defined in previous step
import model
```

Then, we define the variables that we require to be fed into our MLP:

```
# This will feed the raw images
X = model.X
# This will feed the labels associated with the image
Y = model.Y
```

Let's create the folder to which we will save our checkpoints. Checkpoints are basically the intermediate steps that capture the values of w and b in the process of learning. Then, we will use the `tf.train.Saver()` function (more details on this function can be found at https://www.tensorflow.org/api_docs/python/tf/train/Saver) to save and restore checkpoints:

```
checkpoint_dir = os.path.abspath(os.path.join(hy_param.checkpoint_dir,
                                             "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)

# We only keep the last 2 checkpoints to manage storage
saver = tf.train.Saver(tf.global_variables(), max_to_keep=2)
```

In order to begin training, we need to create a new session in TensorFlow. In this session, we'll initialize the graph variables and feed the model operations the valid data:

```
# Initialize the variables
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, hy_param.num_steps+1):
        # Extracting
        batch_x, batch_y = mnist.train.next_batch(hy_param.batch_size)
        # Run optimization op (backprop)
        sess.run(model.train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % hy_param.display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([model.loss_op, model.accuracy],
                                feed_dict={X: batch_x,
                                           Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))
```

```
        "{}\n".format(path))
    if step % hy_param.checkpoint_every == 0:
        path = saver.save(
            sess, checkpoint_prefix, global_step=step)
        print("Saved model checkpoint to {}\n".format(path))

print("Optimization Finished!")
```

We will extract batches of 128 training image-label pairs from the MNIST dataset and feed them into the model. After subsequent steps or epochs, we will store the checkpoints using the `saver` operation:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from __future__ import print_function

# Import MNIST data
import os
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf
import model
import hy_param

## tf Graph input
X = model.X
Y = model.Y

checkpoint_dir = os.path.abspath(os.path.join(hy_param.checkpoint_dir,
                                             "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
saver = tf.train.Saver(tf.global_variables(), max_to_keep=2)
#loss = tf.Variable(0.0)
# Initialize the variables
init = tf.global_variables_initializer()
all_loss = []
# Start training
with tf.Session() as sess:
    writer_1 = tf.summary.FileWriter("./runs/summary/", sess.graph)
    sum_var = tf.summary.scalar("loss", model.accuracy)
```

```
write_op = tf.summary.merge_all()

# Run the initializer
sess.run(init)

for step in range(1, hy_param.num_steps+1):
    # Extracting
    batch_x, batch_y = mnist.train.next_batch(hy_param.batch_size)
    # Run optimization op (backprop)
    sess.run(model.train_op, feed_dict={X: batch_x, Y: batch_y})
    if step % hy_param.display_step == 0 or step == 1:
        # Calculate batch loss and accuracy
        loss, acc, summary = sess.run([model.loss_op, model.accuracy,
write_op], feed_dict={X: batch_x,
                     Y:
batch_y})
        all_loss.append(loss)
        writer_1.add_summary(summary, step)
        print("Step " + str(step) + ", Minibatch Loss= " + \
              "{:.4f}".format(loss) + ", Training Accuracy= " + \
              "{:.3f}".format(acc))
    if step % hy_param.checkpoint_every == 0:
        path = saver.save(
            sess, checkpoint_prefix, global_step=step)
# print("Saved model checkpoint to {}\n".format(path))

print("Optimization Finished!")

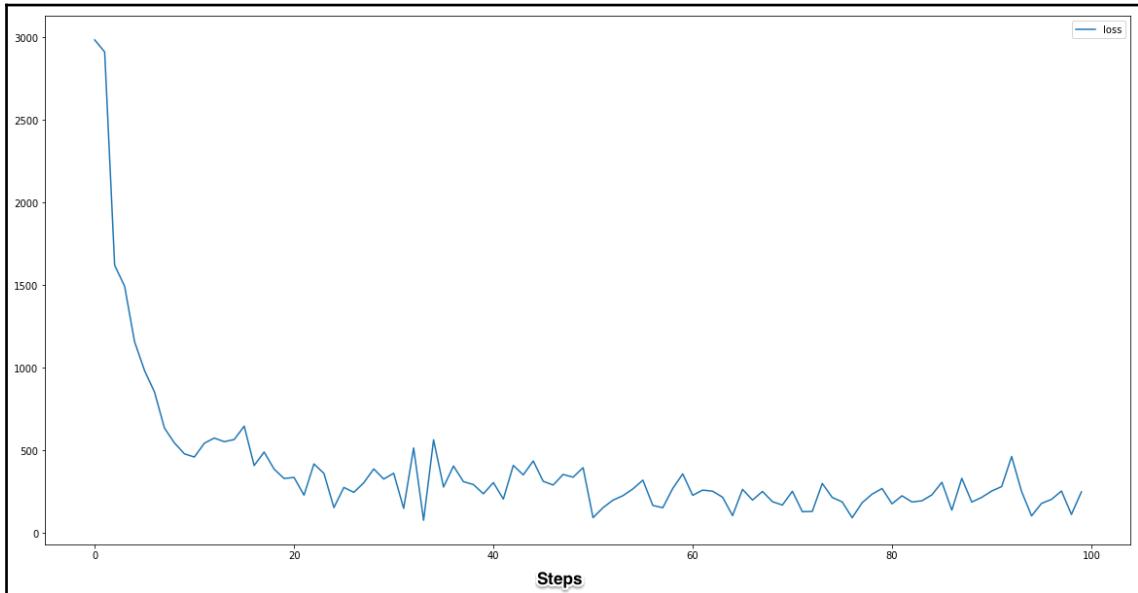
# Calculate accuracy for MNIST test images
print("Testing Accuracy:", \
      sess.run(model.accuracy, feed_dict={X: mnist.test.images,
                                         Y: mnist.test.labels}))
```

Once we have executed the `train.py` file, you will see the progress on your console, as shown in the preceding screenshot. This depicts the loss being reduced after every step, along with accuracy increasing over each step:

```
Step 68, Minibatch Loss= 306.3990, Training Accuracy= 0.852
Step 69, Minibatch Loss= 300.6138, Training Accuracy= 0.844
Step 70, Minibatch Loss= 179.9460, Training Accuracy= 0.867
Step 71, Minibatch Loss= 228.6768, Training Accuracy= 0.844
Step 72, Minibatch Loss= 138.8158, Training Accuracy= 0.938
Step 73, Minibatch Loss= 175.2319, Training Accuracy= 0.906
Step 74, Minibatch Loss= 231.0065, Training Accuracy= 0.867
Step 75, Minibatch Loss= 229.3589, Training Accuracy= 0.883
Step 76, Minibatch Loss= 273.3831, Training Accuracy= 0.891
Step 77, Minibatch Loss= 205.3752, Training Accuracy= 0.891
Step 78, Minibatch Loss= 203.4548, Training Accuracy= 0.844
Step 79, Minibatch Loss= 195.2928, Training Accuracy= 0.883
Step 80, Minibatch Loss= 213.9747, Training Accuracy= 0.859
Step 81, Minibatch Loss= 284.2013, Training Accuracy= 0.836
Step 82, Minibatch Loss= 205.6791, Training Accuracy= 0.859
Step 83, Minibatch Loss= 221.5348, Training Accuracy= 0.875
Step 84, Minibatch Loss= 106.9436, Training Accuracy= 0.875
Step 85, Minibatch Loss= 209.3381, Training Accuracy= 0.867
Step 86, Minibatch Loss= 243.1940, Training Accuracy= 0.867
Step 87, Minibatch Loss= 161.6641, Training Accuracy= 0.859
Step 88, Minibatch Loss= 224.7084, Training Accuracy= 0.844
Step 89, Minibatch Loss= 211.0281, Training Accuracy= 0.875
Step 90, Minibatch Loss= 146.3944, Training Accuracy= 0.883
Step 91, Minibatch Loss= 94.1382, Training Accuracy= 0.898
Step 92, Minibatch Loss= 134.1990, Training Accuracy= 0.938
Step 93, Minibatch Loss= 232.2369, Training Accuracy= 0.867
Step 94, Minibatch Loss= 253.5543, Training Accuracy= 0.898
Step 95, Minibatch Loss= 260.2030, Training Accuracy= 0.859
Step 96, Minibatch Loss= 155.0190, Training Accuracy= 0.914
Step 97, Minibatch Loss= 239.0048, Training Accuracy= 0.828
Step 98, Minibatch Loss= 322.0966, Training Accuracy= 0.836
Step 99, Minibatch Loss= 167.7812, Training Accuracy= 0.883
Step 100, Minibatch Loss= 80.1014, Training Accuracy= 0.914
Optimization Finished!
Testing Accuracy: 0.8742
```

The training epoch's output with minibatch loss and training accuracy parameters

Also, you can see in the plot of minibatch loss, shown in the following diagram, that it approaches toward the minima with each step:



Plotting the loss values computed at each step

It is very important to visualize how your model is performing, so that you can analyze and prevent it from underfitting or overfitting. Overfitting is a very common scenario when you are dealing with the deeper models. Let's spend some time getting to understand them in detail and learning a few tricks to overcome them.

Overfitting and underfitting

With great power comes great responsibility and with deeper models come deeper problems. A fundamental challenge with deep learning is striking the right balance between generalization and optimization. In the deep learning process, we are tuning hyperparameters and often continuously configuring and tweaking the model to produce the best results, based on the data we have for training. This is **optimization**. The key question is, how well does our model generalize in performing predictions on unseen data?

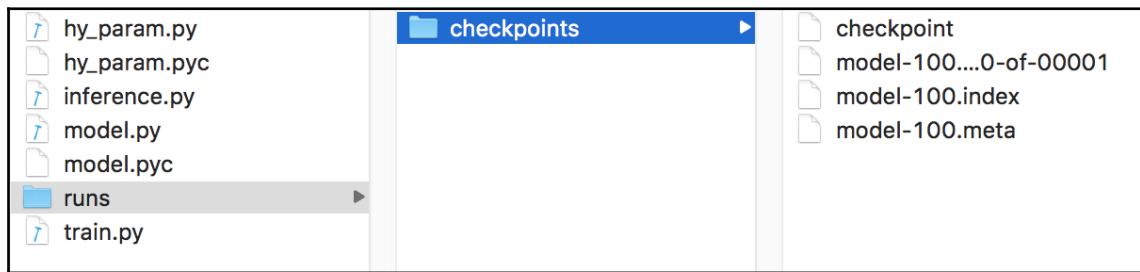
As professional deep learning engineers, our goal is to build models with good real-world generalization. However, generalization is subjective to the model architecture and the training dataset. We work to guide our model for maximum utility by reducing the likelihood that it learns irrelevant patterns or simple similar patterns found in the data used for training. If this is not done, it can affect the generalization process. A good solution is to provide the model with more information that is likely to have a better (that is, more complete and often complex) signal of what you're trying to actually model, by getting more data to train on and to work to optimize the model architecture. Here are few quick tricks that can improve your model by preventing overfitting:

- Getting more data for training
- Reducing network capacity by altering the number of layers or nodes
- Employing L2 (and trying L1) weight regularization techniques
- Adding dropout layers or polling layers in the model



L1 regularization, where the cost added is proportional to the absolute value of the weights coefficients, is also known as *L1 norm*. L2 regularization, where the cost added is proportional to the square of the value of the weight's coefficients, is also known as *L2 norm* or *weight decay*.

When the model gets trained completely, its output, as checkpoints, will get dumped into the `/runs` folder, which will contain the binary dump of `checkpoints`, as shown in the following screenshot:



The checkpoint folder after the training process is completed

Building inference

Now, we will create an inference code that loads the latest checkpoints and then makes predictions on the basis of learned parameters. For that, we need to create a `saver` operation that will pick the latest checkpoints and load the metadata. Metadata contains the information regarding the variables and the nodes that we created in the graph:

```
# Pointing the model checkpoint
checkpoint_file =
tf.train.latest_checkpoint(os.path.join(hy_param.checkpoint_dir,
'checkpoints'))
saver = tf.train.import_meta_graph("{} .meta".format(checkpoint_file))
```

We know the importance of this, because we want to load similar variables and operations back from the stored checkpoint. We load them into memory

using `tf.get_default_graph().get_operation_by_name()`, by passing the operation name in the parameter that we defined in the model:

```
# Load the input variable from the model
input_x =
tf.get_default_graph().get_operation_by_name("input_x").outputs[0]

# Load the Prediction operation
prediction =
tf.get_default_graph().get_operation_by_name("prediction").outputs[0]
```

Now, we need to initialize the session and pass data for a test image to the operation that makes the prediction, as follows:

```
# Load the test data
test_data = np.array([mnist.test.images[0]])

with tf.Session() as sess:
    # Restore the model from the checkpoint
    saver.restore(sess, checkpoint_file)
    # Execute the model to make predictions
    data = sess.run(prediction, feed_dict={input_x: test_data })
    print("Predicted digit: ", data.argmax() )
```

Following is the full code:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from __future__ import print_function
```

```
import os
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

import hy_param

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Pointing the model checkpoint
checkpoint_file =
tf.train.latest_checkpoint(os.path.join(hy_param.checkpoint_dir,
'checkpoints'))
saver = tf.train.import_meta_graph("{}{}.meta".format(checkpoint_file))

# Loading test data
test_data = np.array([mnist.test.images[6]])

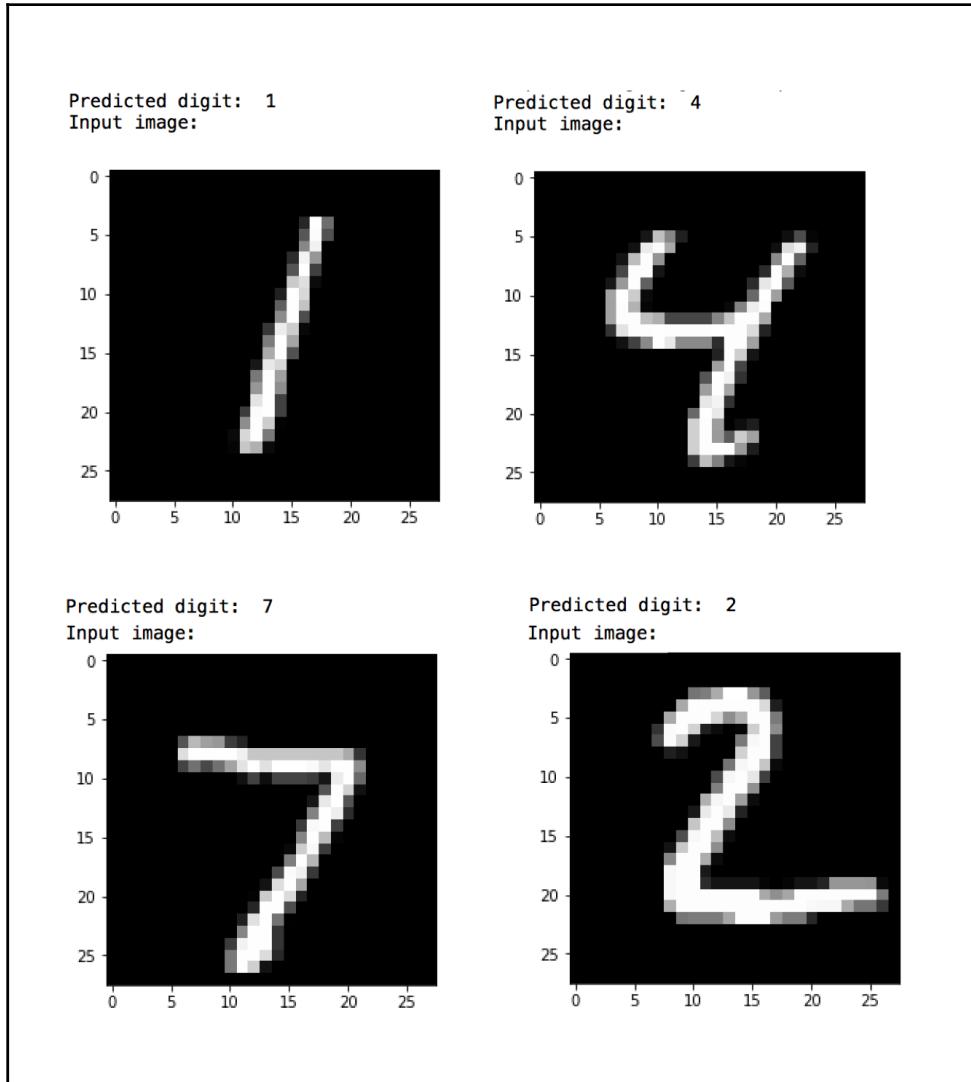
# Loading input variable from the model
input_x =
tf.get_default_graph().get_operation_by_name("input_x").outputs[0]

# Loading Prediction operation
prediction =
tf.get_default_graph().get_operation_by_name("prediction").outputs[0]

with tf.Session() as sess:
    # Restoring the model from the checkpoint
    saver.restore(sess, checkpoint_file)
    # Executing the model to make predictions
    data = sess.run(prediction, feed_dict={input_x: test_data })
    print("Predicted digit: ", data.argmax() )

# Display the feed image
print ("Input image:")
plt.gray()
plt.imshow(test_data.reshape([28,28]))
```

And with that, we are done with our first project that predicts the digits provided in a handwritten image! Here are some of the results that the model predicted when provided with the test image from the MNIST dataset:



The output of the model, depicting the prediction of the model and the input image

Concluding the project

Today's project was to build a classifier to solve the problem of identifying specific handwriting samples from a dataset of images. Our hypothetical use case was to apply deep learning to enable customers of a restaurant chain to write their phone numbers in a simple iPad application, so that they could get a text notification that their party was ready to be seated. Our specific task was to build the intelligence that would drive this application.



Revisit our success criteria: How did we do? Did we succeed? What was the impact of our success? Just as we defined success at the beginning of the project, these are the key questions that we need to ask as deep learning data scientists, as we look to wrap up a project.

Our MLP model accuracy hit 87.42%! Not bad, given the depth of the model and the hyperparameters that we chose at the beginning. See if you can tweak the model to get an even higher test set accuracy.

What are the implications of this accuracy? Let's calculate the incidence of an error occurring that would result in a customer service issue (that is, the customer not getting the text that their table is ready, and getting upset due to an excessively long wait time at the restaurant).

Each customer's phone number is ten digits long. Let's say that our hypothetical restaurant has an average of 30 tables at each location, and those tables turn over two times per night during the rush hour, when the system is likely to be used, and finally, the restaurant chain has 35 locations. This means that each day of operation, there are approximately 21,000 handwritten numbers captured ($30 \text{ tables} \times 2 \text{ turns/day} \times 35 \text{ locations} \times 10 \text{ digit phone number}$).

Obviously, all digits must be correctly classified for the text to get to the waiting restaurant patron. So, any single digit misclassification causes a failure. A model accuracy of 87.42% would improperly classify 2,642 digits per day in our example. The worst case for the hypothetical scenario would be if there occurred only one improperly classified digit in each phone number. Since there are only 2,100 patrons and corresponding phone numbers, this would mean that every phone number had an error in classification (a 100% failure rate), and not a single customer would get their text notification that their party could be seated! The best case, in this scenario, would be if all 10 digits were misclassified in each phone number, which would result in 263 wrong phone numbers out of 2,100 (a 12.5% failure rate). This is still not a level of performance that the restaurant chain would be likely be happy with.



Words of wisdom: Model performance may not equal system or app performance. Many factors contribute to a system being robust or fragile in the real world. Model performance is a key factor, but other items with individual fault tolerances definitely play a part. Know how your deep learning models integrate into the larger project so that you can set proper expectations!

Summary

In the project in this chapter, we successfully built an MLP to produce a regression classification prediction, based on handwritten digits. We gained experience with the MNIST dataset and a deep neural network model architecture, which gave us the added opportunity to define some key hyperparameters. Finally, we looked at the model performance in testing and determined whether we succeeded in achieving our goals.

17

Generative Language Model for Content Creation

This work is certainly getting exciting, and the word is out that we're demonstrating a professional set of deep learning capabilities by producing solutions for a wide range of business use cases! As data scientists, we understand the transferability of our skills. We know that we can provide value by employing core skills when working on problems that we know are similar in structure but that may seem different at first glance. This couldn't be more true than in the next deep learning project. Next, we're (hypothetically) going to be working on a project in which a creative group has asked us to help produce some original content for movie scripts, song lyrics, and even music!

How can we leverage our experience in solving problems for restaurant chains to such a different industry? Let's explore what we know and what we're going to be asked to do. In past project, we demonstrated that we could take an image as input and output a class label (Chapter 16, *Training NN for Prediction Using Regression*);



Define the goal: In this next project, we're going to take the next step in our computational linguistics journey in *Python Deep Learning Projects* and generate new content for our client. We need to help them by providing a deep learning solution that generates new content that can be used in movie scripts, song lyrics, and music.

In this chapter, we will implement a generative model that can generate content using **long short-term memory (LSTM)**, variational autoencoders, and **Generative Adversarial Networks (GANs)**. We will be implementing models for both text and images, which can then generate images and text for artists and various businesses.

In this chapter, we'll cover the following topics:

- Text generation with LSTM
- Additional power of a bi-directional LSTM for text generation
- Deep (multi-layer) LSTM to generate lyrics for a song
- Deep (multi-layer) LSTM music generation for a song

LSTM for text generation

In this section, we'll explore a popular deep learning model: the **recurrent neural network (RNN)**, and how it can be used in the generation of sequence data. The universal way to create sequence data in deep learning is to train a model (usually a RNN or a ConvNet) to predict the next token or next few tokens in a series, based on the previous tokens as input. For instance, let's imagine that we're given the sentence with these words as input: I love to work in deep learning. We will train the network to predict the next character as our target.



When working with textual data, tokens are typically words or characters, and any network that can model the probability of the next token given the previous ones is called a language model that can capture the latent space of language.

Upon training the language model, we can then proceed to feed some initial text and ask it to generate the next token, then add the generated token back into the language model to predict more tokens. For our hypothetical use case, our creative client will use this model and later provide examples of text that we would then be asked to create novel content for in that style.

The first step in building the generative model for text is to import all the modules required. Keras APIs will be used in this project to create the models and Keras utils will be used to download the dataset. In order to build text generation modules, we need a significant amount of simple text data.

You can find the code file for this at https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2002/Chapter06/Basics/generative_text.py:

```
import keras
import numpy as np
from keras import layers
# Gather data
path = keras.utils.get_file(
    'sample.txt',
    origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = open(path).read().lower()
print('Number of words in corpus:', len(text))
```

Data pre-processing

Let's perform the data pre-processing to convert the raw data into its encoded form. We will extract fixed length sentences, encode them using a one-hot encoding process, and finally build a tensor of the (sequence, maxlen, unique_characters) shape, as shown in the following diagram. At the same time, we will prepare the target vector, y , to contain the associated next character that follows each extracted sequence.

The following is the code we'll use to pre-process the data:

```
# Length of extracted character sequences
maxlen = 100

# We sample a new sequence every 5 characters
step = 5

# List to hold extracted sequences
sentences = []

# List to hold the target characters
next_chars = []

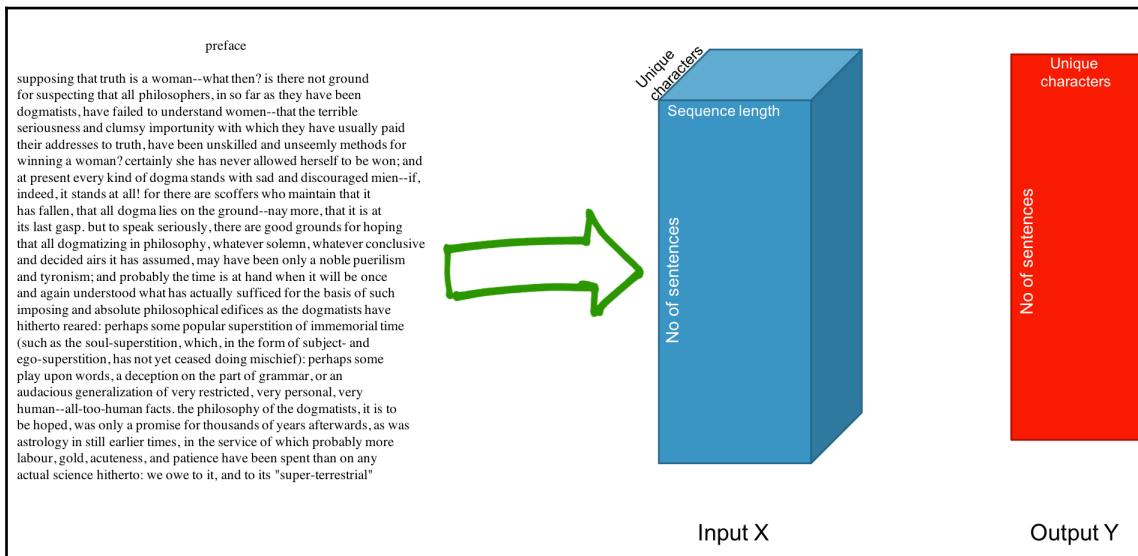
# Extracting sentences and the next characters.
for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('Number of sequences:', len(sentences))

# List of unique characters in the corpus
chars = sorted(list(set(text)))
```

```
# Dictionary mapping unique characters to their index in `chars`
char_indices = dict((char, chars.index(char)) for char in chars)

# Converting characters into one-hot encoding.
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
        y[i, char_indices[next_chars[i]]] = 1
```

Following is how data preprocessing looks like. We have transformed the raw data into the tensors which we will further use for the training purpose:



Defining the LSTM model for text generation

This deep model is a network that's made up of one hidden LSTM layer with 128 memory units, followed by a Dense classifier layer with a softmax activation function over all possible characters. Targets are one-hot encoded, and this means that we'll train the model using categorical_crossentropy as the loss function.

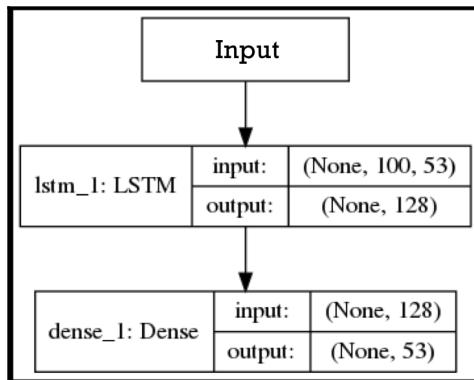
The following code block defines the model's architecture:

```
model = keras.models.Sequential()
model.add(layers.LSTM(128, input_shape=( maxlen, len(chars) )))
model.add(layers.Dense(len(chars), activation='softmax'))
```



```
optimizer = keras.optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

The following diagram helps us visualize the model's architecture:



Training the model

In text generation, the way we choose the succeeding character is crucial. The most common way (greedy sampling) leads to repetitive characters that does not produce a coherent language. This is why we use a different approach called **stochastic sampling**. This adds a degree of randomness to the prediction probability distribution.

Use the following code to re-weight the prediction probability distribution and sample a character index:

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Now, we iterate the training and text generation, beginning with 30 training epochs and then fitting the model for 1 iteration. Then, perform a random selection of the seed text, convert it into one-hot encoding format, and perform predictions of 100 characters. Finally, append the newly generated character to the seed text in each iteration.

After each epoch, generation is performed by utilizing a different temperature from a range of values. This makes it possible to see and understand the evolution of the generated text at model convergence, and the consequences of temperature in the sampling strategy.



Temperature is an LSTM hyperparameter that is used to influence prediction randomness by logit scaling before applying softmax.

We need to execute the following code so that we can train the model:

```
for epoch in range(1, 30):
    print('epoch', epoch)
    # Fit the model for 1 epoch
    model.fit(x, y, batch_size=128, epochs=1, callbacks=callbacks_list)

    # Select a text seed randomly
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('---Seeded text: "' + generated_text + '"')

    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('----- Selected temperature:', temperature)
        sys.stdout.write(generated_text)

        # We generate 100 characters
        for i in range(100):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, char_indices[char]] = 1.

            preds = model.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]

            generated_text += next_char
            generated_text = generated_text[1:]

            sys.stdout.write(next_char)
            sys.stdout.flush()

    print()
```

Inference and results

This gets us to the exciting part of our generative language model—creating custom content! The inference step in deep learning is where we take a trained model and expose it to new data to make predictions or classifications. In the current context of this project, we're looking for model outputs, that is, new sentences, which will be our novel custom content. Let's see what our deep learning model can do!

We will use the following code to store and load the checkpoints into a binary file that stores all of the weights:

```
from keras.callbacks import ModelCheckpoint

filepath="weights-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1,
save_best_only=True, mode='min')
callbacks_list = [checkpoint]
```

Now, we will use the trained model and generate new text:

```
seed_text = 'i want to generate new text after this '
print (seed_text)

# load the network weights
filename = "weights-30-1.545.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')

for temperature in [0.5]:
    print('----- temperature:', temperature)
    sys.stdout.write(seed_text)

    # We generate 400 characters
    for i in range(40):
        sampled = np.zeros((1, maxlen, len(chars)))
        for t, char in enumerate(seed_text):
            sampled[0, t, char_indices[char]] = 1.

        preds = model.predict(sampled, verbose=0)[0]
        next_index = sample(preds, temperature)
        next_char = chars[next_index]

        seed_text += next_char
        seed_text = seed_text[1:]

    sys.stdout.write(next_char)
```

```
    sys.stdout.flush()  
    print()
```

After successfully training the model, we will see the following results at the 30th epoch:

```
--- Generating with seed:  
the "good old time" to which it belongs, and as an expressio"  
----- temperature: 0.2  
the "good old time" to which it belongs, and as an expression of the sense  
of the stronger and subli  
----- temperature: 0.5  
and as an expression of the sense of the stronger and sublication of  
possess and more spirit and in  
----- temperature: 1.0  
e stronger and sublication of possess and more spirit and instinge, and it:  
he ventlumentles, no dif  
----- temperature: 1.2  
d more spirit and instinge, and it: he ventlumentles, no differific and  
does amongly domen--whete ac
```

We find that, with low values for the temperature hyperparameter, the model is able to generate more practical and realistic words. When we use higher temperatures, the generated text becomes more interesting and unusual—some might even say creative. Sometimes, the model will even invent new words that often sound vaguely credible. So, the idea of using low temperature is more reasonable for business use cases where you need to be realistic, while higher temperature values can be used in more creative and artistic use cases.



The art of deep learning and generative linguistic models is a balance between the learned structure and randomness, which makes the output interesting.

Generating lyrics using deep (multi-layer) LSTM

Now that we have built a basic LSTM model for text generation and learned its value, let's move one step further and create a deep LSTM model suited for the task of generating music lyrics. We now have a new goal: to build and train a model that outputs entirely new and original lyrics that is in the style of an arbitrary number of artists.

Let's begin. You can refer to the code file found at [Lyrics-ai \(<https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/tree/master/Module%2002/Chapter06/Lyrics-ai>\)](https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/tree/master/Module%2002/Chapter06/Lyrics-ai) for this exercise.

Data pre-processing

To build a model that can generate lyrics, we will need a huge amount of lyric data, which can easily be extracted from various sources. We collected lyrics from around 10,000 songs and stored them in a text file called `lyrics_data.txt`. You can find the data file in our GitHub repository (https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2002/Chapter06/Lyrics-ai/lyrics_data.txt).

Now that we have our data, we need to convert this raw text into the one-hot encoding version:

```
import numpy as np
import codecs

# Class to perform all preprocessing operations
class Preprocessing:
    vocabulary = {}
    binary_vocabulary = {}
    char_lookup = {}
    size = 0
    separator = '->'

    # This will take the data file and convert data into one hot encoding and
    # dump the vocab into the file.
    def generate(self, input_file_path):
        input_file = codecs.open(input_file_path, 'r', 'utf_8')
        index = 0
        for line in input_file:
            for char in line:
                if char not in self.vocabulary:
                    self.vocabulary[char] = index
                    self.char_lookup[index] = char
                index += 1
        input_file.close()
        self.set_vocabulary_size()
        self.create_binary_representation()

    # This method is to load the vocab into the memory
    def retrieve(self, input_file_path):
        input_file = codecs.open(input_file_path, 'r', 'utf_8')
        buffer = ""
```

```
        for line in input_file:
            try:
                separator_position = len(buffer) +
line.index(self.separator)
                buffer += line
                key = buffer[:separator_position]
                value = buffer[separator_position + len(self.separator):]
                value = np.fromstring(value, sep=',')
                self.binary_vocabulary[key] = value
                self.vocabulary[key] = np.where(value == 1)[0][0]
                self.char_lookup[np.where(value == 1)[0][0]] = key

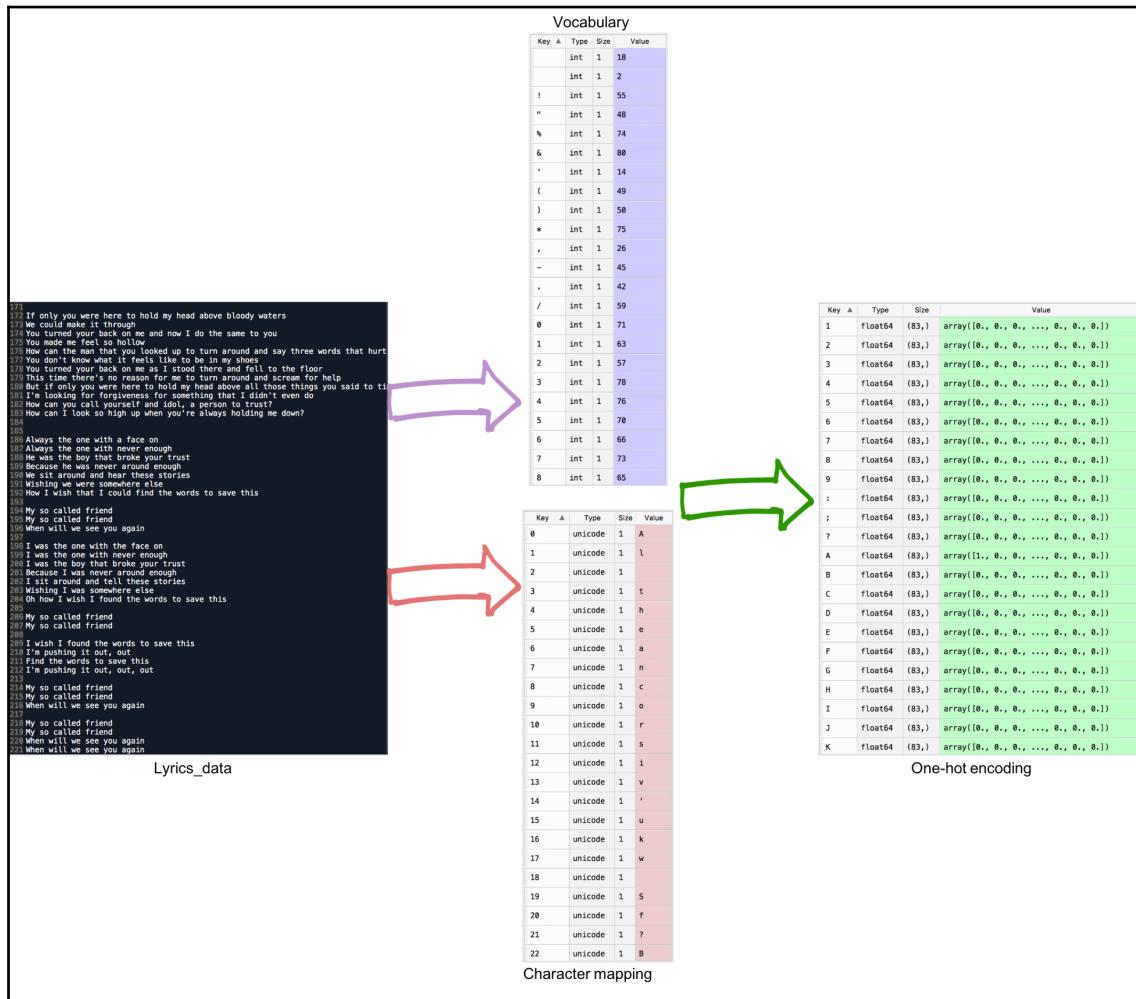
                buffer = ""
            except ValueError:
                buffer += line
        input_file.close()
        self.set_vocabulary_size()

# Below are some helper functions to perform pre-processing.
def create_binary_representation(self):
    for key, value in self.vocabulary.iteritems():
        binary = np.zeros(self.size)
        binary[value] = 1
        self.binary_vocabulary[key] = binary

def set_vocabulary_size(self):
    self.size = len(self.vocabulary)
    print "Vocabulary size: {}".format(self.size)

def get_serialized_binary_representation(self):
    string = ""
    np.set_printoptions(threshold='nan')
    for key, value in self.binary_vocabulary.iteritems():
        array_as_string = np.array2string(value, separator=',',
max_line_width=self.size * self.size)
        string += "{}{}{}\n".format(key.encode('utf-8'),
self.separator, array_as_string[1:len(array_as_string) - 1])
    return string
```

The overall objective of the pre-processing module is to convert the raw text data into one-hot encoding, as shown in the following diagram:



This figure represents the data preprocessing part. The law lyrics data is used to build the vocabulary mapping which is further been transformed into the on-hot encoding.

After the successful execution of the pre-processing module, a binary file will be dumped as `{dataset_filename}.vocab`. This `vocab` file is one of the mandatory files that needs to be fed into the model during the training process, along with the dataset.

Defining the model

We will be using a approach from the Keras model that we used earlier in this project to build this model. To build a more complex model, we will use TensorFlow to write each layer from scratch. TensorFlow gives us, as data scientists and deep learning engineers, more fine-tuned control over our model's architecture.

For this model, we will use the code in the following block to create two placeholders that will store the input and output values:

```
import tensorflow as tf
import pickle
from tensorflow.contrib import rnn

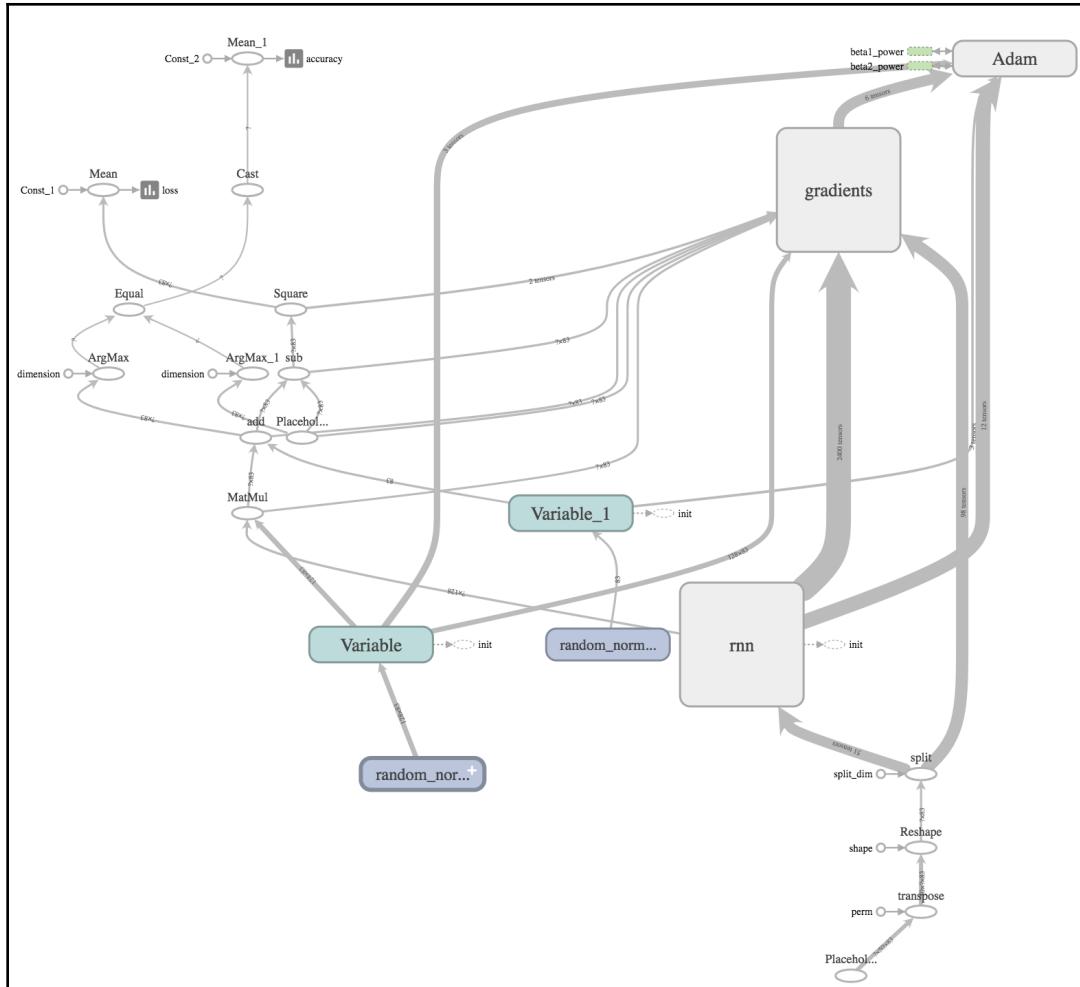
def build(self, input_number, sequence_length, layers_number,
          units_number, output_number):
    self.x = tf.placeholder("float", [None, sequence_length,
                                     input_number])
    self.y = tf.placeholder("float", [None, output_number])
    self.sequence_length = sequence_length
```

Next, we need to store the weights and bias in the variables that we've created:

```
self.weights = {
    'out': tf.Variable(tf.random_normal([units_number,
                                         output_number]))
}
self.biases = {
    'out': tf.Variable(tf.random_normal([output_number]))
}

x = tf.transpose(self.x, [1, 0, 2])
x = tf.reshape(x, [-1, input_number])
x = tf.split(x, sequence_length, 0)
```

We can build this model by using multiple LSTM layers, with the basic LSTM cells assigning each layer with the specified number of cells, as shown in the following diagram:



Tensorboard visualization of the LSTM architecture

The following is the code for this:

```

lstm_layers = []
for i in range(0, layers_number):
    lstm_layer = rnn.BasicLSTMCell(units_number)
    lstm_layers.append(lstm_layer)

deep_lstm = rnn.MultiRNNCell(lstm_layers)

self.outputs, states = rnn.static_rnn(deep_lstm, x,
dtype=tf.float32)

```

```
        print "Build model with input_number: {}, sequence_length: {},\n"
layers_number: {}, ". \
        "units_number: {}, output_number: {}".format(input_number,
sequence_length, layers_number,
                                                units_number,
output_number)
# This method is using to dump the model configurations
    self.save(input_number, sequence_length, layers_number,
units_number, output_number)
```

Training the deep TensorFlow-based LSTM model

Now that we have the mandatory inputs, that is, the dataset file path, the vocab file path, and the model name, we will initiate the training process. Let's define all of the hyperparameters for the model:

```
import os
import argparse
from modules.Model import *
from modules.Batch import *

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--training_file', type=str, required=True)
    parser.add_argument('--vocabulary_file', type=str, required=True)
    parser.add_argument('--model_name', type=str, required=True)

    parser.add_argument('--epoch', type=int, default=200)
    parser.add_argument('--batch_size', type=int, default=50)
    parser.add_argument('--sequence_length', type=int, default=50)
    parser.add_argument('--log_frequency', type=int, default=100)
    parser.add_argument('--learning_rate', type=int, default=0.002)
    parser.add_argument('--units_number', type=int, default=128)
    parser.add_argument('--layers_number', type=int, default=2)
    args = parser.parse_args()
```

Since we are batch training the model, we will divide the dataset into batches of a defined batch_size using the Batch module:

```
batch = Batch(training_file, vocabulary_file, batch_size, sequence_length)
```

Each batch will return two arrays. One will be the input vector of the input sequence, which will have a shape of [batch_size, sequence_length, vocab_size], and the other array will hold the label vector, which will have a shape of [batch_size, vocab_size].

Now, we initialize our model and create the optimizer function. In this model, we used the Adam Optimizer.



The Adam Optimizer is a powerful tool. You can read up on it from the official TensorFlow documentation at
https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer.

Then, we will train our model and perform the optimization over each batch:

```
# Building model instance and classifier
model = Model(model_name)
model.build(input_number, sequence_length, layers_number, units_number,
classes_number)
classifier = model.get_classifier()

# Building cost functions
cost = tf.reduce_mean(tf.square(classifier - model.y))
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Computing the accuracy metrics
expected_prediction = tf.equal(tf.argmax(classifier, 1),
tf.argmax(model.y, 1))
accuracy = tf.reduce_mean(tf.cast(expected_prediction, tf.float32))

# Preparing logs for Tensorboard
loss_summary = tf.summary.scalar("loss", cost)
acc_summary = tf.summary.scalar("accuracy", accuracy)
train_summary_op = tf.summary.merge_all()
out_dir = "{} / {}".format(model_name, model_name)
train_summary_dir = os.path.join(out_dir, "summaries")

##
# Initializing the session and executing the training

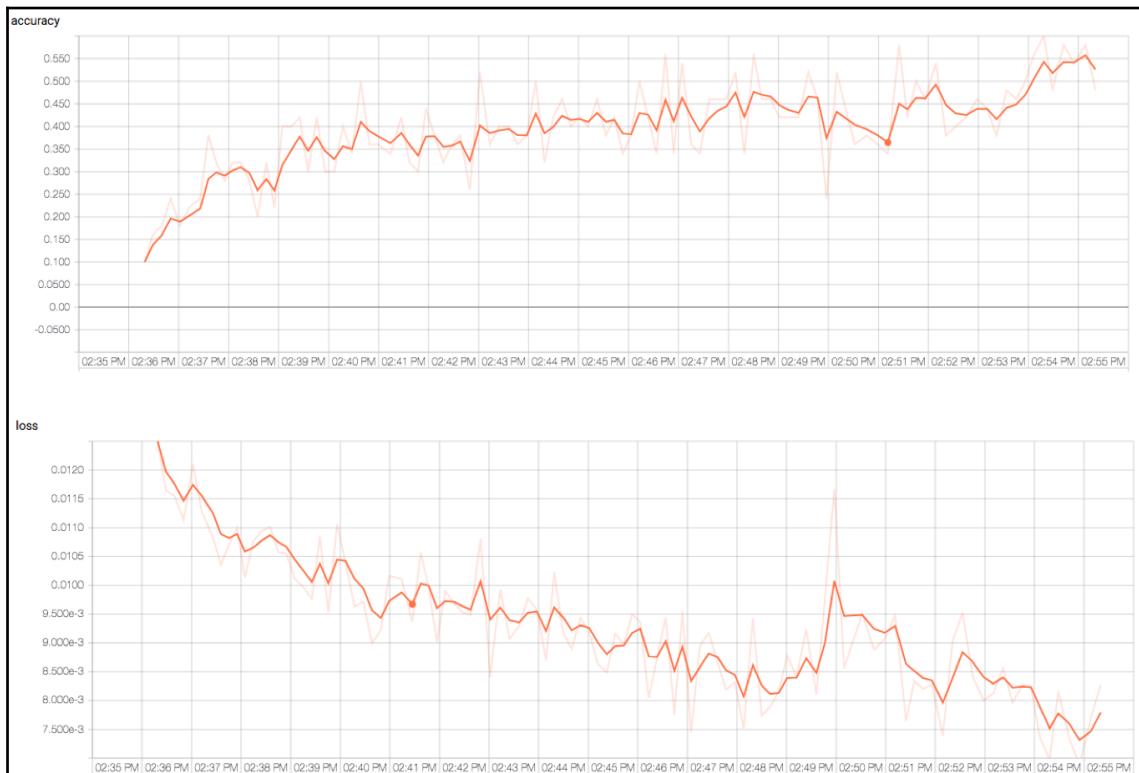
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    iteration = 0

    while batch.dataset_full_passes < epoch:
        iteration += 1
        batch_x, batch_y = batch.get_next_batch()
        batch_x = batch_x.reshape((batch_size, sequence_length,
input_number))
```

```

        sess.run(optimizer, feed_dict={model.x: batch_x, model.y:
batch_y})
        if iteration % log_frequency == 0:
            acc = sess.run(accuracy, feed_dict={model.x: batch_x,
model.y: batch_y})
            loss = sess.run(cost, feed_dict={model.x: batch_x, model.y:
batch_y})
            print("Iteration {}, batch loss: {:.6f}, training accuracy:
{:.5f}".format(iteration * batch_size,
loss, acc))
            batch.clean()
    
```

Once the model completes its training, the checkpoints are stored. We can use later on for inferencing. The following is a graph of the accuracy and the loss that occurred during the training process:



The accuracy (top) and the loss (bottom) plot with respect to the time. We can see that accuracy getting increased and loss getting reduced over the period of time.

Inference

Now that the model is ready, we can use it to make predictions. We will start by defining all of the parameters. While building inference, we need to provide some seed text, just like we did in the previous model. Along with that, we will also provide the path of the vocab file and the output file in which we will store the generated lyrics. We will also provide the length of the text that we need to generate:

```
import argparse
import codecs
from modules.Model import *
from modules.Preprocessing import *
from collections import deque

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--model_name', type=str, required=True)
    parser.add_argument('--vocabulary_file', type=str, required=True)
    parser.add_argument('--output_file', type=str, required=True)

    parser.add_argument('--seed', type=str, default="Yeah, oho ")
    parser.add_argument('--sample_length', type=int, default=1500)
    parser.add_argument('--log_frequency', type=int, default=100)
```

Next, we will load the model by providing the name of model that we used in the training step in the preceding code, and we will restore the vocabulary from the file:

```
model = Model(model_name)
model.restore()
classifier = model.get_classifier()

vocabulary = Preprocessing()
vocabulary.retrieve(vocabulary_file)
```

We will be using the stack methods to store the generated characters, append the stack, and then use the same stack to feed it into the model in an interactive fashion:

```
# Preparing the raw input data
for char in seed:
    if char not in vocabulary.vocabulary:
        print char,"is not in vocabulary file"
        char = u' '
    stack.append(char)
    sample_file.write(char)

# Restoring the models and making inferences
with tf.Session() as sess:
```

```

tf.global_variables_initializer().run()

saver = tf.train.Saver(tf.global_variables())
ckpt = tf.train.get_checkpoint_state(model_name)

if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)

for i in range(0, sample_length):
    vector = []
    for char in stack:
        vector.append(vocabulary.binary_vocabulary[char])
    vector = np.array([vector])
    prediction = sess.run(classifier, feed_dict={model.x:
vector})
    predicted_char =
vocabulary.char_lookup[np.argmax(prediction)]

    stack.popleft()
    stack.append(predicted_char)
    sample_file.write(predicted_char)

    if i % log_frequency == 0:
        print "Progress: {}%".format((i * 100) / sample_length)

sample_file.close()
print "Sample saved in {}".format(output_file)

```

Output

After successful execution, we will get our own freshly brewed, AI generated lyrics reviewed and published. The following is one sample of such lyrics. We have modified some of the spelling so that the sentence makes sense:

Yeah, oho once upon a time, on ir intasd

I got monk that wear your good
So heard me down in my clipp

Cure me out brick
Coway got baby, I wanna sheart in faic

I could sink awlrook and heart your all feeling in the firing of to the
still hild, gavelly mind, have before you, their lead
Oh, oh shor,s sheld be you und make

Oh, fseh where sufl gone for the runtome
Weaaabe the ligavus I feed themust of hear

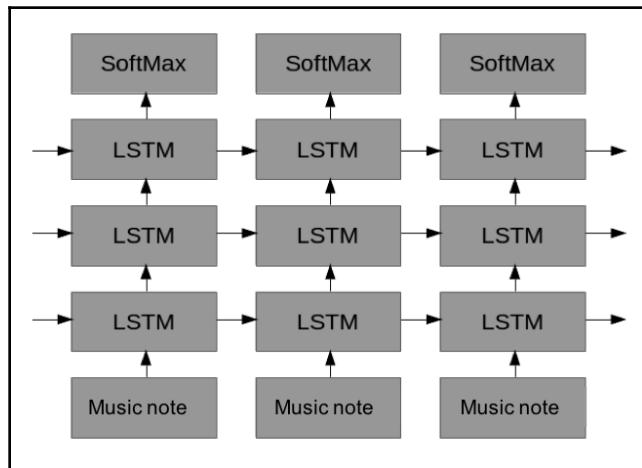
Here, we can see that the model has learned in the way it has generated the paragraphs and sentences with appropriate spacing. It still lacks perfection and also doesn't make sense.



Seeing signs of success: The first task is to create a model that can learn, and then the second one is used to improve on that model. This can be obtained by training the model with a larger training dataset and longer training durations.

Generating music using a multi-layer LSTM

Our (hypothetical) creative agency client loves what we've done in how we can generate music lyrics. Now, they want us to create some music. We will be using multiple layers of LSTMs, as shown in the following diagram:



By now, we know that RNNs are good for sequential data, and we can also represent a music track as notes and chord sequences. In this paradigm, notes become data objects containing octave, offset, and pitch information. Chords become data container objects holding information for the combination of notes played at one time.



Pitch is the sound frequency of a note. Musicians represent notes with letter designations [A, B, C, D, E, F, G], with G being the lowest and A being the highest.

Octave identifies the set of pitches used at any one time while playing an instrument.

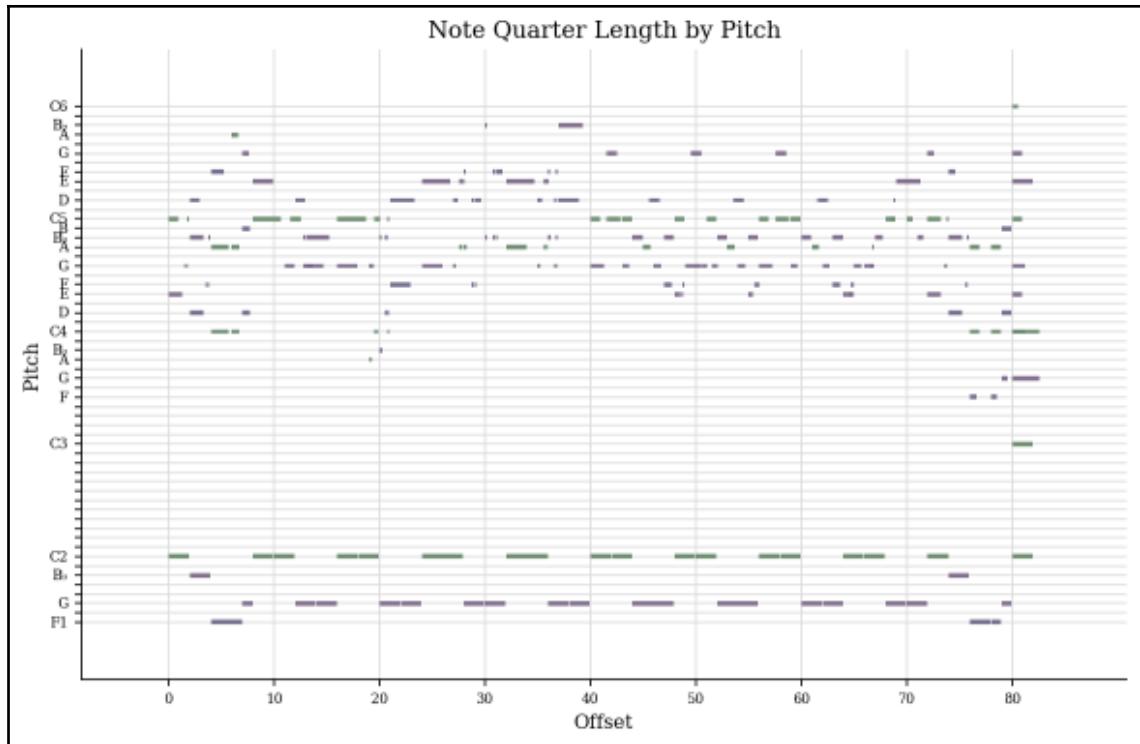
Offset identifies the location of a note in the piece of music.

Let's explore the following section to build our intuition on how to generate music by first processing the sound files, converting them into the sequential mapping data, and then using the RNN to train the model.

Let's do it. You can refer to the Music-ai code for this exercise, which can be found at <https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/tree/master/Module%2002/Chapter06/Music-ai>.

Pre-processing data

To generate music, we will need a good size set of training data of music files. These will be used to extract sequences while building our training dataset. To simplify this process, in this chapter, we are using the soundtrack of a single instrument. We collected some melodies and stored them in MIDI files. The following sample of a MIDI file shows you what this looks like:



The image represents the pitch and note distribution for a sample MIDI file

We can see the intervals between notes, the offset for each note, and the pitch.



To extract the contents of our dataset, we will be using music21. This also takes the output of the model and translates it into musical notation. Music21 (<http://web.mit.edu/music21/>) is a very helpful Python toolkit that's used for computer-aided musicology.

To get started, we will load each file and use the `converter.parse(file)` function to create a `music21.stream` object. We will get a list of all of the notes and chords in the file by using this `stream` object later. Because the most salient features of a note's pitch can be recreated from string notation, we'll append the pitch of every note. To handle chords, we will encode the ID of every note in the chord as a single string, where each note is separated by a dot, and append this to the chord. This encoding process makes it possible for us to decode the model generated output with ease into the correct notes and chords.

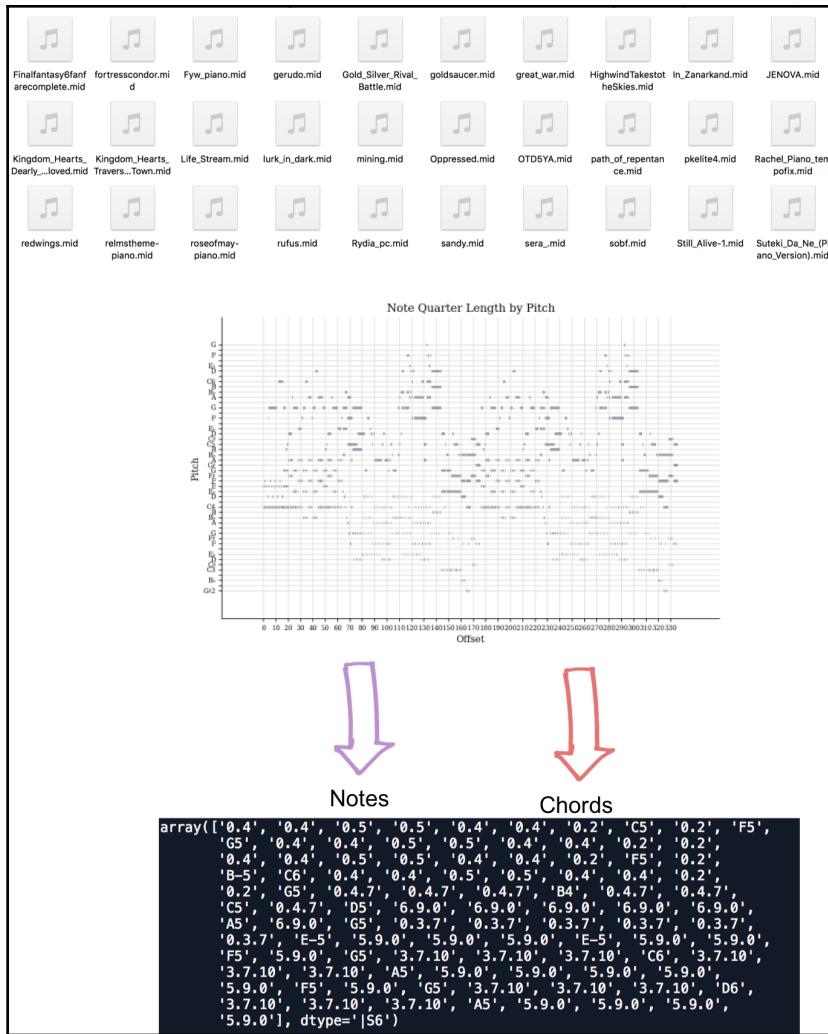
We will load the data from the MIDI files into an array, as you can see in the following code snippet:

```
from music21 import converter, instrument, note, chord
import glob

notes = []

for file in glob.glob("/data/*.mid"):
    midi = converter.parse(file)
    notes_to_parse = None
    parts = instrument.partitionByInstrument(midi)
    if parts: # file has instrument parts
        notes_to_parse = parts.parts[0].recurse()
    else: # file has notes in a flat structure
        notes_to_parse = midi.flat.notes
    for element in notes_to_parse:
        if isinstance(element, note.Note):
            notes.append(str(element.pitch))
        elif isinstance(element, chord.Chord):
            notes.append('.'.join(str(n) for n in element.normalOrder))
```

The next step is to create input sequences for the model and the corresponding outputs, as shown in the following diagram:



The overview of data processing part in which we take the MIDI files, extract the notes and chords from each file and store them as an array.

The model outputs a note or chord for each input sequence. We use the first note or chord, following the input sequence in our list of notes. To complete the final step in data preparation for our network, we need to one-hot encode the output. This normalizes the input for the next iteration.

We can do this with the following code:

```

sequence_length = 100
# get all pitch names

```

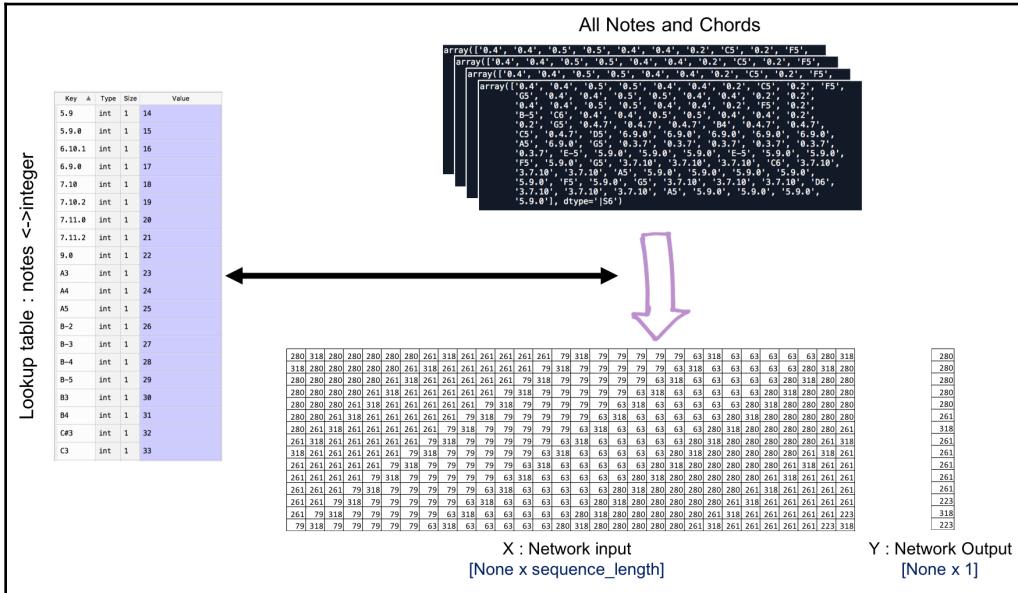
```

pitchnames = sorted(set(item for item in notes))

# create a dictionary to map pitches to integers
note_to_int = dict((note, number) for number, note in
enumerate(pitchnames))
network_input = []
network_output = []
# create input sequences and the corresponding outputs
for i in range(0, len(notes) - sequence_length, 1):
    sequence_in = notes[i:i + sequence_length]
    sequence_out = notes[i + sequence_length]
    network_input.append([note_to_int[char] for char in sequence_in])
    network_output.append(note_to_int[sequence_out])
n_patterns = len(network_input)
# reshape the input into a format compatible with LSTM layers
network_input = numpy.reshape(network_input, (n_patterns, sequence_length,
1))
# normalize input
network_input = network_input / float(n_vocab)
network_output = np_utils.to_categorical(network_output)

```

Now that we have all the notes and chords extracted. We will create our training data X and Y as shown in the following figure:



The captured notes any chords in the array is further transformed into a one-hot encoding vector by mapping the values from the vocabulary. So we will feed the sequences in X matrix and expect the model to learn to predict Y for the given sequence.

Defining the model and training

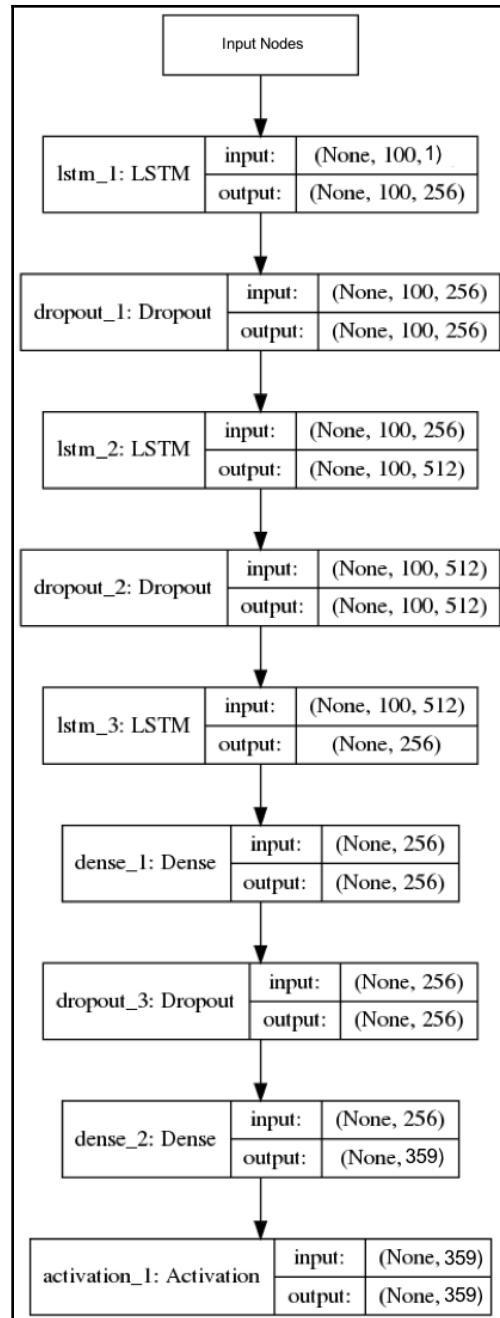
Now, we are getting to the part that all deep learning engineers love: designing the model's architecture! We will be using four distinctive types of layers in our model architecture:

- **LSTM:** This is a type of RNN layer.
- **Dropout:** A technique for regularization. This helps prevent the model from overfitting by randomly dropping some nodes.
- **Dense:** This is a fully connected layer where every input node is connected to every output node.
- **Activation:** This determines the activation function that's going to be used to produce the node's output.

We will again employ the Keras APIs to make the implementation quick:

```
model = Sequential()
model.add(LSTM(
    256,
    input_shape=(network_input.shape[1], network_input.shape[2]),
    return_sequences=True
))
model.add(Dropout(0.5))
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.3))
model.add(Dense(n_vocab))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

The generative model architecture we designed has three LSTM layers, three Dropout layers, two Dense layers, and one Activation layer, as shown in the following diagram:



The model architecture for music generation

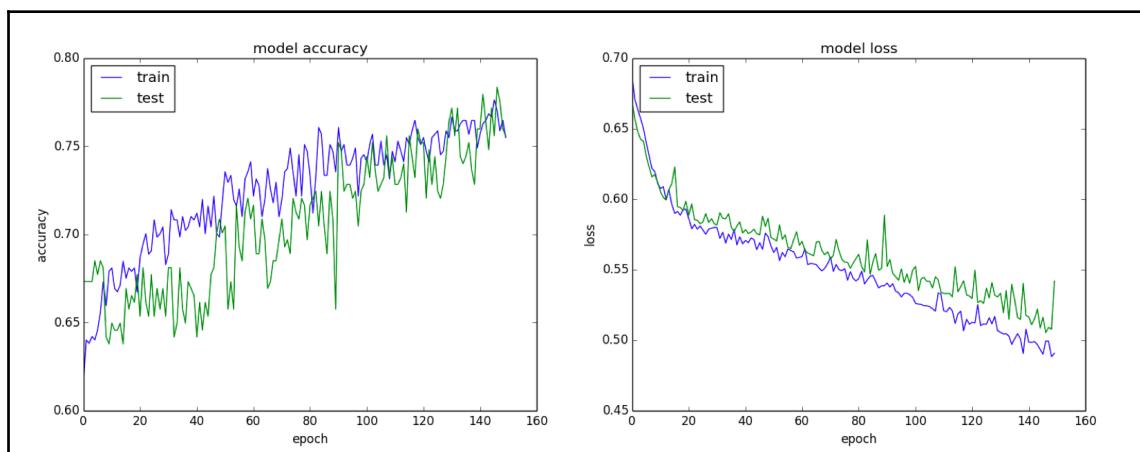
Categorical cross entropy will be used to calculate the loss for each iteration of the training. We will once again use the Adam optimizer in this network. Now that we have our deep learning model architecture configured, it's time to train the model. We have decided to train the model for 200 epochs, each with 25 batches, by using `model.fit()`. We also want to track the reduction in loss over each epoch and will use checkpoints for this purpose.

Now we will perform the training operation and dump the model in the file mentioned in the following code:

```
filepath = "weights-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(
    filepath,
    monitor='loss',
    verbose=0,
    save_best_only=True,
    mode='min'
)
callbacks_list = [checkpoint]

history = model.fit(network_input, network_output, epochs=200,
batch_size=64, callbacks=callbacks_list)
```

The performance of the model can be seen as follows:



The accuracy and the loss plot over the epochs

Now that the training process is completed, we will load the trained models and generate our own music.

Generating music

It's time for the real fun! Let's generate some instrumental music. We will use the code from the model setup and training, but instead of executing the training (as our model is already trained), we will insert the learned weights that we obtained in earlier training.

The following code block executes these two steps:

```
model = Sequential()
model.add(LSTM(
    512,
    input_shape=(network_input.shape[1], network_input.shape[2]),
    return_sequences=True
))
model.add(Dropout(0.5))
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(512))
model.add(Dense(256))
model.add(Dropout(0.3))
model.add(Dense(n_vocab))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')

# Load the weights to each node
model.load_weights('weights_file.hdf5')
```

By doing this, we created the same model, but this time for prediction purposes, and added one extra line of code to load the weights into memory.

Because we need a seed input so that the model can start generating music, we chose to use a random sequence of notes that we obtained from our processed files. You can also send your own nodes as long as you can ensure that the sequence length is precisely 100:

```
# Randomly selected a note from our processed data
start = numpy.random.randint(0, len(network_input)-1)
pattern = network_input[start]

int_to_note = dict((number, note) for number, note in
enumerate(pitchnames))

prediction_output = []

# Generate 1000 notes of music
for note_index in range(1000):
    prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
    prediction_input = prediction_input / float(n_vocab)
```

```
prediction = model.predict(prediction_input, verbose=0)

index = numpy.argmax(prediction)
result = int_to_note[index]
prediction_output.append(result)

pattern.append(index)
pattern = pattern[1:len(pattern)]
```

We iterated the model generation 1,000 times, which created 1,000 notes using the network, producing approximately five minutes of music. The process we used to select the next sequence for each iteration was that we'd start with the first sequence to submit, since it was of the sequence of notes that was at the starting index. For subsequent input sequences, we removed the first note and appended the output from the previous iteration at the end of the sequence. This is a very crude way to do this and is known as the sliding window approach. You can play around and add some randomness to each sequence we select, which could give more creativity to the music that is generated.

It is at this point that we have an array of all of the encoded representations of the notes and chords. To turn this array back into `Note` and `Chord` objects, we need to decode it.

When we detect that the pattern is that of a `Chord` object, we will separate the string into an array of notes. We will then loop through the string's representation of each note to create a `Note` object for each item. The `Chord` object is then created, which contains each of these notes.

When the pattern is that of a `Note` object, we will use the string representation of the pitch pattern to create a `Note` object. At the end of each iteration, we increase the offset by `0.5`, which can again be changed and randomness can be introduced to it.

The following function is responsible for determining whether the output is a `Note` or `Chord` object. Finally, can we use the `music21` `output stream` object to create the MIDI file. Here are a few samples of generated music: https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/tree/master/Module%2002/Chapter06/Music-ai/generated_music.

To execute these steps, you can make use of this `helper` function, as shown in the following code block:

```
def create_midi_file(prediction_output):
    """ convert the output from the prediction to notes and create a midi file"""
    offset = 0
    output_notes = []
```

```
for pattern in prediction_output:  
    # pattern is a chord  
    if ('.' in pattern) or pattern.isdigit():  
        notes_in_chord = pattern.split('.')  
        notes = []  
        for current_note in notes_in_chord:  
            new_note = note.Note(int(current_note))  
            new_note.storedInstrument = instrument.Piano()  
            notes.append(new_note)  
        new_chord = chord.Chord(notes)  
        new_chord.offset = offset  
        output_notes.append(new_chord)  
    # pattern is a note  
else:  
    new_note = note.Note(pattern)  
    new_note.offset = offset  
    new_note.storedInstrument = instrument.Piano()  
    output_notes.append(new_note)  
  
# increase offset each iteration so that notes do not stack  
offset += 0.5  
  
midi_stream = stream.Stream(output_notes)  
  
midi_stream.write('midi', fp='generated.mid')
```

Summary

Wow, that's an impressive set of practical examples of using deep learning projects in Python to build solutions in a creative space! Let's revisit the goals we set up for ourselves.

Defining the goal:



In this project, we're going to take the next step in our computational linguistics journey in deep learning projects in Python and generate new content for our client. We need to help them by providing a deep learning solution that generates new content that can be used in movie scripts, song lyrics, and music.

Deep learning generated content for creative purposes is obviously very tricky. Our realistic goal in this chapter was to demonstrate and train you on the skills and architecture needed to get started on these types of projects. Producing acceptable results takes interacting with the data, the model, and the outputs and testing it with the appropriate audiences. The key takeaway to remember is that the outputs of your models can be quite personalized to the task at hand and that you can expand your thinking of what types of business use cases you should feel comfortable working on in your career.

In this chapter, we implemented a generative model, which generated content with the use of LSTMs. We implemented models for both text and audio that generated content for artists and various businesses in the creative space (hypothetically): the music and movie industries.

What we learned in this chapter was the following:

- Text generation with LSTM
- The additional power of a Bi-directional LSTM for text generation
- Deep (multi-layer) LSTM to generate lyrics for a song
- Deep (multi-layer) LSTM to generate the music for a song

This is some exciting work regarding deep learning, and it keeps on coming in the next chapter. Let's see what's in store!

18

Building Speech Recognition with DeepSpeech2

It's been a great journey, building awesome deep learning projects in Python using image, text, and sound data.

We've been working quite heavily on language models in building chatbots in our previous chapters. Chatbots are a powerful tool for customer engagement and the automation of a wide range of business processes from customer service to sales. Chatbots enable the automation of repetitive and/or redundant interactions such as frequently asked questions or product-ordering workflows. This automation saves time and money for businesses and enterprises. If we've done our job well as deep-learning engineers, it also means that the consumers are receiving a much-improved **user experience (UX)** as a result.

The new interaction between a business and its customers via a chatbot is very effective in each party receiving value. Let's look at the interaction scenario and see if we can identify any constraints that should be the focus of our next project. Up until now, all of our chat interactions have been through text. Let's think about what this means for the consumer. Text interactions are often (but not exclusively) initiated via mobile devices. Secondly, chatbots open up a new **user interaction (UI)**—for conversational UI. Part of the power of conversational UI is that it can remove the constraint of the physical keyboard and open the range of locations and devices that are now possible for this interaction to take place.



Conversational UI is made possible by speech recognition systems working through popular devices, such as your smartphone with Apple's Siri, Amazon's Echo, and Google Home. It's very cool technology, consumers love it, and businesses that adopt this technology gain an advantage over those in their industry that do not keep up with the times.

In this chapter, we will build a system that recognizes English speech, using the **DeepSpeech2 (DS2)** model.

You will learn the following:

- To work with speech and spectrograms
- To build an end-to-end speech recognition system
- The **Connectionist Temporal Classification (CTC)** loss function
- Batch normalization and SortaGrad for **recurrent neural networks (RNNs)**

Let's get started and deep dive into the speech data, learn to feature engineer the speech data, extract various kinds of features from it, and then build a speech recognition system that can detect your or a registered user's voice.



Define the goal: The goal of this project is to build and train an **automatic speech recognition (ASR)** system to take in and convert an audio call to text that could then be used as input for a text-based chatbot that could understand and respond.

Data preprocessing

In this project, we will use *LibriSpeech ASR corpus* (<http://www.openslr.org/12/>), which is 1,000 hours of 16 kHz-read English speech.

Let's use the following commands to download the corpus and unpack the LibriSpeech data:

```
mkdir -p data/librispeech
cd data/librispeech
wget http://www.openslr.org/resources/12/train-clean-100.tar.gz
wget http://www.openslr.org/resources/12/dev-clean.tar.gz
wget http://www.openslr.org/resources/12/test-clean.tar.gz
mkdir audio
cd audio
tar xvzf ../../train-clean-100.tar.gz LibriSpeech/train-clean-100 --strip-components=1
tar xvzf ../../dev-clean.tar.gz LibriSpeech/dev-clean --strip-components=1
tar xvzf ../../test-clean.tar.gz LibriSpeech/test-clean --strip-components=1
```

This will take a while and once the process is completed, we will have the data folder structure, as shown in the following screenshot:

```
.  
|-- audio  
    |-- train-clean-100  
        |-- 12722  
            |-- 1281042  
                |-- 12722-1281042-0000.flac  
                |-- 12722-1281042-0001.flac  
                |-- 12722-1281042-00030.flac  
                |-- 12722-1281042-trans.txt  
    |-- dev-clean  
        |-- 1272  
            |-- 128104  
                |-- 1272-128104-0000.flac  
                |-- 1272-128104-0001.flac  
                |-- 1272-128104-0002.flac  
                |-- 1272-128104-0013.flac  
                |-- 1272-128104-trans.txt  
    |-- test-clean  
        |-- 12722  
            |-- 1281042  
                |-- 12722-1281042-0000.flac  
                |-- 12722-1281042-0001.flac  
                |-- 12722-1281042-trans.txt  
|-- dev-clean.tar.gz  
|-- train-clean-100.tar.gz  
`-- test-clean.tar.gz
```

We now have three folders named as `train-clean-100`, `dev-clean`, and `test-clean`. Each folder will have subfolders that are the associated IDs used for mapping the small segment of the transcript and the audio. All the audio files are in the `.flac` extension, and all the folders will have one `.txt` file, which is the transcript for the audio files.

Corpus exploration

Let's explore the dataset in detail. First, let's look into the audio file by reading it from the file and plotting it. To read the audio file, we will use the `pysoundfile` package with the following command:

```
pip install pysoundfile
```

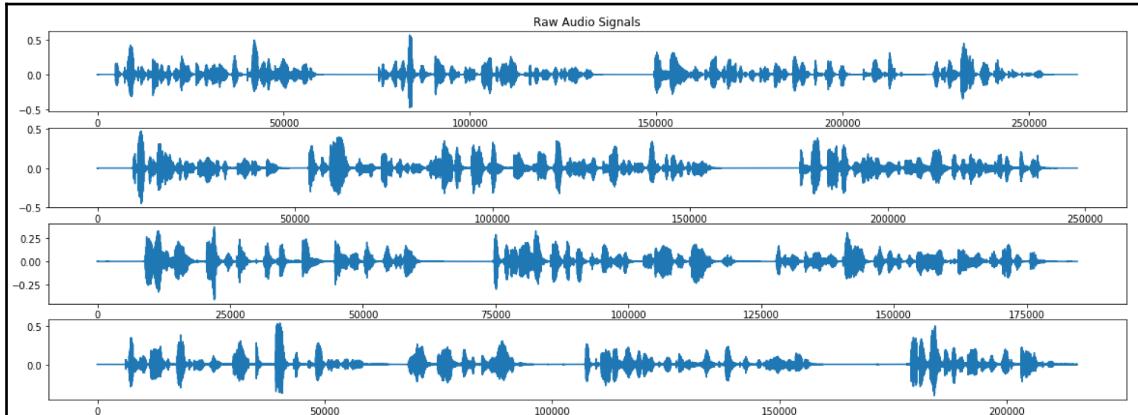
Next, we will import the modules, read the audio files, and plot them with the following code block:

```
import soundfile as sf
import matplotlib.pyplot as plt

def plot_audio(audio):
    fig, axs = plt.subplots(4, 1, figsize=(20, 7))
    axs[0].plot(audio[0]);
    axs[0].set_title('Raw Audio Signals')
    axs[1].plot(audio[1]);
    axs[2].plot(audio[2]);
    axs[3].plot(audio[3]);

    audio_list = []
    for i in xrange(4):
        file_path = 'data/128684/911-128684-000{}.flac'.format(i+1)
        a, sample_rate = sf.read(file_path)
        audio_list.append(a)
    plot_audio(audio_list)
```

The following is the frequency representation of each segment of speech:



The raw audio signal plot from the audio MIDI file

Now let's look into the content of the transcript text file. It's a clean version of the text with the audio file IDs in the beginning and the associated text following:

```
911-128684-0001 OR CONCENTRATING ITSELF IN THE SHAPE OF PASSION OR EMOTION CAN BE DIRECTLY FELT AS THE SPIRITUAL ACTIVITY WHICH IT IS AND KNOWN IN CONTRAST WITH THE SPACE FII  
911-128684-0002 IN OPPOSITION TO THIS DUALISTIC PHILOSOPHY I TRIED IN THE FIRST ESSAY TO SHOW THAT THOUGHTS AND THINGS ARE ABSOLUTELY HOMOGENEOUS AS TO THEIR MATERIAL AND THI  
911-128684-0003 THERE IS NO THOUGHT STUFF DIFFERENT FROM THING STUFF I SAID BUT THE SAME IDENTICAL PIECE OF PURE EXPERIENCE WHICH WAS THE NAME I GAVE TO THE MATERIA PRIMA OF  
911-128684-0004 CAN STAND ALTERNATELY FOR A FACT OF CONSCIOUSNESS OR FOR A PHYSICAL REALITY ACCORDING AS IT IS TAKEN IN ONE CONTEXT OR IN ANOTHER FOR THE RIGHT UNDERSTANDING
```

Audio file names	
Transcript data	
911-128684-0001	OR CONCENTRATING ITSELF IN THE SHAPE OF PASSION OR
911-128684-0002	IN OPPOSITION TO THIS DUALISTIC PHILOSOPHY I TRIED
911-128684-0003	THERE IS NO THOUGHT STUFF DIFFERENT FROM THING STUF
911-128684-0004	CAN STAND ALTERNATELY FOR A FACT OF CONSCIOUSNESS O

The transcript data is stored a specific format. Left numbers are the midi file name and the right part is the actually transcript. This helps in building the mapping between the midi file and its respective transcript.

What we see is that each audio file is the narration of the transcript contained in the file. Our model will try to learn this sequence pattern. But before we work on the model, we need to extract some features from the audio file and convert the text into one-hot encoding format.

Feature engineering

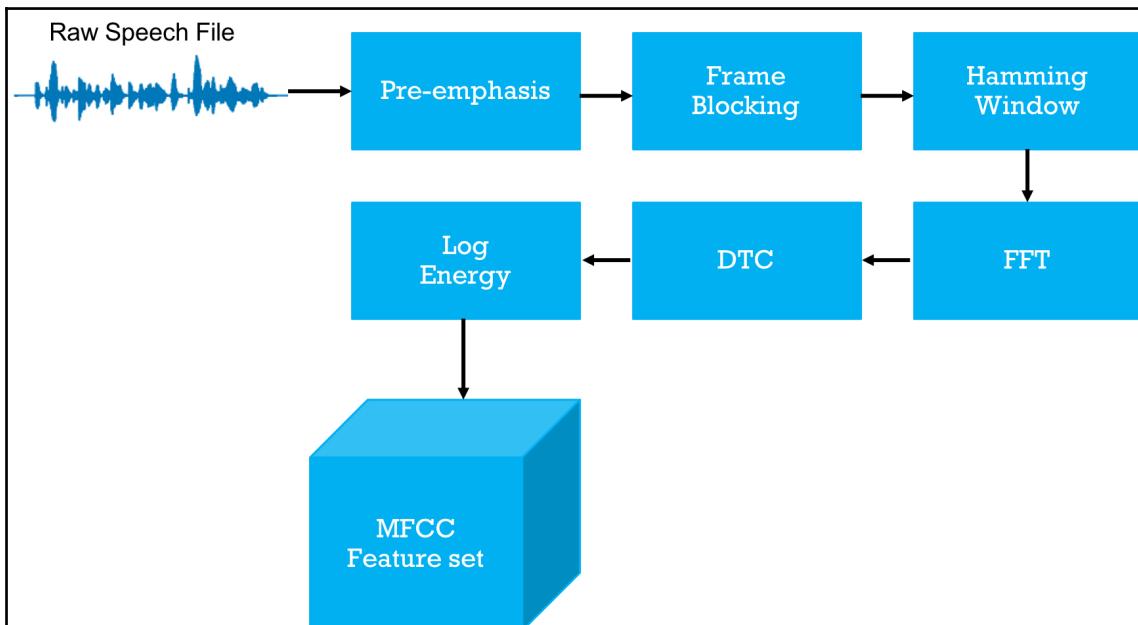
So, before we feed the raw audio data into our model, we need to transform the data into numerical representations that are features. In this section, we will explore various techniques to extract features from the speech data that we can use to feed into the model. The accuracy and performance of the model vary based on the type of features we use. As an inquisitive deep-learning engineer, it's your opportunity to explore and learn the features with these techniques and use the best one for the use case at hand.

The following table gives us a list of techniques and their properties:

Techniques	Properties
Principal component analysis (PCA)	<ul style="list-style-type: none"> Eigenvector-based method Non-linear feature extraction method Supported to linear map Faster than other techniques Good for Gaussian data
Linear discriminate analysis (LDA)	<ul style="list-style-type: none"> Linear feature extraction method Supported to the supervised linear map Faster than other techniques Better than PCA for classification
Independent component analysis (ICA)	<ul style="list-style-type: none"> Blind course separation method Support to linear map Iterative in nature Good for non-Gaussian data

Cepstral analysis	<ul style="list-style-type: none"> • Static feature extraction method • Power spectrum method • Used to represent spectral envelope
Mel-frequency scale analysis	<ul style="list-style-type: none"> • Static feature extraction method • Spectral analysis method • Mel scale is calculated
Mel-frequency cepstral coefficient (MFCCs)	<ul style="list-style-type: none"> • Power spectrum is computed by performing Fourier Analysis • Robust and dynamic method for speech feature extraction
Wavelet technique	<ul style="list-style-type: none"> • Better time resolution than Fourier transform • Real-time factor is minimum

The MFCC technique is the most efficient and is often used for the extraction of speech features for speech recognition. The MFCC is based on the known variation of the human ear's critical bandwidth frequencies, with filters spaced linearly at low frequencies. The process of MFCC is shown in the following diagram:



Block diagram of MFCC process

For our implementation purposes, we are not going to perform each step; instead, we will use a Python package called `python_speech_features` that provides common speech features for ASR, including MFCCs and filterbank energies.

Let's pip install the package with the following command:

```
pip install python_speech_features
```

So, let's define a function that will normalize the audio time series data and extract the MFCC features:

```
from python_speech_features import mfcc

def compute_mfcc(audio_data, sample_rate):
    """ Computes the MFCCs.

    Args:
        audio_data: time series of the speech utterance.
        sample_rate: sampling rate.

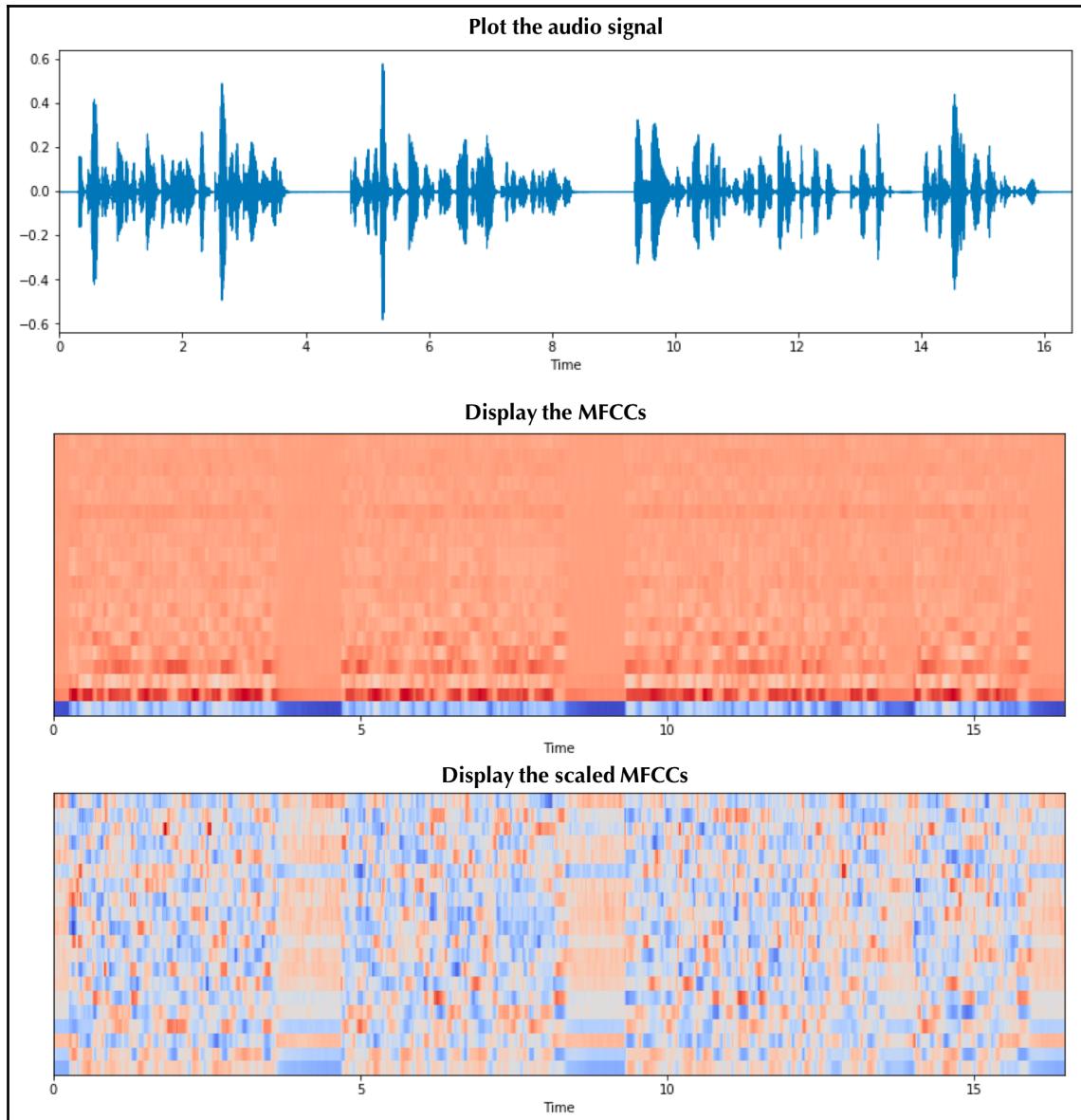
    Returns:
        mfcc_feat: [num_frames x F] matrix representing the mfcc.
    """

    audio_data = audio_data - np.mean(audio_data)
    audio_data = audio_data / np.max(audio_data)
    mfcc_feat = mfcc(audio_data, sample_rate, winlen=0.025, winstep=0.01,
                     numcep=13, nfilt=26, nfft=512, lowfreq=0,
                     highfreq=None,
                     preemph=0.97, ceplifter=22, appendEnergy=True)
    return mfcc_feat
```

Let's plot the audio and MFCC features and visualize them:

```
audio, sample_rate = sf.read(file_path)
feats[audio_file] = compute_mfcc(audio, sample_rate)
plot_audio(audio, feats[audio_file])
```

The following is the output of the spectrogram:



Data transformation

Once we have all the features that we need to feed into the model, we will transform the raw NumPy tensors into the TensorFlow specific format called TFRecords.

In the following code snippet, we are creating the folders to store all the processed records. The `make_example()` function creates the sequence example for a single utterance given the sequence length, MFCC features, and corresponding transcript. Multiple sequence records are then written into TFRecord files using the

`tf.python_io.TFRecordWriter()` function:

```
if os.path.basename(partition) == 'train-clean-100':
    # Create multiple TFRecords based on utterance length for training
    writer = {}
    count = {}
    print('Processing training files...')
    for i in range(min_t, max_t+1):
        filename = os.path.join(write_dir, 'train' + '_' + str(i) +
                               '.tfrecords')
        writer[i] = tf.python_io.TFRecordWriter(filename)
        count[i] = 0

    for utt in tqdm(sorted_utts):
        example = make_example(utt_len[utt], feats[utt].tolist(),
                               transcripts[utt])
        index = int(utt_len[utt]/100)
        writer[index].write(example)
        count[index] += 1

    for i in range(min_t, max_t+1):
        writer[i].close()
    print(count)

    # Remove bins which have fewer than 20 utterances
    for i in range(min_t, max_t+1):
        if count[i] < 20:
            os.remove(os.path.join(write_dir, 'train' +
                                   '_' + str(i) + '.tfrecords'))
else:
    # Create single TFRecord for dev and test partition
    filename = os.path.join(write_dir, os.path.basename(write_dir) +
                           '.tfrecords')
    print('Creating', filename)
    record_writer = tf.python_io.TFRecordWriter(filename)
    for utt in sorted_utts:
        example = make_example(utt_len[utt], feats[utt].tolist(),
                               transcripts[utt])
```

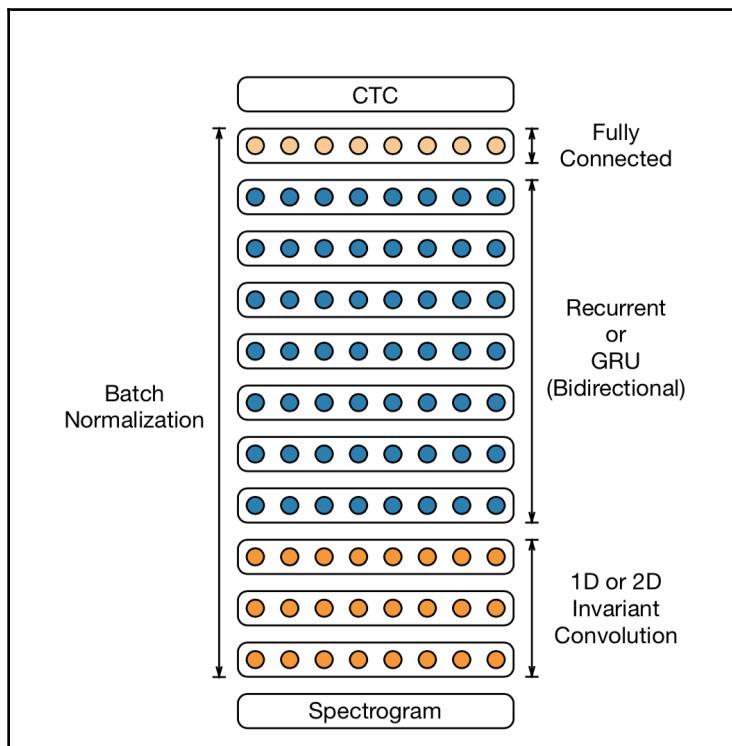
```
record_writer.write(example)
record_writer.close()
print('Processed '+str(len(sorted_utts))+' audio files')
```

All the data-processing code is written in the preprocess_LibriSpeech.py file, which will perform all the previously mentioned data manipulation part, and once the operation is complete, the resulting processed data gets stored at the data/librispeech/processed/ location. Use the following command to run the file:

```
python preprocess_LibriSpeech.py
```

DS2 model description and intuition

DS2 architecture is composed of many layers of recurrent connections, convolutional filters, and non-linearities, as well as the impact of a specific instance of batch normalization, applied to RNNs, as shown here:



To learn from datasets with a large amount of data, DS2 model's capacity is increased by adding more depth. The architectures are made up to 11 layers of many bidirectional recurrent layers and convolutional layers. To optimize these models successfully, batch normalization for RNNs and a novel optimization curriculum called SortaGrad were used.

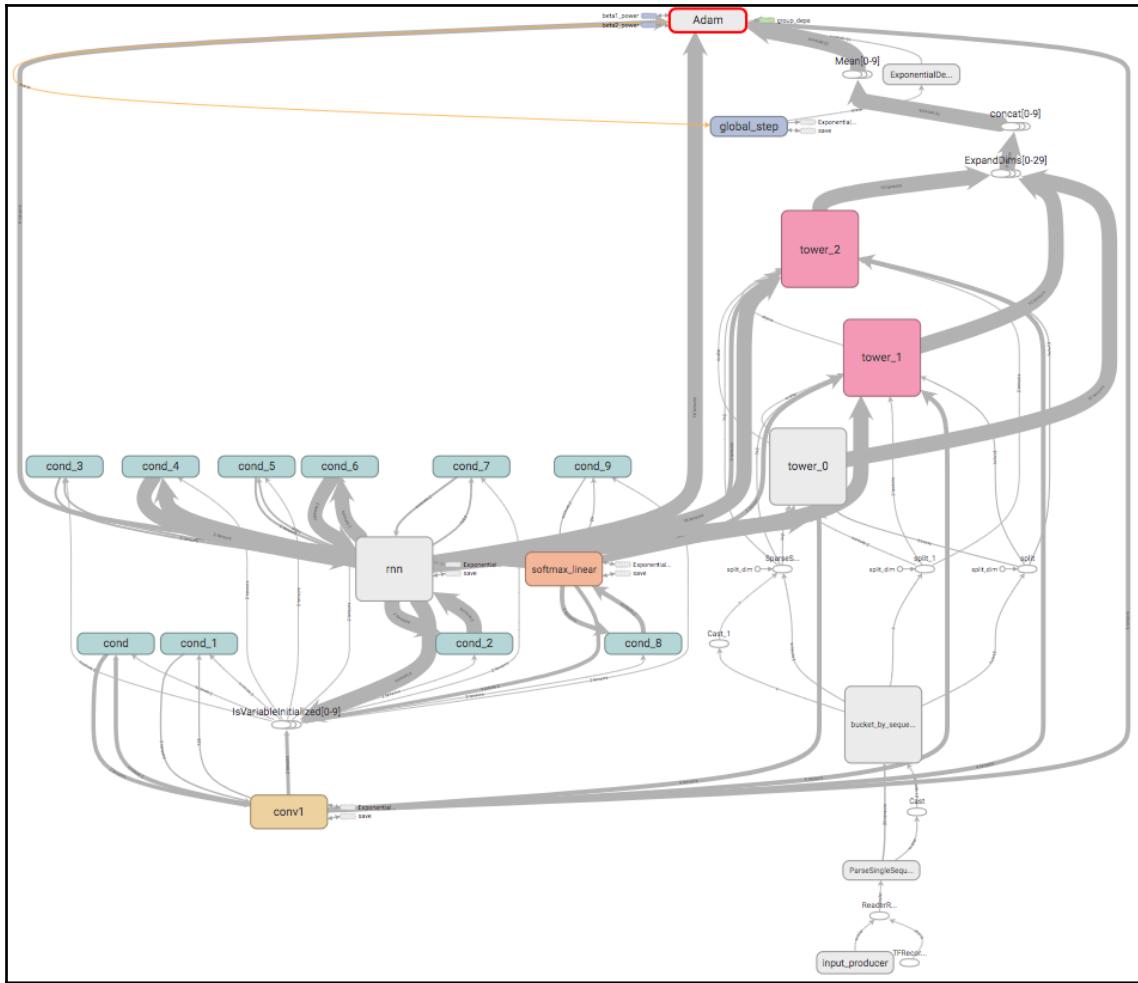
The training data is a combination of input sequence $x(i)$ and the transcript $y(i)$, whereas the goal of the RNN layers is to learn the features between $x(i)$ and $y(i)$:

```
training set X = { (x(1), y(1)), (x(2), y(2)), . . . }
utterance = x(i)
label = y(i)
```

The spectrogram of power normalized audio clips are used as the features to the system and the outputs of the network are the graphemes of each language. In terms of adding non-linearity, the clipped **rectified linear unit (ReLU)** function $\sigma(x) = \min\{\max\{x, 0\}, 20\}$ was used. After the bidirectional recurrent layers, one or more fully connected layers are placed and the output layer L is a softmax, computing a probability distribution over characters.

Now let's look into the implementation of the DS2 architecture.

The following is what the model looks like in TensorBoard:



For the convolution layers, we have the kernel of size [11, input_seq_length, number_of_filter] followed by the 2D convolution operation on the input sequence, and then dropout is applied to prevent overfitting.

The following code segment executes these steps:

```
with tf.variable_scope('conv1') as scope:
    kernel = _variable_with_weight_decay(
        'weights',
        shape=[11, feat_len, 1, params.num_filters],
```

```

        wd_value=None, use_fp16=params.use_fp16)

    feats = tf.expand_dims(feats, dim=-1)
    conv = tf.nn.conv2d(feats, kernel,
                       [1, params.temporal_stride, 1, 1],
                       padding='SAME')
    biases = _variable_on_cpu('biases', [params.num_filters],
                              tf.constant_initializer(-0.05),
                              params.use_fp16)
    bias = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(bias, name=scope.name)
    _activation_summary(conv1)

    # dropout
    conv1_drop = tf.nn.dropout(conv1, params.keep_prob)

```

Then, we next have the recurrent layer, where we reshape the output of the convolution layer to fit the data into the RNN layer. Then, the custom RNN cells are created based on the hyperparameter called `rnn_type`, which can be of two types, uni-directional or bi-directional, followed by the dropout cells.

The following code block creates the RNN part of the model:

```

with tf.variable_scope('rnn') as scope:

    # Reshape conv output to fit rnn input
    rnn_input = tf.reshape(conv1_drop, [params.batch_size, -1,
                                         feat_len*params.num_filters])
    # Permute into time major order for rnn
    rnn_input = tf.transpose(rnn_input, perm=[1, 0, 2])
    # Make one instance of cell on a fixed device,
    # and use copies of the weights on other devices.
    cell = rnn_cell.CustomRNNCell(
        params.num_hidden, activation=tf.nn.relu6,
        use_fp16=params.use_fp16)
    drop_cell = tf.contrib.rnn.DropoutWrapper(
        cell, output_keep_prob=params.keep_prob)
    multi_cell = tf.contrib.rnn.MultiRNNCell(
        [drop_cell] * params.num_rnn_layers)

    seq_lens = tf.div(seq_lens, params.temporal_stride)
    if params.rnn_type == 'uni-dir':
        rnn_outputs, _ = tf.nn.dynamic_rnn(multi_cell, rnn_input,
                                           sequence_length=seq_lens,
                                           dtype=dtype,
                                           time_major=True,
                                           scope='rnn',
                                           swap_memory=True)

```

```
        else:
            outputs, _ = tf.nn.bidirectional_dynamic_rnn(
                multi_cell, multi_cell, rnn_input,
                sequence_length=seq_lens, dtype=dtype,
                time_major=True, scope='rnn',
                swap_memory=True)
            outputs_fw, outputs_bw = outputs
            rnn_outputs = outputs_fw + outputs_bw
            _activation_summary(rnn_outputs)
```

Further more, the linear layer is created to perform the CTC loss function and output from the softmax layer:

```
with tf.variable_scope('softmax_linear') as scope:
    weights = _variable_with_weight_decay(
        'weights', [params.num_hidden, NUM_CLASSES],
        wd_value=None,
        use_fp16=params.use_fp16)
    biases = _variable_on_cpu('biases', [NUM_CLASSES],
        tf.constant_initializer(0.0),
        params.use_fp16)
    logit_inputs = tf.reshape(rnn_outputs, [-1, cell.output_size])
    logits = tf.add(tf.matmul(logit_inputs, weights),
                   biases, name=scope.name)
    logits = tf.reshape(logits, [-1, params.batch_size, NUM_CLASSES])
    _activation_summary(logits)
```



Production scale tip: Training a single model at these scales requires tens of exaFLOPs that would take three to six weeks to execute on a single GPU. This makes model exploration a very time-consuming exercise, so the developers of DeepSpeech have built a highly optimized training system that uses eight or 16 GPUs to train one model, as well as synchronous **stochastic gradient descent (SGD)**, which is easier to debug while testing new ideas, and also converges faster for the same degree of data parallelism.

Training the model

Now that we understand the data that we are using and the DeepSpeech model architecture, let's set up the environment to train the model. There are some preliminary steps to create a virtual environment for the project that are optional, but always recommended to use. Also, it's recommended to use GPUs to train these models.

Along with Python Version 3.5 and TensorFlow version 1.7+, the following are some of the prerequisites:

- `python-Levenshtein`: To compute **character error rate (CER)**, basically the distance
- `python_speech_features`: To extract MFCC features from raw data
- `pysoundfile`: To read FLAC files
- `scipy`: Helper functions for windowing
- `tqdm`: For displaying a progress bar

Let's create the virtual environment and install all the dependencies:

```
conda create -n 'SpeechProject' python=3.5.0
source activate SpeechProject
```

Install the following dependencies:

```
(SpeechProject)$ pip install python-Levenshtein
(SpeechProject)$ pip install python_speech_features
(SpeechProject)$ pip install pysoundfile
(SpeechProject)$ pip install scipy
(SpeechProject)$ pip install tqdm
```

Install TensorFlow with GPU support:

```
(SpeechProject)$ conda install tensorflow-gpu
```

If you see a `sndfile` error, use the following command:

```
(SpeechProject)$ sudo apt-get install libsndfile1
```

Now you will need to clone the repository that contains all the code:

```
(SpeechRecog)$ git clone https://github.com/FordSpeech/deepSpeech.git
(SpeechRecog)$ cd deepSpeech
```

Let's move the TFRecord files that we created in the *Data transformation* section. The computed MFCC features are stored inside the `data/librispeech/processed/` directory:

```
cp -r ./data/librispeech/audio /home/deepSpeech/data/librispeech
cp -r ./data/librispeech/processed /home/deepSpeech/librispeech
```

Once we have all the data files in place, it's time to train the model. We are defining four hyperparameters as num_rnn_layers set to 3, rnn_type set to bi-dir, max_steps is set to 30000, and initial_lr is set to 3e-4:

```
(SpeechRecog) $ python deepSpeech_train.py --num_rnn_layers 3 --rnn_type 'bi-dir' --initial_lr 3e-4 --max_steps 30000 --train_dir ./logs/
```

Also, if you want to resume the training using the pre-trained models from <https://drive.google.com/file/d/1E65g4H1QU666RhgY712Sn6FuU2wvZTnQ/view>, you can download and



unzip them to the logs folder:

```
(SpeechRecog) $ python deepSpeech_train.py --checkpoint_dir ./logs/ --max_steps 40000
```

Note that during the first epoch, the cost will increase and it will take longer to train on later steps because the utterances are presented in a sorted order to the network.

The following are the steps involved during the training process:

```
# Learning rate set up from the hyper-param.  
learning_rate, global_step = set_learning_rate()  
  
# Create an optimizer that performs gradient descent.  
optimizer = tf.train.AdamOptimizer(learning_rate)  
  
# Fetch a batch worth of data for each tower to train.  
data = fetch_data()  
  
# Construct loss and gradient ops.  
loss_op, tower_grads, summaries = get_loss_grads(data, optimizer)  
  
# Calculate the mean of each gradient. Note that this is the  
# synchronization point across all towers.  
grads = average_gradients(tower_grads)  
  
# Apply the gradients to adjust the shared variables.  
apply_gradient_op = optimizer.apply_gradients(grads,  
                                              global_step=global_step)  
  
# Track the moving averages of all trainable variables.  
variable_averages = tf.train.ExponentialMovingAverage(  
    ARGS.moving_avg_decay, global_step)  
variables_averages_op = variable_averages.apply(  
    tf.trainable_variables())
```

```

# Group all updates to into a single train op.
train_op = tf.group(apply_gradient_op, variables_averages_op)

# Build summary op.
summary_op = add_summaries(summaries, learning_rate, grads)

# Create a saver.
saver = tf.train.Saver(tf.all_variables(), max_to_keep=100)

# Start running operations on the Graph with allow_soft_placement set to True
# to build towers on GPU.
sess = tf.Session(config=tf.ConfigProto(
    allow_soft_placement=True,
    log_device_placement=ARGS.log_device_placement))

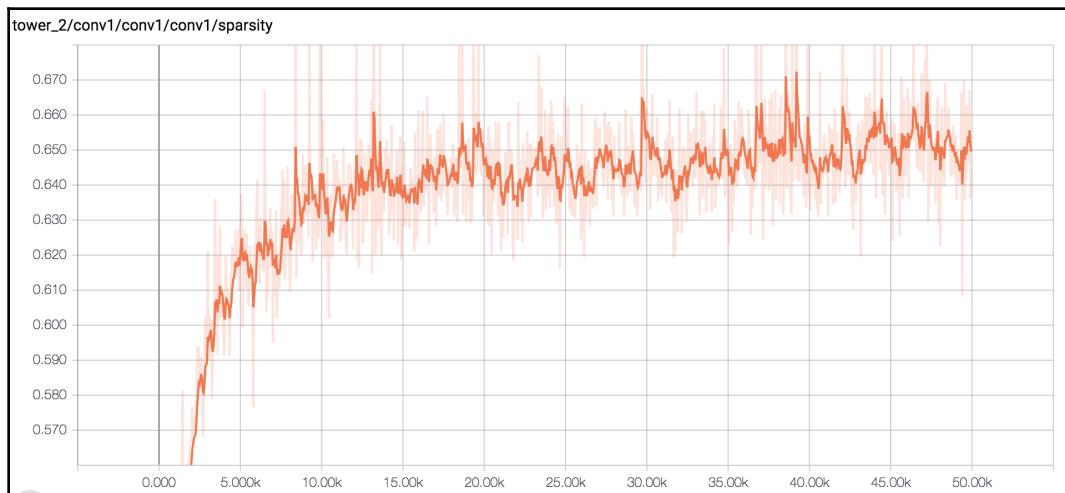
# Initialize vars depending on the checkpoints.
if ARGS.checkpoint is not None:
    global_step = initialize_from_checkpoint(sess, saver)
else:
    sess.run(tf.initialize_all_variables())

# Start the queue runners.
tf.train.start_queue_runners(sess)

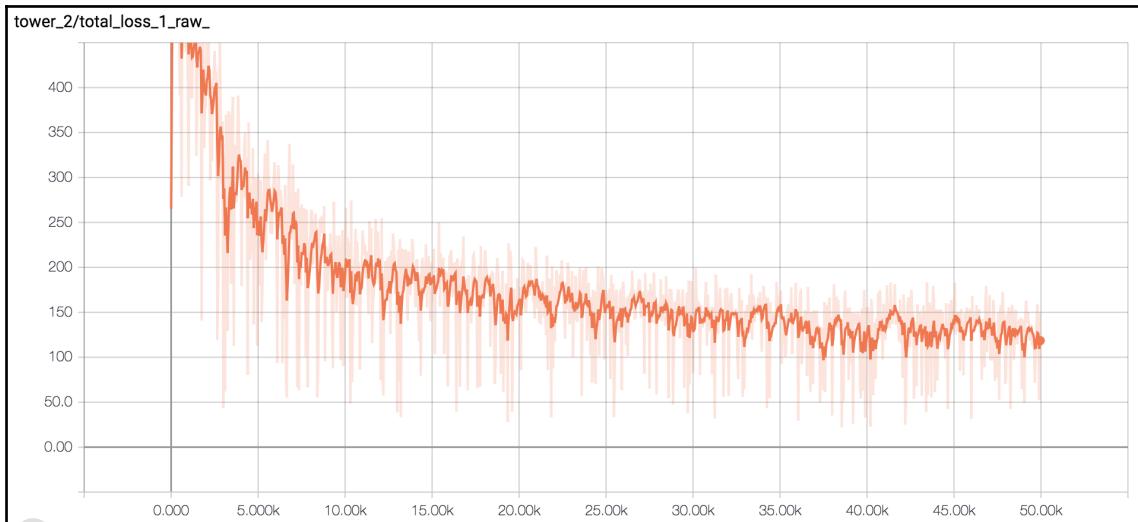
# Run training loop.
run_train_loop(sess, (train_op, loss_op, summary_op), saver)

```

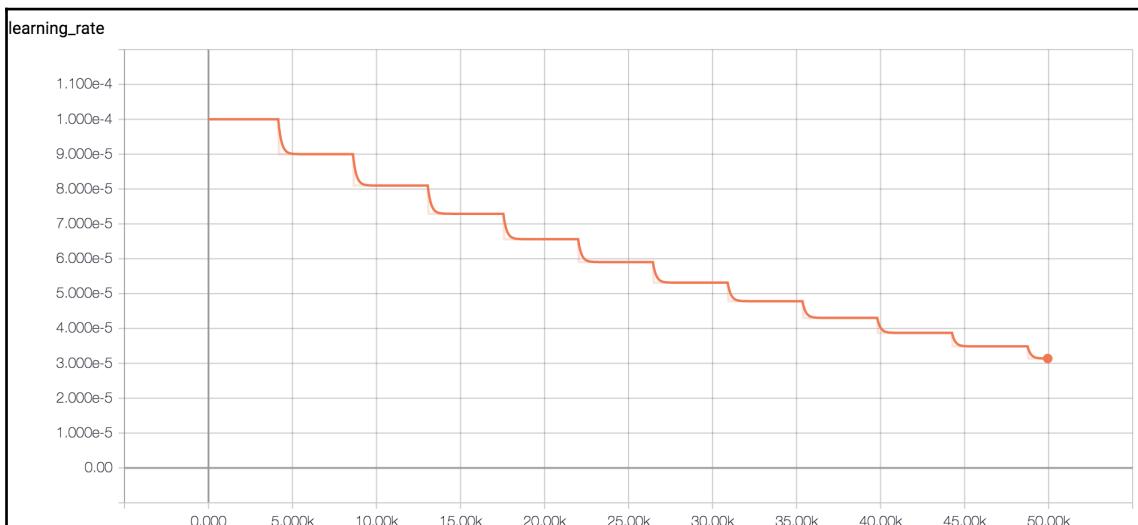
While the training process happens, we can see significant improvements, as shown in the following plots. Following graph shows the accuracy of the plot after 50k steps:



Here are the loss plots over 50k steps:



The learning rate is slowing down over the period of time:



Testing and evaluating the model

Once the model is trained, you can perform the following command to execute the test steps using the test dataset:

```
(SpeechRecog) $python deepSpeech_test.py --eval_data 'test' --checkpoint_dir  
./logs/
```

We evaluate its performance by testing it on previously unseen utterances from a test set. The model generates sequences of probability vectors as outputs, so we need to build a decoder to transform the model's output into word sequences. Despite being trained on character sequences, DS2 models are still able to learn an implicit language model and are already quite adept at spelling out words phonetically, as shown in the following table. The model's spelling performance is typically measured using CERs calculated using the Levenshtein distance (https://en.wikipedia.org/wiki/Levenshtein_distance) at the character level:

Ground truth	Model output
This had some effect in calming him	This had some offectind calming him
He went in and examined his letters but there was nothing from carrier	He went in an examined his letters but there was nothing from carry
The design was different but the thing was clearly the same	The design was differampat that thing was clarly the same



Although the model exhibit excellent CERs, their tendency to spell out words phonetically results in relatively high word-error rates. You can improve the model's performance **word-error rate (WER)** by allowing the decoder to incorporate constraints from an external lexicon and language model.

We have observed that many of the errors in the model's predictions occur in words that do not appear in the training set. It is thus reasonable to expect that the overall CER would continue to improve as we increased the size of the training set and training steps. It achieved 15% CERs after 30k steps or training.

Summary

We dove right into this deep-learning project in Python, creating and training an ASR model that understands speech data. We learned to feature engineer the speech data to extract various kinds of features from it and then build a speech recognition system that could detect a user's voice.

We're happy to have achieved our stated goal!

In this chapter, we built a system that recognizes English speech, using the DS2 model.

You learned following:

- To work with speech and spectrograms
- To build an end-to-end speech recognition system
- The CTC loss function
- Batch normalization and SortaGrad for RNNs

This caps off a major section of the deep-learning projects in this Python book that explores chatbots, NLP, and speech recognition with RNNs (uni and bi-directional, with and without LSTM components), and CNNs. We've seen the power of these technologies to provide intelligence to existing business processes and to create entirely new and smart systems. This is exciting work at the cutting edge of applied AI using deep learning! In the remaining half of the book, we'll explore deep-learning projects in Python that are generally grouped into computer vision technologies.

Let's turn the page and get started!

19

Handwritten Digits

Classification Using ConvNets

Welcome to this chapter on using **convolution neural networks (ConvNets)** for the classification of handwritten digits. In [Chapter 16, Training NN for Prediction Using Regression](#), we built a simple neural network for classifying handwritten digits. This was 87% accurate, but we were not happy with its performance. In this chapter, we will understand what convolution is and build a ConvNet for classifying the handwritten digits to help the restaurant chain become more accurate in sending text messages to the right person. If you have not been through [Chapter 16, Training NN for Prediction Using Regression](#), please go through it once so that you can get an understanding of the use case.

The following topics will be covered in this chapter:

- Convolution
- Pooling
- Dropout
- Training the model
- Testing the model
- Building deeper models

It would be better if you implement the code snippets as you go through this chapter, either in a Jupyter Notebook or any source code editor. This will make it easier for you to follow along as well as understand how the different sections of the code work.

Code implementation

In this exercise, we will be using the Keras deep learning library, which is a high-level neural network API capable of running on top of TensorFlow, Theano, and CNTK.



Know the code! We will not spend time understanding how Keras works, but if you are interested, refer to this easy-to-understand official documentation from Keras at <https://keras.io/>.

Importing all of the dependencies

We will be using the numpy, matplotlib, keras, scipy, and tensorflow packages in this exercise. Here, TensorFlow is used as the backend for Keras. You can install these packages with pip. For the MNIST data, we will be using the dataset available in the keras module with a simple import:

```
import numpy as np
```

It is important that you set seed for reproducibility:

```
# set seed for reproducibility
seed_val = 9000
np.random.seed(seed_val)
```

Exploring the data

Let's import the mnist module that's available in keras with the following code:

```
from keras.datasets import mnist
```

Then, unpack the mnist train and test images with the following code:

```
# unpack mnist data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Now that the data has been imported, let's explore these digits:

```
print('Size of the training_set: ', X_train.shape)
print('Size of the test_set: ', X_test.shape)
print('Shape of each image: ', X_train[0].shape)
print('Total number of classes: ', len(np.unique(y_train)))
print('Unique class labels: ', np.unique(y_train))
```

The following is the output of the preceding code:

```
('Size of the training_set: ', (60000, 28, 28))
('Size of the test_set: ', (10000, 28, 28))
('Shape of each image: ', (28, 28))
('Total number of classes: ', 10)
('Unique class labels: ', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8))
```

Figure 8.1: Printout information of the data

From the preceding screenshot, we can see that we have 60000 train images, 10000 test images with each image being 28*28 in size, and a total of 10 predictable classes.

Now, let's plot 9 handwritten digits. Before that, we will need to import `matplotlib` for plotting:

```
import matplotlib.pyplot as plt
# Plot of 9 random images
for i in range(0, 9):
    plt.subplot(331+i) # plot of 3 rows and 3 columns
    plt.axis('off') # turn off axis
    plt.imshow(X_train[i], cmap='gray') # gray scale
```

The following is the output of the preceding code:

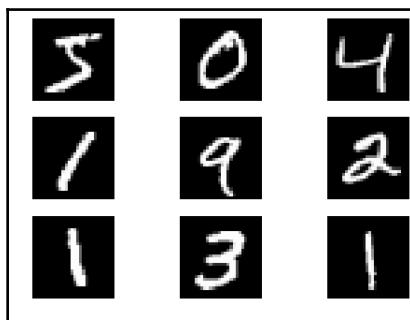


Figure 8.2: Visualizing MNIST digits

Print out the maximum and minimum pixel value of the pixels in the `training_set`:

```
# maximum and minimum pixel values
print('Maximum pixel value in the training_set: ', np.max(X_train))
print('Minimum pixel value in the training_set: ', np.min(X_train))
```

The following is the output of the preceding code:

```
('Maximum pixel value in the training_set: ', 255)
('Minimum pixel value in the training_set: ', 0)
```

Figure 8.3: Printout of the maximum and minimum pixel value in the data

We can see that the maximum and minimum pixel values in the training set are 255 and 0.

Defining the hyperparameters

The following are some of the hyperparameters that we will be using throughout our code. These are totally configurable:

```
# Number of epochs
epochs = 20

# Batchsize
batch_size = 128

# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.0001)

# Shape of the input image
input_shape = (28, 28, 1)
```

If you look back at Chapter 16, *Training NN for Prediction Using Regression*, you'll see that the optimizer used there was Adam. Therefore, we will import the Adam optimizer from the keras module and set its learning rate, as shown in the preceding code. For most cases that will follow, we will be training for 20 epochs for ease of comparison.



To learn more about the optimizers and their APIs in Keras, visit <https://keras.io/optimizers/>.



Experiment with different learning rates, optimizers, and batch sizes to see how these factors affect the quality of your model. If you get better results, show this to the deep learning community.

Building and training a simple deep neural network

Now that we have loaded the data into memory, we need to build a simple neural network model to predict the MNIST digits. We will use the same architecture we used in Chapter 16, *Training NN for Prediction Using Regression*.

We will be building a `Sequential` model. So, let's import it from Keras and initialize it with the following code:

```
from keras.models import Sequential  
model = Sequential()
```



To learn more about the Keras Model API, visit <https://keras.io/models/model/>.

The next thing that we need to do is define the `Dense`/Perceptron layer. In Keras, this can be done by importing the `Dense` layer, as follows:

```
from keras.layers import Dense
```

Then, we need to add the `Dense` layer to the `Sequential` model as follows:

```
model.add(Dense(300, input_shape=(784,), activation = 'relu'))
```



To learn more about the Keras Dense API call, visit <https://keras.io/layers/core/>.

The `add` command performs the job of appending a layer to the `Sequential` model, in this case, `Dense`.

In the `Dense` layer in the preceding code, we have defined the number of neurons in the first hidden layer, which is 300. We have also defined the `input_shape` parameter as being equal to `(784,)` to indicate to the model that it will be accepting input arrays of the shape `(784,)`. This means that the input layer will have 784 neurons.

The type of activation function that needs to be applied to the result can be defined with the `activation` parameter. In this case, this is `relu`.

Add another Dense layer of 300 neurons by using the following code:

```
model.add(Dense(300, activation='relu'))
```

And the final Dense layer with the following code:

```
model.add(Dense(10, activation='softmax'))
```

Here, the final layer has 10 neurons as we need it to predict scores for 10 classes. The activation function that has been chosen here is `softmax` so that we can limit the scores between 0 and 1, and the sum of scores to 1.

Compiling the model in Keras is super-easy and can be done with following code:

```
# compile the model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer
, metrics = ['accuracy'])
```

All you need to do to compile the model is call the `compile` method of the model and specify the `loss`, `optimizer`, and `metrics` parameters, which in this case are `sparse_categorical_crossentropy`, `Adam`, and `['accuracy']`.

To learn more about the Keras Model's `compile` method, visit <https://keras.io/models/model/>.



The metrics that need to be monitored during this learning process must be specified as a list to the `metrics` parameter of the `compile` method.

Print out the summary of the model with the following code:

```
# print model summary
model.summary()
```

The following is the output of the preceding code:

Layer (type)	Output Shape	Param #
<hr/>		
dense_1 (Dense)	(None, 300)	235500
dense_2 (Dense)	(None, 300)	90300
dense_3 (Dense)	(None, 10)	3010
<hr/>		
Total params:	328,810	
Trainable params:	328,810	
Non-trainable params:	0	

Figure 8.4: Summary of the multilayer Perceptron model

Notice that this model has 328,810 trainable parameters, which is reasonable.

Now, split the train data into train and validation data by using the `train_test_split` function that we imported from `sklearn`:

```
from sklearn.model_selection import train_test_split

# create train and validation data
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
stratify = y_train, test_size = 0.08333, random_state=42)

X_train = X_train.reshape(-1, 784)
X_val = X_val.reshape(-1, 784)
X_test = X_test.reshape(-1, 784)

print('Training Examples', X_train.shape[0])
print('Validation Examples', X_val.shape[0])
print('Test Examples', X_test.shape[0])
```

We have split the data so that we end up with 55,000 training examples and 5,000 validation examples.

You will also see that we have reshaped the arrays so that each image is of shape `(784,)`. This is because we have defined the model to accept images/arrays of shape `(784,)`.

Like we did in Chapter 16, *Training NN for Prediction Using Regression*, we will now train our model on 55,000 training examples, validate on 5,000 examples, and test on 10,000 examples.

Assigning the fit to a variable stores relevant information inside it, such as train and validation loss and accuracy at each epoch, which can then be used for plotting the learning process.

Fitting a model

To fit a model in Keras, along with train digits and train labels, call the `fit` method of the model with the following parameters:

- `epochs`: The number of epochs
- `batch_size`: The number of images in each batch
- `validation_data`: The tuple of validation images and validation labels

Look at the *Defining the hyperparameters* section of the chapter for the defined values of `epochs` and `batch_size`:

```
# fit the model
history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size, validation_data=(X_val, y_val))
```

The following is the output of the preceding code:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/20
55000/55000 [=====] - 4s 66us/step - loss: 9.1007 - acc: 0.4256 - val_loss: 7.3798 - val_acc: 0.5342
Epoch 2/20
55000/55000 [=====] - 3s 54us/step - loss: 6.4443 - acc: 0.5922 - val_loss: 5.3994 - val_acc: 0.6572
Epoch 3/20
55000/55000 [=====] - 3s 54us/step - loss: 5.3398 - acc: 0.6625 - val_loss: 5.3888 - val_acc: 0.6578
Epoch 4/20
55000/55000 [=====] - 3s 55us/step - loss: 5.1974 - acc: 0.6723 - val_loss: 5.3200 - val_acc: 0.6624
```

The following is the output at the end of the code's execution:

```
Epoch 19/20
55000/55000 [=====] - 4s 69us/step - loss: 1.7686 - acc: 0.8871 - val_loss: 2.0171 - val_acc: 0.8696
Epoch 20/20
55000/55000 [=====] - 4s 67us/step - loss: 1.7533 - acc: 0.8889 - val_loss: 1.9683 - val_acc: 0.8728
```

Figure 8.5: Metrics printed out during the training of MLP

Evaluating a model

To evaluate the model on test data, you can call the `evaluate` method of the `model` by feeding the test images and test labels:

```
# evaluate the model
loss, acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

The following is the output of the preceding code:

```
10000/10000 [=====] - 1s 95us/step
Test loss: 1.9103740268349634
Accuracy: 0.8764
```

Figure 8.6: Printout of the evaluation of MLP

From the validation and test accuracy, we can see that after 20 epochs of training, we have reached the same level of accuracy as we did in [Chapter 16, Training NN for Prediction Using Regression](#), but with very few lines of code.

Now, let's define a function to plot the train and validation loss and accuracy that we have stored in the `history` variable:

```
import matplotlib.pyplot as plt

def loss_plot(history):
    train_acc = history.history['acc']
    val_acc = history.history['val_acc']

    plt.figure(figsize=(9,5))
    plt.plot(np.arange(1,21),train_acc, marker = 'D', label = 'Training Accuracy')
    plt.plot(np.arange(1,21),val_acc, marker = 'o', label = 'Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Train/Validation Accuracy')
    plt.legend()
    plt.margins(0.02)
    plt.show()

    train_loss = history.history['loss']
    val_loss = history.history['val_loss']
```

```
plt.figure(figsize=(9,5))
plt.plot(np.arange(1,21),train_loss, marker = 'D', label = 'Training Loss')
plt.plot(np.arange(1,21),val_loss, marker = 'o', label = 'Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train/Validation Loss')
plt.legend()
plt.margins(0.02)
plt.show()
# plot training loss
loss_plot(history)
```

The following is the output of the preceding code:

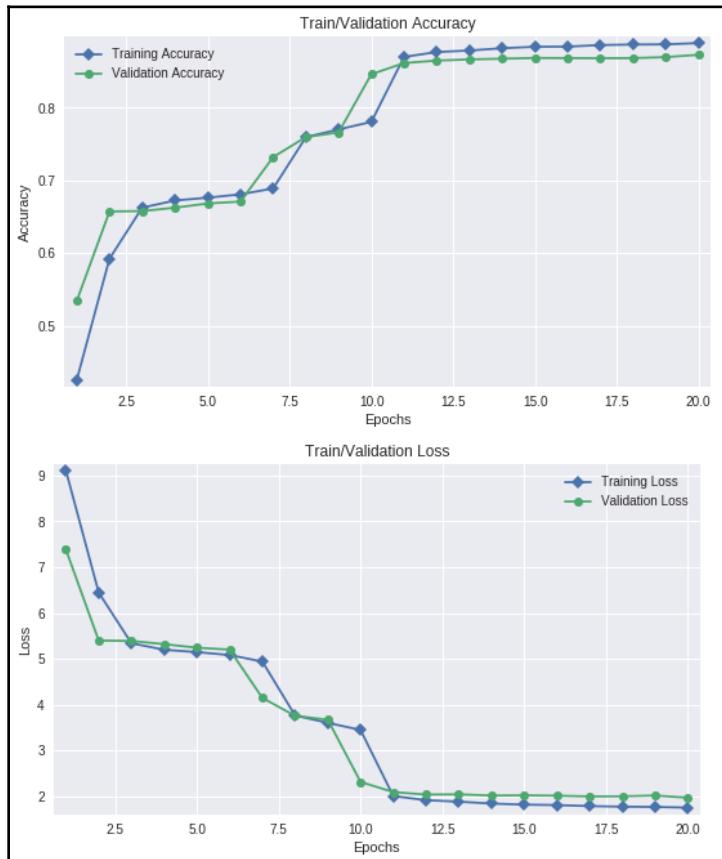


Figure 8.7: MLP loss/accuracy plot during training

MLP – Python file

This module implements training and evaluation of a simple MLP:

```
"""This module implements a simple multi layer perceptron in keras."""
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from loss_plot import loss_plot

# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.0001)
# Shape of the input image
input_shape = (28,28,1)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                stratify = y_train,
                                                test_size = 0.08333,
                                                random_state=42)

X_train = X_train.reshape(-1, 784)
X_val = X_val.reshape(-1, 784)
X_test = X_test.reshape(-1, 784)

model = Sequential()
model.add(Dense(300, input_shape=(784,), activation = 'relu'))
model.add(Dense(300, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss = 'sparse_categorical_crossentropy',
optimizer=optimizer,
metrics = ['accuracy'])

history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size,
validation_data=(X_val, y_val))

loss,acc = model.evaluate(X_test, y_test)
```

```
print('Test loss:', loss)
print('Accuracy:', acc)

loss_plot(history)
```

Convolution

Convolution can be defined as the process of striding a small kernel/filter/array over a target array and obtaining the sum of element-wise multiplication between the kernel and a subset of equal size of the target array at that location.

Consider the following example:

```
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
kernel = np.array([-1, 1, 0])
```

Here, you have a target array of length 10 and a kernel of length 3.

When you start the convolution, implement the following steps:

1. The kernel will be multiplied with the subset of the target array within indices 0 through 2. This will be between [-1,1,0] (kernel) and [0,1,0] (from index 0 through to 2 of the target array). The result of this element-wise multiplication will then be summed up to obtain what is called the result of convolution.
2. The kernel will then be stridden by 1 unit and then multiplied with the subset of the target array within the indices 1 through 3, just like in Step 1, and the result is obtained.
3. Step 2 is repeated until a subset equal to the length of the kernel is not possible at a new stride location.

The result of convolution at each stride is stored in an array. This array that's holding the result of the convolution is called the feature map. The length of the 1-D feature map (with step/stride of 1) is equal to the difference in length of the kernel and the target array plus 1.

Only in this case, we need to take the following equation into account:

$$\text{length of the feature map} = \text{length of the target array} - \text{length of the kernel} + 1$$

Here is a code snippet implementing 1-D convolution:

```
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
kernel = np.array([-1, 1, 0])

# empty feature map
conv_result = np.zeros(array.shape[0] - kernel.shape[0] + 1).astype(int)

for i in range(array.shape[0] - kernel.shape[0] + 1):
    # convolving
    conv_result[i] = (kernel * array[i:i+3]).sum()
    print(kernel, '*', array[i:i+3], '=', conv_result[i])
print('Feature Map :', conv_result)
```

The following is the output of the preceding code:

[-1 1 0] * [0 1 0] = 1
[-1 1 0] * [1 0 1] = -1
[-1 1 0] * [0 1 0] = 1
[-1 1 0] * [1 0 1] = -1
[-1 1 0] * [0 1 0] = 1
[-1 1 0] * [1 0 1] = -1
[-1 1 0] * [0 1 0] = 1
[-1 1 0] * [1 0 1] = -1
[-1 1 0] * [0 1 0] = 1
[-1 1 0] * [1 0 1] = -1
Feature Map : [1 -1 1 -1 1 -1 1 -1]

Figure 8.8: Printout of example feature map

Convolution in Keras

Now that you have an understanding of how convolution works, let's put it into use and build a CNN classifier on MNIST digits.

For this, import the `Conv2D` API from the `layers` module of Keras. You can do this with the following code:

```
from keras.layers import Conv2D
```

Since the convolution will be defined to accept images of shape $28 \times 28 \times 1$, we need to reshape all the images to be of $28 \times 28 \times 1$:

```
# reshape data
X_train = X_train.reshape(-1, 28, 28, 1)
X_val = X_val.reshape(-1, 28, 28, 1)
```

```
X_test = X_test.reshape(-1, 28, 28, 1)

print('Train data shape:', X_train.shape)
print('Val data shape:', X_val.shape)
print('Test data shape:', X_test.shape)
```

The following is the output of the preceding code:

```
Train data shape: (55000, 28, 28, 1)
Val data shape: (5000, 28, 28, 1)
Test data shape: (10000, 28, 28, 1)
```

Figure 8.9: Shape of data after reshaping

To build the model, just like we did previously, we need to initialize the model as Sequential:

```
model = Sequential()
```

Now, add the Conv2D layer to the model with the following code:

```
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=input_shape, activation = 'relu'))
```

In the Conv2D API, we have defined the following parameters:

- units: 32 (number of kernels/filters)
- kernel_size: (3, 3) (size of each kernel)
- input_shape: 28*28*1 (shape of the input array it will receive)
- activation: relu



For additional information on the Conv2D API, visit <https://keras.io/layers/convolutional/>.

The result of the preceding convolution is 32 feature maps of size 26*26. These 2-D feature maps now have to be converted into a 1-D feature map. This can be done in Keras with the following code:

```
from keras.layers import Flatten
model.add(Flatten())
```

The result of the preceding snippet is just like a layer of neurons in a simple neural network. The `Flatten` function converts all of the 2-D feature maps into a single `Dense` layer. In this layer, will we add a `Dense` layer with 128 neurons:

```
model.add(Dense(128, activation = 'relu'))
```

Since we need to get scores for each of the 10 possible classes, we must add another `Dense` layer with 10 neurons, with `softmax` as the activation function:

```
model.add(Dense(10, activation = 'softmax'))
```

Now, just like in the case of the simple dense neural network we built in the preceding code, we will compile and fit the model:

```
# compile model
model.compile(loss = 'sparse_categorical_crossentropy',
optimizer=optimizer, metrics = ['accuracy'])

# print model summary
model.summary()
```

The following is the output of the preceding code:

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_10 (Conv2D)	(None, 26, 26, 32)	320
<hr/>		
flatten_6 (Flatten)	(None, 21632)	0
<hr/>		
dense_36 (Dense)	(None, 128)	2769024
<hr/>		
dense_37 (Dense)	(None, 10)	1290
<hr/>		
Total params: 2,770,634		
Trainable params: 2,770,634		
Non-trainable params: 0		

Figure 8.10: Summary of the convolution classifier

From the model's summary, we can see that this convolution classifier has 2,770,634 parameters. This is a lot of parameters compared to the Perceptron model. Let's fit this model and evaluate its performance.

Fitting the model

Fit the convolution neural network model on the data with the following code:

```
# fit model
history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size, validation_data=(X_val, y_val))
```

The following is the output of the preceding code:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/20
55000/55000 [=====] - 7s 124us/step - loss: 2.8752 - acc: 0.7995 - val_loss: 1.9732 - val_acc: 0.8614
Epoch 2/20
55000/55000 [=====] - 6s 105us/step - loss: 1.2099 - acc: 0.8994 - val_loss: 0.3333 - val_acc: 0.9404
Epoch 3/20
55000/55000 [=====] - 6s 105us/step - loss: 0.1757 - acc: 0.9643 - val_loss: 0.1891 - val_acc: 0.9592
```

The following is the output from the end of the code's execution:

```
Epoch 19/20
55000/55000 [=====] - 6s 104us/step - loss: 0.0170 - acc: 0.9964 - val_loss: 0.1326 - val_acc: 0.9786
Epoch 20/20
55000/55000 [=====] - 6s 108us/step - loss: 0.0132 - acc: 0.9971 - val_loss: 0.1381 - val_acc: 0.9772
```

Figure 8.11: Metrics printed out during the training of the convolution classifier

We can see that the convolution classifier's accuracy is 97.72% on the validation data.

Evaluating the model

You can evaluate the convolution model on the test data with the following code:

```
# evaluate model
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

The following is the output of the preceding code:

```
10000/10000 [=====] - 2s 175us/step
Test loss: 0.1275109763626777
Accuracy: 0.9792
```

Figure 8.12: Printout of the evaluation of the convolution classifier

We can see that the model is 97.92% accurate on test data, 97.72% on validation data, and 99.71% on train data. It is clear from the loss as well that the model is slightly overfitting on the train data. We will talk about how to handle overfitting later.

Now, let's plot the train and validation metrics to see how the training has progressed:

```
# plot training loss
loss_plot(history)
```

The following is the output of the preceding code:

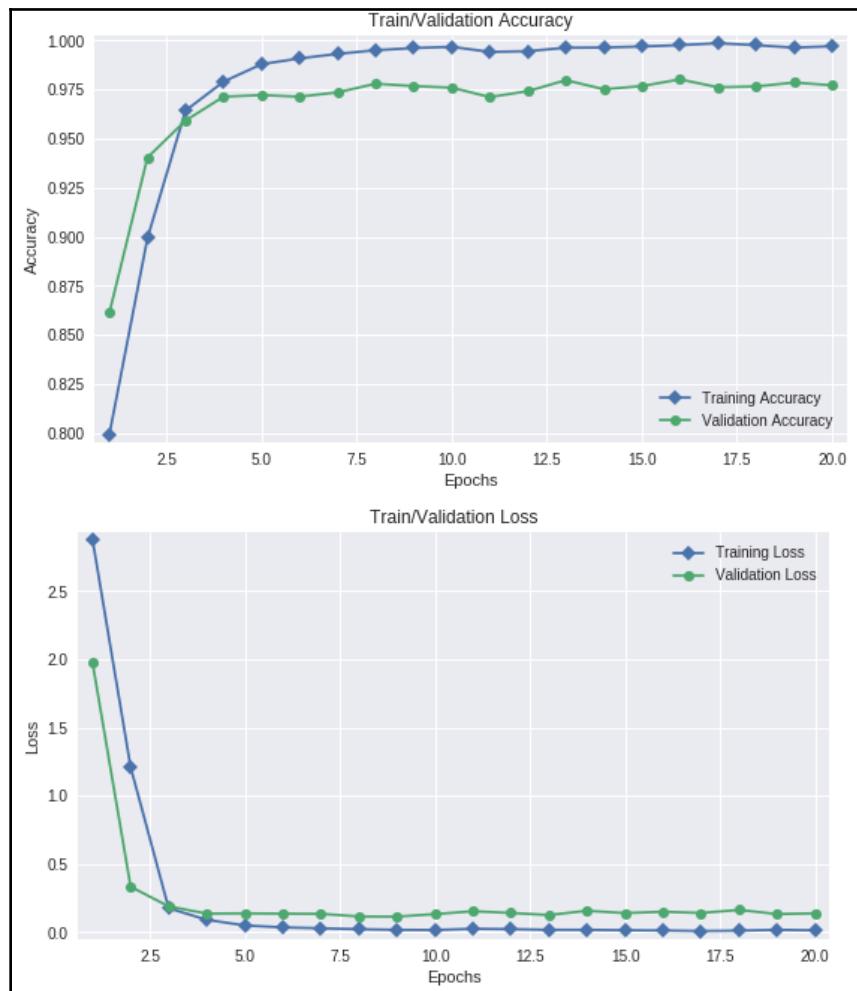


Figure 8.13: Loss/accuracy plot of the convolution classifier during training

Convolution – Python file

This module implements the training and evaluation of a convolution classifier:

```
"""This module implements a simple convolution classifier."""
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from loss_plot import loss_plot

# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.0001)
# Shape of the input image
input_shape = (28,28,1)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                stratify = y_train,
                                                test_size = 0.08333,
                                                random_state=42)

X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape,
                activation = 'relu'))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss = 'sparse_categorical_crossentropy',
optimizer=optimizer,
metrics = ['accuracy'])

history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size,
validation_data=(X_val, y_val))
```

```
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)

loss_plot(history)
```

Pooling

Max pooling can be defined as the process of summarizing a group of values with the maximum value within that group. Similarly, if you computed the average, it would be average pooling. Pooling operations are usually performed on the generated feature maps after convolution to reduce the number of parameters.

Let's take the example array we considered for convolution:

```
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
```

Now, if you were to perform max pooling on this `array` with the pool size set to size 1×2 and a stride of 2, the result would be an array of [1,1,1,1]. The `array` of size 1×10 has been reduced to a size of 1×5 due to max pooling.

Here, since the pool size is of shape 1×2 , you would take the subset of the target `array` from index 0 to index 2, which will be [0,1], and compute the maximum of this subset as 1. You would do the same for the subset from index 2 to index 4, from index 4 to index 6, index 6 to index 8, and finally index 8 to 10.

Similarly, average pooling can be implemented by computing the average value of the pooled section. In this case, it would result in the array [0.5, 0.5, 0.5, 0.5, 0.5].

The following are a couple of code snippets that are implementing max and average pooling:

```
# 1D Max Pooling
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
result = np.zeros(len(array)//2)
for i in range(len(array)//2):
    result[i] = np.max(array[2*i:2*i+2])
result
```

The following is the output of the preceding code:

```
array([1., 1., 1., 1., 1.])
```

Figure 8.14: Max pooling operation's result on an array

The following is the code snippet for average pooling:

```
# 1D Average Pooling
array = np.array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1])
result = np.zeros(len(array)//2)
for i in range(len(array)//2):
    result[i] = np.mean(array[2*i:2*i+2])
result
```

The following is the output of the preceding code:

```
array([0.5, 0.5, 0.5, 0.5, 0.5])
```

Figure 8.15: Average pooling operation's result on an array

The following is a diagram explaining the max pooling operation:

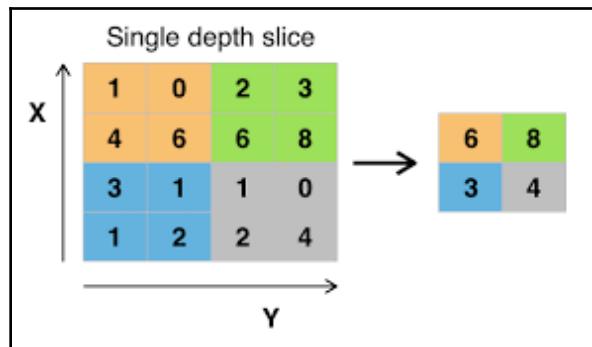


Figure 8.16: 2×2 max pooling with stride 2 (Source: https://en.wikipedia.org/wiki/Convolutional_neural_network)

Consider the following code for a digit:

```
plt.imshow(X_train[0].reshape(28,28), cmap='gray')
```

The following is the output of the preceding code:

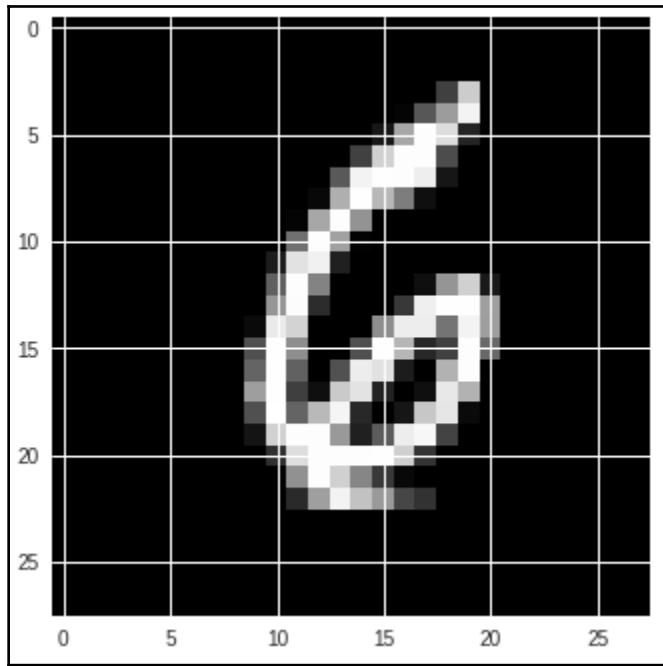


Figure 8.17: Random MNIST digit

This image is of shape 28*28. Now, if you were to perform a 2*2 max pooling operation of this, the resulting image would have a shape of 14*14.

Now, let's write a function to implement a 2*2 max pooling operation on a MNIST digit:

```
def square_max_pool(image, pool_size=2):
    result = np.zeros((14,14))
    for i in range(result.shape[0]):
        for j in range(result.shape[1]):
            result[i,j] = np.max(image[i*pool_size : i*pool_size+pool_size,
j*pool_size : j*pool_size+pool_size])
    return result

# plot a pooled image
plt.imshow(square_max_pool(X_train[0].reshape(28,28)), cmap='gray')
```

The following is the output of the preceding code:

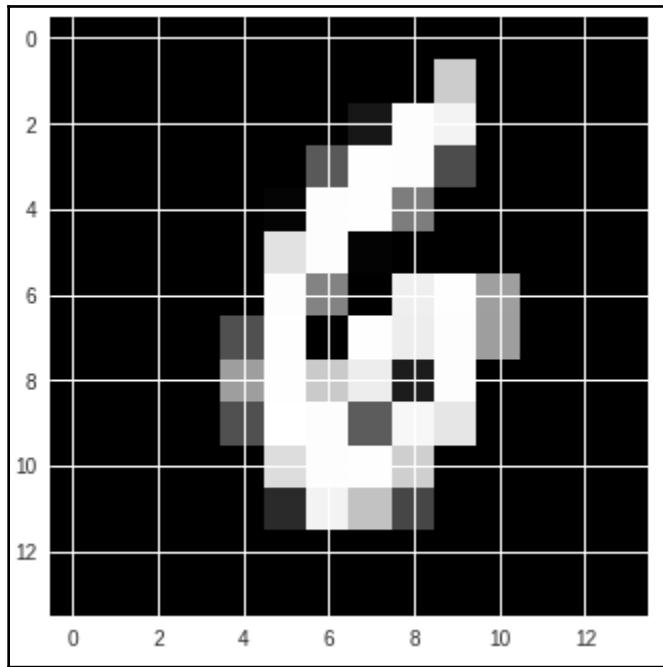


Figure 8.18: Random MNIST digit after max pooling

You may have noticed that the convolution classifier that we built in the previous section has around 2.7 million parameters. It has been proven that having a lot of parameters can lead to overfitting in a lot of cases. This is where pooling comes in. It helps us to retain the important features in the data as well as reduce the number of parameters.

Now, let's implement a convolution classifier with max pooling.

Import the max pool operation from Keras with the following code:

```
from keras.layers import MaxPool2D
```

Then, define and compile the model:

```
# model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=input_shape, activation = 'relu'))
model.add(MaxPool2D(2, 2))
model.add(Dropout(0.2))
```

```

model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))

# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=
optimizer, metrics = ['accuracy'])

# print model summary
model.summary()

```

The following is the output of the preceding code:

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_12 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_10 (MaxPooling)	(None, 13, 13, 32)	0
flatten_8 (Flatten)	(None, 5408)	0
dense_43 (Dense)	(None, 128)	692352
dense_44 (Dense)	(None, 10)	1290
<hr/>		
Total params:	693,962	
Trainable params:	693,962	
Non-trainable params:	0	

Figure 8.19: Summary of the convolution classifier with max pooling

From the summary, we can see that with a pooling filter of 2*2 with stride 2, the number of parameters has come down to 693,962, which is 1/4th of the number of parameters in the convolution classifier.

Fitting the model

Now, let's fit the model on the data:

```

# fit model
history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size, validation_data=(X_val, y_val))

```

The following is the output of the preceding code:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/20
55000/55000 [=====] - 6s 117us/step - loss: 1.5566 - acc: 0.8555 - val_loss: 0.4041 - val_acc: 0.9416
Epoch 2/20
55000/55000 [=====] - 5s 97us/step - loss: 0.2344 - acc: 0.9603 - val_loss: 0.2542 - val_acc: 0.9596
Epoch 3/20
55000/55000 [=====] - 5s 95us/step - loss: 0.1293 - acc: 0.9742 - val_loss: 0.2171 - val_acc: 0.9634
```

The following is the output at the end of the code's execution:

```
Epoch 19/20
55000/55000 [=====] - 5s 91us/step - loss: 0.0092 - acc: 0.9983 - val_loss: 0.1411 - val_acc: 0.9788
Epoch 20/20
55000/55000 [=====] - 5s 90us/step - loss: 0.0113 - acc: 0.9974 - val_loss: 0.1630 - val_acc: 0.9772
```

Figure 8.20: Metrics printed out during the training of the convolution classifier with max pooling

We can see that the convolution classifier with max pooling has an accuracy of 97.72% on the validation data.

Evaluating the model

Now, evaluate the convolution model with max pooling on the test data:

```
# evaluate model
loss, acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

The following is the output of the preceding code:

```
10000/10000 [=====] - 1s 111us/step
Test loss: 0.13552795079282295
Accuracy: 0.9788
```

Figure 8.21: Printout of the evaluation of the convolution classifier with max pooling

We can see that the model is 97.88% accurate on the test data, 97.72% on the validation data, and 99.74% on the train data. The convolution model with pooling gives the same level of performance as the convolution model without pooling, but with four times less parameters.

In this case, we can clearly see from the loss that the model is slightly overfitting on the train data.

Just like we did previously, plot the train and validation metrics to see how the training has progressed:

```
# plot training loss
loss_plot(history)
```

The following is the output of the preceding code:

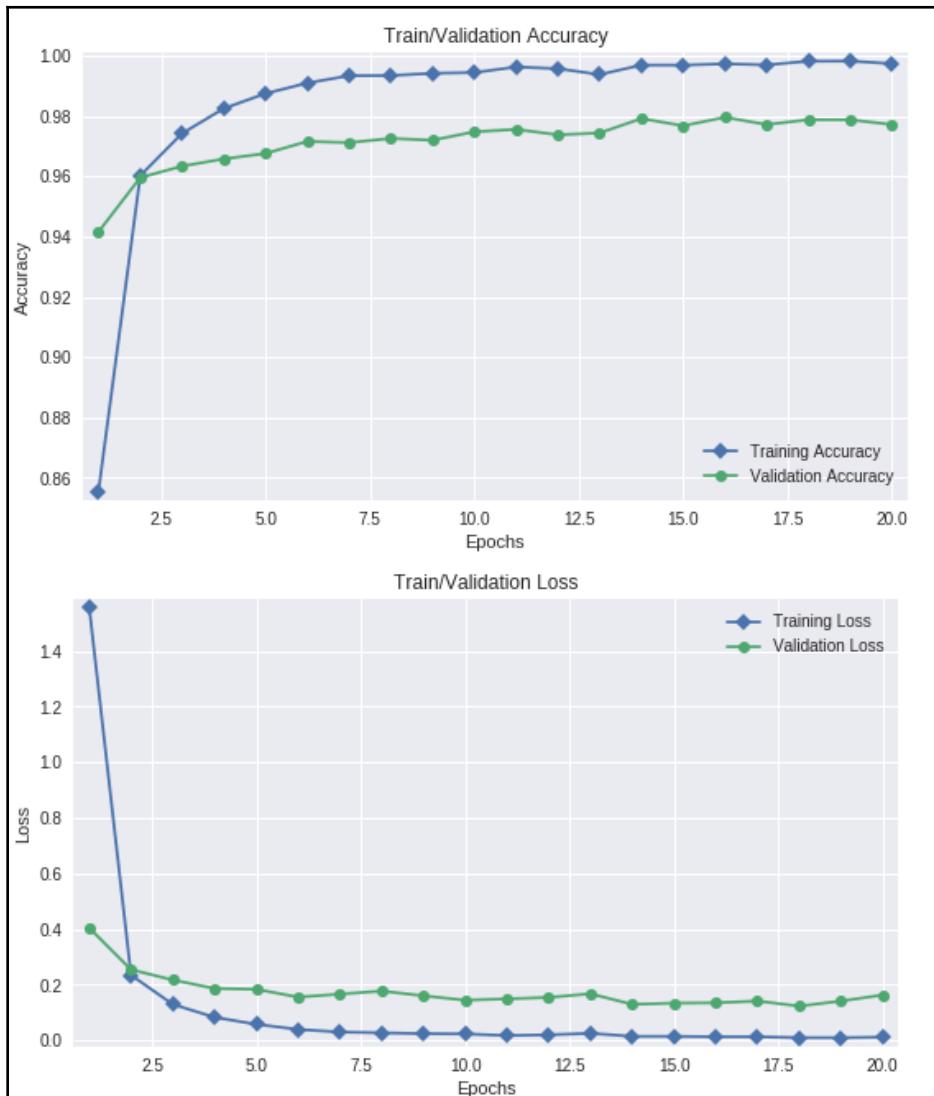


Figure 8.22: Loss/accuracy plot of the convolution classifier with max pooling during training

Convolution with pooling – Python file

This module implements the training and evaluation of a convolution classifier with the pooling operation:

```
"""This module implements a convolution classifier with maxpool
operation."""
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPool2D
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from loss_plot import loss_plot

# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.0001)
# Shape of the input image
input_shape = (28, 28, 1)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                stratify = y_train,
                                                test_size = 0.08333,
                                                random_state=42)

X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape,
                activation='relu'))
model.add(MaxPool2D(2,2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss = 'sparse_categorical_crossentropy',
optimizer=optimizer,
metrics = ['accuracy'])
```

```
history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size,
validation_data=(X_val, y_val))

loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)

loss_plot(history)
```

Dropout

Dropout is a regularization technique used to prevent overfitting. During training, it is implemented by randomly sampling a neural network from the original neural network during each forward and backward propagation, and then training this subset network on the batch of input data. During testing, no dropout is implemented. The test results are obtained as an ensemble of all of the sampled networks:

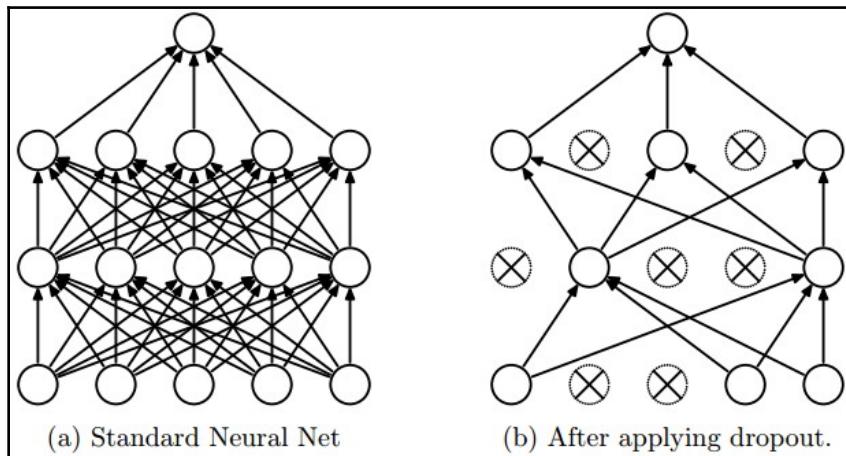


Figure 8.23: Dropout, as shown in the Dropout: A Simple Way to Prevent Neural Networks from Overfitting paper (Source: <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>)

In Keras, implementing Dropout is easy. First, import it from the `layers` module of `keras`:

```
from keras.layers import Dropout
```

Then, place the layer where needed. In the case of our CNN, we will place one after the max pool operation and one after the Dense layer, as shown in the following code:

```
# model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape, activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax')))

# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer, metrics = ['accuracy'])

# model summary
model.summary()
```

The following is the output of the preceding code:

Layer (type)	Output Shape	Param #
conv2d_13 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_11 (MaxPooling)	(None, 13, 13, 32)	0
dropout_11 (Dropout)	(None, 13, 13, 32)	0
flatten_9 (Flatten)	(None, 5408)	0
dense_45 (Dense)	(None, 128)	692352
dropout_12 (Dropout)	(None, 128)	0
dense_46 (Dense)	(None, 10)	1290
<hr/>		
Total params:	693,962	
Trainable params:	693,962	
Non-trainable params:	0	

Figure 8.24: Summary of the convolution classifier

Since Dropout is a regularization technique, adding it to a model will not result in a change in the number of trainable parameters.

Fitting the model

Again, train the model on the standard 20 epochs:

```
# fit model
history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size, validation_data=(X_val, y_val))
```

The following is the output of the preceding code:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/20
55000/55000 [=====] - 7s 128us/step - loss: 1.8460 - acc: 0.7737 - val_loss: 0.3178 - val_acc: 0.9122
Epoch 2/20
55000/55000 [=====] - 6s 107us/step - loss: 0.4344 - acc: 0.8889 - val_loss: 0.1863 - val_acc: 0.9468
Epoch 3/20
55000/55000 [=====] - 6s 108us/step - loss: 0.2815 - acc: 0.9251 - val_loss: 0.1443 - val_acc: 0.9584
```

The following is the output at the end of the code's execution:

```
Epoch 19/20
55000/55000 [=====] - 6s 105us/step - loss: 0.0255 - acc: 0.9916 - val_loss: 0.0671 - val_acc: 0.9860
Epoch 20/20
55000/55000 [=====] - 6s 102us/step - loss: 0.0228 - acc: 0.9926 - val_loss: 0.0677 - val_acc: 0.9852
```

Figure 8.25: Metrics printed out during the training of the convolution classifier with max pooling and dropout

We see that the convolution classifier with max pooling and dropout is 98.52% accurate on the validation data.

Evaluating the model

Now, let's evaluate the model and capture the loss and the accuracy:

```
# evaluate model
loss, acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

The following is the output of the preceding code:

```
10000/10000 [=====] - 1s 128us/step
Test loss: 0.06125994629900379
Accuracy: 0.9842
```

Figure 8.26: Printout of the evaluation of the convolution classifier with max pooling and dropout

We can see that the model is 98.42% accurate on the test data, 98.52% on the validation data, and 99.26% on the train data. The convolution model with pooling and dropout gives the same level of performance as the convolution model without pooling, but with four times fewer parameters. If you look at the `loss` as well, this model was able to reach a better minima than the other models we have trained before.

Plot the metrics to understand how the training has progressed:

```
# plot training loss
loss_plot(history)
```

The following is the output of the preceding code:

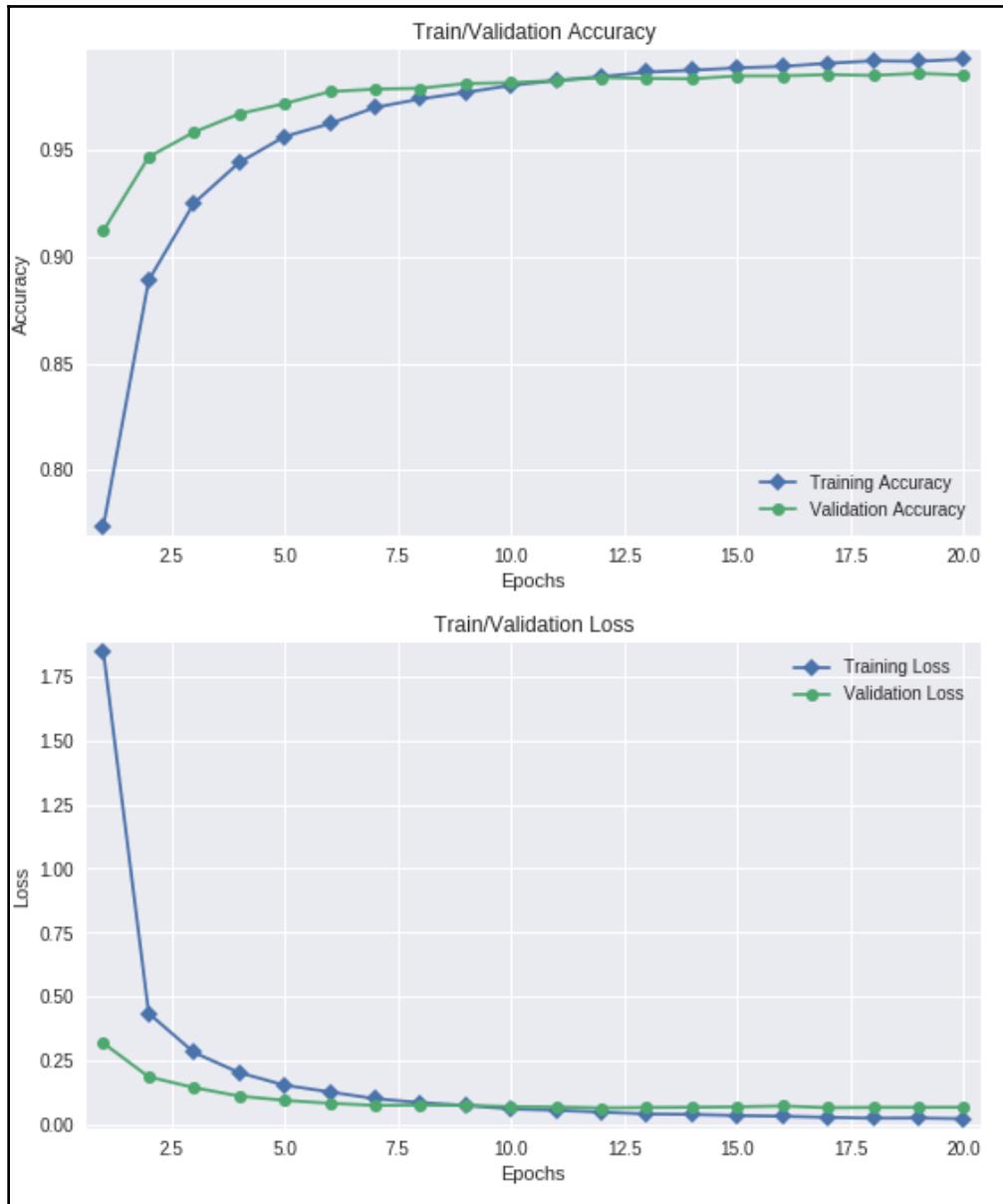


Figure 8.27: Loss/accuracy plot of the convolution classifier with max pooling and dropout during training

Convolution with pooling – Python file

This module implements the training and evaluation of a convolution classifier with the max pool and Dropout operations:

```
"""This module implements a deep conv classifier with max pool and
dropout."""
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPool2D, Dropout
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from loss_plot import loss_plot

# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.0001)
# Shape of the input image
input_shape = (28, 28, 1)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                stratify = y_train,
                                                test_size = 0.08333,
                                                random_state=42)

X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape,
                activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
```

```
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax'))\n\nmodel.compile(loss = 'sparse_categorical_crossentropy', optimizer=
optimizer,
metrics = ['accuracy'])\n\nhistory = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size,
validation_data=(X_val, y_val))\n\nloss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)\n\nloss_plot(history)
```

Going deeper

The convolution classifier with max pooling and dropout seems to be the best classifier so far. However, we also noticed that there was a slight amount of overfitting on the train data.

Let's build a deeper model to see if we can create a classifier that is more accurate than the other models we have trained so far, and see if we can get it to reach an even better minima.

We will build a deeper model by adding two more convolution layers to our best model so far:

- The first layer is a convolution 2-D layer with 32 filters of size 3*3 with activation as `relu`, followed by downsampling with max pooling of size 2*2, followed by `Dropout` as the regularizer
- The second layer is a convolution 2-D layer with 64 filters of size 3*3 with activation as `relu`, followed by downsampling with max pooling of size 2*2, followed by `Dropout` as the regularizer
- The third layer is a convolution 2-D layer with 128 filters of size 3*3 with activation as `relu`, followed by downsampling with max pooling of size 2*2, followed by `Dropout` as the regularizer

Compiling the model

The following is the code for the deeper model:

```
# model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape, activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax')))

# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer, metrics = ['accuracy'])

# print model summary
model.summary()
```

The following is the output of the preceding code:

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_20 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_18 (MaxPooling)	(None, 13, 13, 32)	0
dropout_21 (Dropout)	(None, 13, 13, 32)	0
conv2d_21 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_19 (MaxPooling)	(None, 5, 5, 64)	0
dropout_22 (Dropout)	(None, 5, 5, 64)	0
conv2d_22 (Conv2D)	(None, 3, 3, 128)	73856
max_pooling2d_20 (MaxPooling)	(None, 1, 1, 128)	0
dropout_23 (Dropout)	(None, 1, 1, 128)	0
flatten_12 (Flatten)	(None, 128)	0
dense_51 (Dense)	(None, 128)	16512
dropout_24 (Dropout)	(None, 128)	0
dense_52 (Dense)	(None, 10)	1290
<hr/>		
Total params: 110,474		
Trainable params: 110,474		
Non-trainable params: 0		

Figure 8.28: Summary of the deep convolution classifier

From the summary, we can see that the deeper model has only 110,474 parameters. Now, let's see if a deeper model with fewer parameters can do a better job than we have done so far.

Fitting the model

Just like we did previously, fit the model, but with epochs set as 40 instead of 20, since the deeper model takes longer to learn. Try training the model for 20 epochs first to see what happens:

```
# fit model
history = model.fit(X_train, y_train, epochs = 40, batch_size=batch_size,
validation_data=(X_val, y_val))
```

The following is the output of the preceding code:

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/40
55000/55000 [=====] - 9s 172us/step - loss: 4.1928 - acc: 0.3508 - val_loss: 1.0419 - val_acc: 0.7016
Epoch 2/40
55000/55000 [=====] - 8s 150us/step - loss: 1.2163 - acc: 0.5978 - val_loss: 0.5370 - val_acc: 0.8628
Epoch 3/40
55000/55000 [=====] - 8s 150us/step - loss: 0.7717 - acc: 0.7507 - val_loss: 0.3267 - val_acc: 0.9118
```

The following is the output at the end of the code's execution:

```
Epoch 39/40
55000/55000 [=====] - 8s 154us/step - loss: 0.0517 - acc: 0.9843 - val_loss: 0.0363 - val_acc: 0.9898
Epoch 40/40
55000/55000 [=====] - 8s 154us/step - loss: 0.0510 - acc: 0.9838 - val_loss: 0.0370 - val_acc: 0.9884
```

Figure 8.29: Metrics printed out during the training of the deep convolution classifier

Evaluating the model

Now, evaluate the model with the following code:

```
# evaluate model
loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)
```

The following is the output of the preceding code:

```
10000/10000 [=====] - 1s 123us/step
Test loss: 0.03408890862933331
Accuracy: 0.9901
```

Figure 8.30: Printout of the evaluation of the deep convolution classifier

We can see that the model is 99.01% accurate on the test data, 98.84% on the validation data, and 98.38% on the train data. The deeper convolution model with pooling and dropout gives a much better performance with just 110,000 parameters. If you look at the loss as well, this model was able to reach a better minima than the other models that we trained previously:

Plot the metrics to understand how the training has progressed:

```
# plot training loss
loss_plot(history)
```

The following is the output of the preceding code:

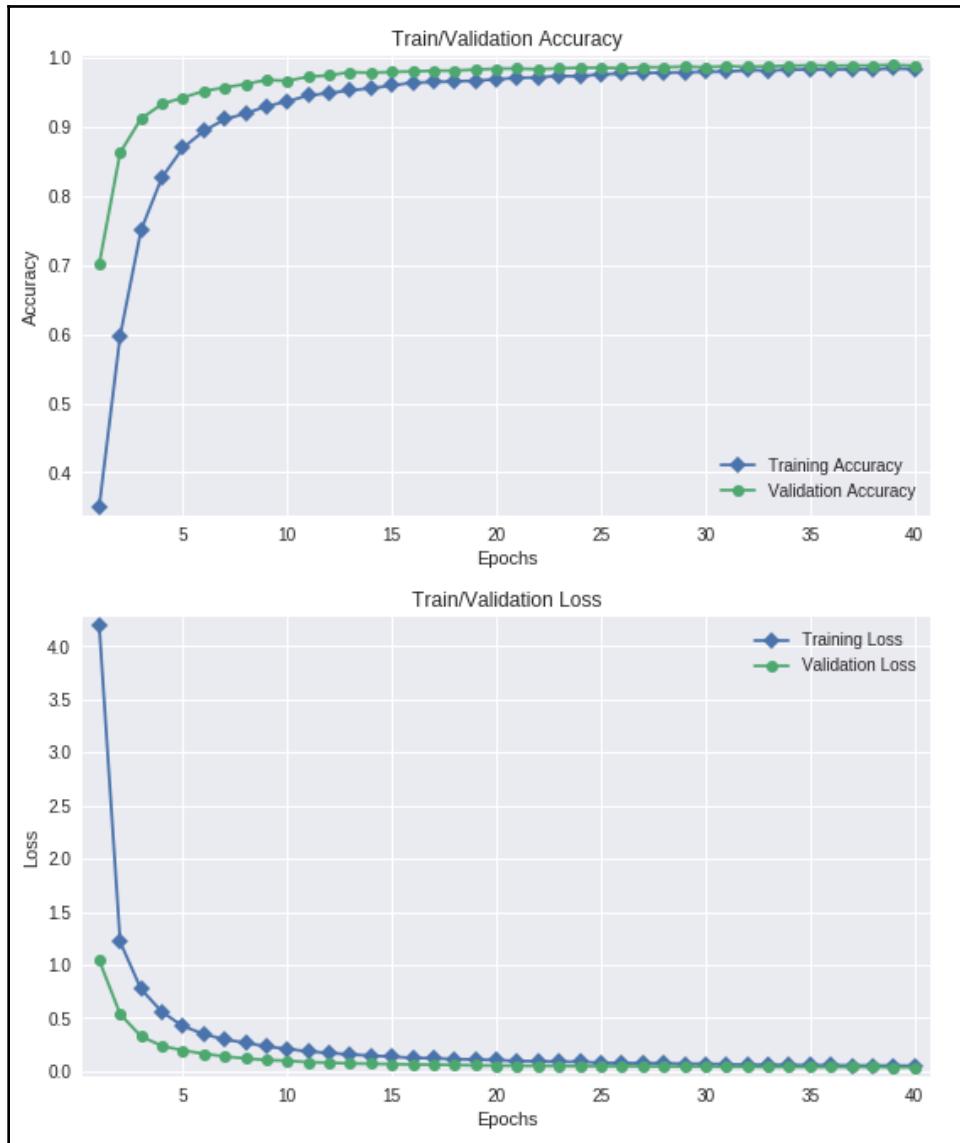


Figure 8.31: Loss/accuracy plot of the deep convolution classifier during training

This is one of the best training plots you can get. We can see no overfitting at all.

Convolution with pooling and Dropout – Python file

This module implements the training and evaluation of a deep convolution classifier with the max pool and Dropout operations:

```
"""This module implements a deep conv classifier with max pool and
dropout."""
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPool2D, Dropout
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from loss_plot import loss_plot

# Number of epochs
epochs = 20
# Batchsize
batch_size = 128
# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.0001)
# Shape of the input image
input_shape = (28, 28, 1)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                stratify = y_train,
                                                test_size = 0.08333,
                                                random_state=42)

X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape,
                activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
```

```
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax'))

model.compile(loss = 'sparse_categorical_crossentropy', optimizer=
optimizer,
               metrics = ['accuracy'])

history = model.fit(X_train, y_train, epochs = epochs,
batch_size=batch_size,
               validation_data=(X_val, y_val))

loss,acc = model.evaluate(X_test, y_test)
print('Test loss:', loss)
print('Accuracy:', acc)

loss_plot(history)
```

Data augmentation

Imagine a situation where you might want to build a convolution classifier on a small set of images. The problem here is that the classifier will easily overfit on this small set of data. The reason why the classifier will overfit is that there are very few images that are similar. That is, there are not a lot of variations for the model to capture within a specific class so that it can be robust and perform well on new data.

Keras provides a preprocessing utility called `ImageDataGenerator` that can be used to augment image data with simple configuration.

Its capabilities include the following:

- `zoom_range`: Randomly zoom in on images to a given zoom level
- `horizontal_flip`: Randomly flip images horizontally
- `vertical_flip`: Randomly flip images vertically
- `rescale`: Multiply the data with the factor provided

It also includes capabilities for random rotations, random shear, and many more.



Visit the official Keras documentation (<https://keras.io/preprocessing/image/>) to learn more about some of the additional functionalities of the `image_data_generator` API.

Using ImageDataGenerator

The `image_data_generator` API transforms and augments the data in batches on the go, and is also super easy to use.

First, import the `ImageDataGenerator`:

```
from keras.preprocessing.image import ImageDataGenerator
```

Implement a random horizontal flip augmenter:

```
train_datagen = ImageDataGenerator(horizontal_flip=True)
```

Fit the augmenter on the train data:

```
# fit the augmenter
train_datagen.fit(X_train)
```

After the fit, we usually use the `transform` command. Here, instead of `transform`, we have the `flow` command. It accepts the images and its corresponding labels, and then generates batches of transformed data of the specified batch size.

Let's transform a bunch of images and look at the result:

```
# transform the data
for img, label in train_datagen.flow(X_train, y_train, batch_size=6):
    for i in range(0, 6):
        plt.subplot(2, 3, i+1)
        plt.title('Label {}'.format(label[i]))
        plt.imshow(img[i].reshape(28, 28), cmap='gray')
    break
plt.tight_layout()
plt.show()
```

The following is the output of the preceding code:

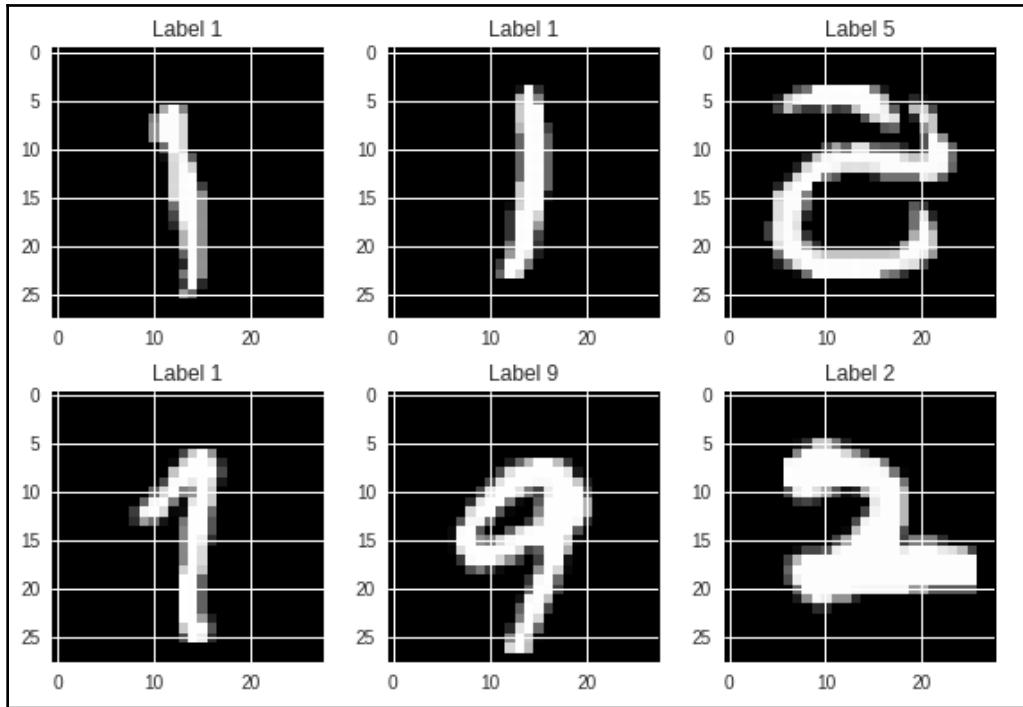


Figure 8.32: Digits after horizontal flip augmentation

Similarly, we can implement a random zoom augmenter, like so:

```
train_datagen = ImageDataGenerator(zoom_range=0.3)

#fit
train_datagen.fit(X_train)

#transform
for img, label in train_datagen.flow(X_train, y_train, batch_size=6):
    for i in range(0, 6):
        plt.subplot(2,3,i+1)
        plt.title('Label {}'.format(label[i]))
        plt.imshow(img[i].reshape(28, 28), cmap='gray')
    break
plt.tight_layout()
plt.show()
```

The following is the output of the preceding code:

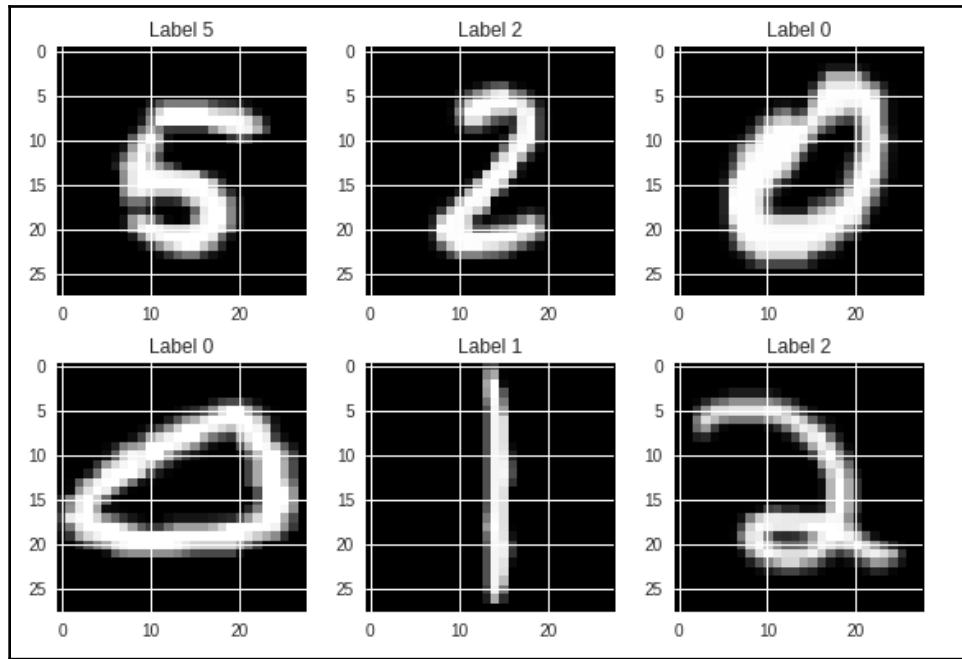


Figure 8.33: Digits after zoom augmentation

Fitting ImageDataGenerator

Now, let's build a classifier using the same architecture as the deep convolution model with pooling and Dropout, but on augmented data.

First, define the features of the `ImageDataGenerator`, as follows:

```
train_datagen = ImageDataGenerator(  
    rescale = 1./255,  
    zoom_range = 0.2,  
    horizontal_flip = True)
```

We have defined that the `ImageDataGenerator` can perform the following operations

- Rescaling
- Random zoom
- Random horizontal flip



The rescaling operation scales the pixel values to a range between 0 and 1.

The next step is to fit this generator on the train data:

```
train_datagen.fit(X_train)
```

Compiling the model

We need to define and compile the deep convolution model like so:

```
# define model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape, activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax')))

# compile model
model.compile(loss = 'sparse_categorical_crossentropy', optimizer=optimizer, metrics = ['accuracy'])
```

Fitting the model

Finally, we need to fit the model:

```
# fit the model on batches with real-time data augmentation
history = model.fit_generator(train_datagen.flow(X_train, y_train,
batch_size=128), steps_per_epoch=len(X_train) / 128, epochs=10,
validation_data=(train_datagen.flow(X_val, y_val)))
```

The following is the output of the preceding code:

```
Epoch 1/10
430/429 [=====] - 18s 43ms/step - loss: 0.7970 - acc: 0.7317 - val_loss: 0.3087 - val_acc: 0.9038
Epoch 2/10
430/429 [=====] - 17s 41ms/step - loss: 0.3466 - acc: 0.8902 - val_loss: 0.1981 - val_acc: 0.9412
Epoch 3/10
430/429 [=====] - 18s 41ms/step - loss: 0.2568 - acc: 0.9203 - val_loss: 0.1654 - val_acc: 0.9486
```

The following is the output at the end of the code's execution:

```
Epoch 9/10
430/429 [=====] - 18s 42ms/step - loss: 0.1367 - acc: 0.9578 - val_loss: 0.0953 - val_acc: 0.9720
Epoch 10/10
430/429 [=====] - 17s 41ms/step - loss: 0.1289 - acc: 0.9602 - val_loss: 0.0916 - val_acc: 0.9718
```

Figure 8.34: Metrics printed out during the training of the deep convolution classifier on augmented data

Evaluating the model

Now, we need to evaluate the model:

```
# transform/augment test data
for test_img, test_lab in train_datagen.flow(X_test, y_test, batch_size =
X_test.shape[0]):
    break

# evaluate model on test data
loss,acc = model.evaluate(test_img, test_lab)
print('Test loss:', loss)
print('Accuracy:', acc)
```

The following is the output of the preceding code:

```
10000/10000 [=====] - 1s 107us/step
Test loss: 0.07422855299189687
Accuracy: 0.9764
```

Figure 8.35: Printout of the evaluation of the deep convolution classifier on augmented data

Then, we need to plot the deep convolution classifier:

```
# plot the learning
loss_plot(history)
```

The following is the output of the preceding code:

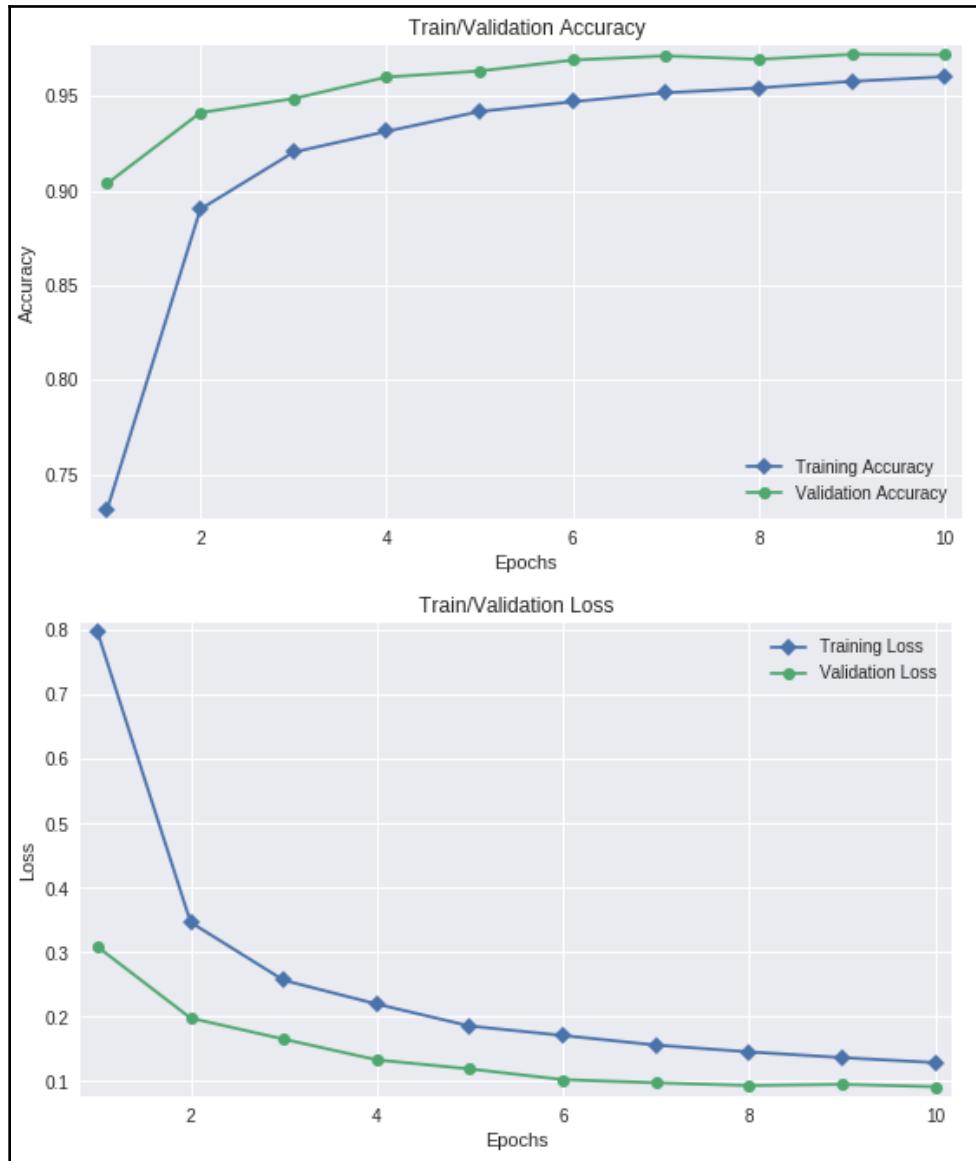


Figure 8.36: Loss/accuracy plot of the deep convolution classifier during training on augmented data

Augmentation – Python file

This module implements the training and evaluation of a deep convolution classifier on augmented data:

```
"""This module implements a deep conv classifier on augmented data."""
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPool2D, Dropout
import matplotlib.pyplot as plt
from keras.preprocessing.image import ImageDataGenerator

from sklearn.model_selection import train_test_split
from loss_plot import loss_plot

# Number of epochs
epochs = 10
# Batchsize
batch_size = 128
# Optimizer for the generator
from keras.optimizers import Adam
optimizer = Adam(lr=0.001)
# Shape of the input image
input_shape = (28, 28, 1)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
                                                stratify = y_train,
                                                test_size = 0.08333,
                                                random_state=42)

X_train = X_train.reshape(-1,28,28,1)
X_val = X_val.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

train_datagen = ImageDataGenerator(
    rescale=1./255,
    zoom_range=0.2,
    horizontal_flip=True)

train_datagen.fit(X_train)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), input_shape=input_shape,
                activation = 'relu'))
model.add(MaxPool2D(2,2))
```

```
model.add(Dropout(0.2))
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Conv2D(128, kernel_size=(3,3), activation = 'relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation = 'softmax'))


model.compile(loss = 'sparse_categorical_crossentropy', optimizer=
optimizer,
               metrics = ['accuracy'])

# fits the model on batches with real-time data augmentation:
history = model.fit_generator(train_datagen.flow(X_train, y_train,
                                                 batch_size=128),
                               steps_per_epoch=len(X_train) / 128,
                               epochs=epochs,
                               validation_data=(train_datagen.flow(X_val,
                                                     y_val))) 

for test_img, test_lab in train_datagen.flow(X_test, y_test,
                                             batch_size = X_test.shape[0]):
    break

loss,acc = model.evaluate(test_img, test_lab)
print('Test loss:', loss)
print('Accuracy:', acc)

loss_plot(history)
```

Additional topic – convolution autoencoder

An autoencoder is a combination of two parts: an encoder and a decoder. The encoder and decoder of a simple autoencoder are usually made up of dense layers, whereas in a convolution autoencoder, they are made of convolution layers:

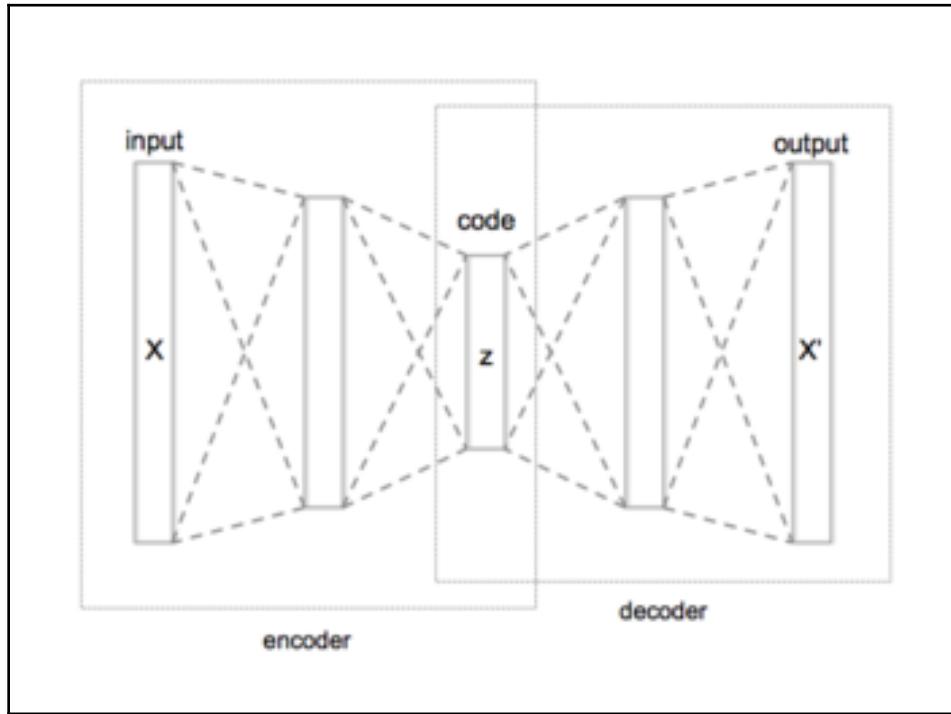


Figure 8.37: The structure of an autoencoder (image source: Wikipedia)

The encoder part of the autoencoder accepts an image and compresses it into a smaller size with the help of a pooling operation. In our case, this is max pooling. The decoder accepts the input of the encoder and learns to expand the image to our desired size by using convolution and upsampling.

Imagine a situation where you want to build high-resolution images out of blurred images:

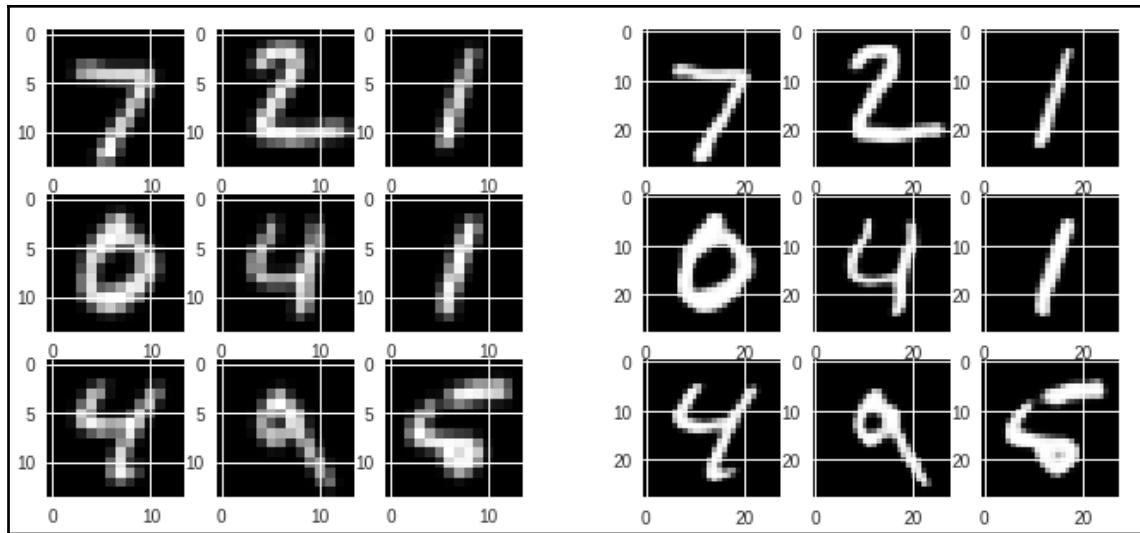


Figure 8.38: Low-resolution digits on the left and high-resolution digits on the right

Convolution autoencoders are capable of doing this job very well. The preceding high-resolution digits that you can see were actually generated using convolution autoencoders.

By the end of this section, you will have built a convolution autoencoder that accepts low-resolution 14*14*1 MNIST digits and generates high-resolution 28*28*1 digits.

Importing the dependencies

Consider restarting your session before starting this section:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model, Sequential
from keras.optimizers import Adam
from keras import backend as k

# for resizing images
from scipy.misc import imresize
```

Generating low-resolution images

To generate low-resolution images, define a function called `reshape()` that will resize the input image/digit to size 14×14 . After defining this, we will use the `reshape()` function to generate low-resolution train and test images:

```
def reshape(x):
    """Reshape images to 14*14"""
    img = imresize(x.reshape(28,28), (14, 14))
    return img

# create 14*14 low resolution train and test images
XX_train = np.array([*map(reshape, X_train.astype(float))])
XX_test = np.array([*map(reshape, X_test.astype(float))])
```

`XX_train` and `XX_test` will be the images that we will feed into the encoder, and `X_train` and `X_test` will be the targets.

Scaling

Scale the train input, test input, and target images to range between 0 and 1 so that the learning process is faster:

```
# scale images to range between 0 and 1
# 14*14 train images
XX_train = XX_train/255
# 28*28 train label images
X_train = X_train/255

# 14*14 test images
XX_test = XX_test/255
# 28*28 test label images
X_test = X_test/255
```

Defining the autoencoder

The convolution autoencoder we are going to build will accept $14 \times 14 \times 1$ images as input with $28 \times 28 \times 1$ images as the targets, and will have the following characteristics:

In the encoder:

- The first layer is a convolution 2-D layer with 64 filters of size 3*3, followed by batch normalization, with activation as `relu`, followed by downsampling with `MaxPooling2D` of size 2*2
- The second layer, or the final layer in this encoder part, is again a convolution 2-D layer with 128 filters of size 3*3, batch normalization, with activation as `relu`

In the decoder:

- The first layer is a convolution 2-D layer with 128 filters of size 3*3 with activation as `relu`, followed by upsampling that's performed with `UpSampling2D`
- The second layer is a convolution 2-D layer with 64 filters of size 3*3 with activation as `relu`, followed by upsampling with `UpSampling2D`
- The third layer, or the final layer in this decoder part, is again a convolution 2-D layer with 1 filter of size 3*3 with activation as `sigmoid`

The following is the code for our autoencoder:

```
batch_size = 128
epochs = 40
input_shape = (14, 14, 1)

# define autoencoder
def make_autoencoder(input_shape):
    generator = Sequential()
    generator.add(Conv2D(64, (3, 3), activation='relu', padding='same',
input_shape=input_shape))
    generator.add(MaxPooling2D(pool_size=(2, 2)))
    generator.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    generator.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    generator.add(UpSampling2D((2, 2)))
    generator.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
    generator.add(UpSampling2D((2, 2)))
    generator.add(Conv2D(1, (3, 3), activation='sigmoid', padding='same'))

    return generator

autoencoder = make_autoencoder(input_shape)

# compile auto encoder
autoencoder.compile(loss='mean_squared_error', optimizer = Adam(lr=0.0002,
beta_1=0.5))
```

```
# auto encoder summary  
autoencoder.summary()
```

The following is the output of the preceding code:

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 14, 14, 64)	640
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 64)	0
conv2d_7 (Conv2D)	(None, 7, 7, 128)	73856
conv2d_8 (Conv2D)	(None, 7, 7, 128)	147584
up_sampling2d_3 (UpSampling2D)	(None, 14, 14, 128)	0
conv2d_9 (Conv2D)	(None, 14, 14, 64)	73792
up_sampling2d_4 (UpSampling2D)	(None, 28, 28, 64)	0
conv2d_10 (Conv2D)	(None, 28, 28, 1)	577
<hr/>		
Total params: 296,449		
Trainable params: 296,449		
Non-trainable params: 0		

Figure 8.39: Autoencoder summary

We are using `mean_squared_error` as the loss, as we want the model to predict the pixel values.

If you take a look at the summary, the input image of size $14 \times 14 \times 1$ is compressed along the width and the height dimensions to a size of 7×7 , but is expanded along the channel dimension from 1 to 128. These small/compressed feature maps are then fed to the decoder to learn the mappings that are required to generate high-resolution images of the defined dimension, which in this case is $28 \times 28 \times 1$.



If you have any questions about the usage of the Keras API, please visit the Keras official documentation at <https://keras.io/>.

Fitting the autoencoder

Like any regular model fit, fit the autoencoder:

```
# fit autoencoder
autoencoder_train = autoencoder.fit(XX_train.reshape(-1,14,14,1),
X_train.reshape(-1,28,28,1), batch_size=batch_size,
                                         epochs=epochs, verbose=1,
                                         validation_split = 0.2)
```

The following is the output of the preceding code:

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/40
48000/48000 [=====] - 13s 271us/step - loss: 0.0279 - val_loss: 0.0101
Epoch 2/40
48000/48000 [=====] - 11s 226us/step - loss: 0.0081 - val_loss: 0.0065
Epoch 3/40
48000/48000 [=====] - 11s 226us/step - loss: 0.0059 - val_loss: 0.0052
```

The following is the output at the end of the code's execution:

```
Epoch 39/40
48000/48000 [=====] - 11s 227us/step - loss: 0.0017 - val_loss: 0.0018
Epoch 40/40
48000/48000 [=====] - 11s 224us/step - loss: 0.0017 - val_loss: 0.0020
```

Figure 8.40: Printout during the training of the autoencoder

You will notice that inside the fit, we have specified a parameter called validation_split and that we have set it to 0.2. This will split the train data into train and validation data, with validation data having 20% of the original train data.

Loss plot and test results

Now, let's get to plotting the train and validation loss progression during training. We will also plot the high-resolution image result from the model by feeding the test images:

```
loss = autoencoder_train.history['loss']
val_loss = autoencoder_train.history['val_loss']
epochs_ = [x for x in range(epochs)]
plt.figure()
plt.plot(epochs_, loss, label='Training loss')
plt.plot(epochs_, val_loss, label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
```

```
plt.show()

print('Input')
plt.figure(figsize=(5, 5))
for i in range(9):
    plt.subplot(331 + i)
    plt.imshow(np.squeeze(XX_test.reshape(-1, 14, 14)[i]), cmap='gray')
plt.show()

# Test set results
print('GENERATED')
plt.figure(figsize=(5, 5))
for i in range(9):
    pred = autoencoder.predict(XX_test.reshape(-1, 14, 14, 1)[i:i+1],
verbose=0)
    plt.subplot(331 + i)
    plt.imshow(pred[0].reshape(28, 28), cmap='gray')
plt.show()
```

The following is the output of the preceding code:

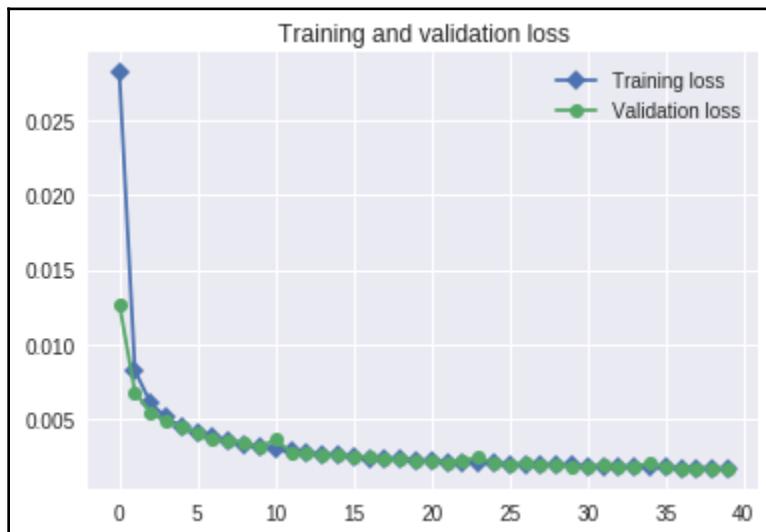


Figure 8.41: Train/val loss plot

The following is the output of high-resolution images that have been generated from low-resolution images:

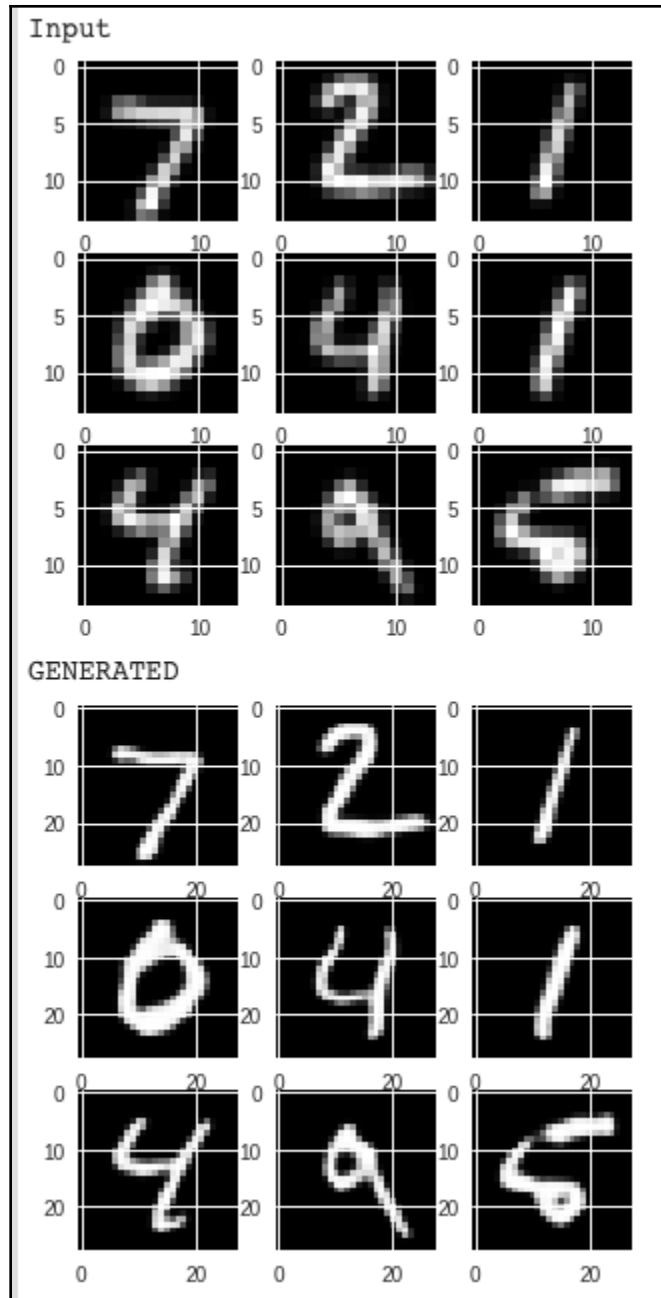


Figure 8.42: High-resolution test (28*28) images generated from low-resolution test (14*14) images

Autoencoder – Python file

This module implements training an autoencoder on MNIST data:

```
"""This module implements a convolution autoencoder on MNIST data."""
import numpy as np
import matplotlib.pyplot as plt

from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model, Sequential
from keras.optimizers import Adam
from keras import backend as k

# for resizing images
from scipy.misc import imresize

def reshape(x):
    """Reshape images to 14*14"""
    img = imresize(x.reshape(28,28), (14, 14))
    return img

# create 14*14 low resolution train and test images
XX_train = np.array([*map(reshape, X_train.astype(float))])
XX_test = np.array([*map(reshape, X_test.astype(float))])

# scale images to range between 0 and 1
#14*14 train images
XX_train = XX_train/255
#28*28 train label images
X_train = X_train/255

#14*14 test images
XX_test = XX_test/255
#28*28 test label images
X_test = X_test/255

batch_size = 128
epochs = 40
input_shape = (14,14,1)

def make_autoencoder(input_shape):

    generator = Sequential()
    generator.add(Conv2D(64, (3, 3), activation='relu', padding='same',
                        input_shape=input_shape))
```

```
generator.add(MaxPooling2D(pool_size=(2, 2)))

generator.add(Conv2D(128, (3, 3), activation='relu', padding='same'))

generator.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
generator.add(UpSampling2D((2, 2)))

generator.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
generator.add(UpSampling2D((2, 2)))

generator.add(Conv2D(1, (3, 3), activation='sigmoid', padding='same'))

return generator

autoencoder = make_autoencoder(input_shape)
autoencoder.compile(loss='mean_squared_error', optimizer = Adam(lr=0.0002,
beta_1=0.5))

autoencoder_train = autoencoder.fit(XX_train.reshape(-1,14,14,1),
X_train.reshape(-1,28,28,1),
batch_size=batch_size,
epochs=epochs, verbose=1,
validation_split = 0.2)

loss = autoencoder_train.history['loss']
val_loss = autoencoder_train.history['val_loss']
epochs_ = [x for x in range(epochs)]
plt.figure()
plt.plot(epochs_, loss, label='Training loss', marker = 'D')
plt.plot(epochs_, val_loss, label='Validation loss', marker = 'o')
plt.title('Training and validation loss')
plt.legend()
plt.show()

print('Input')
plt.figure(figsize=(5,5))
for i in range(9):
    plt.subplot(331 + i)
    plt.imshow(np.squeeze(XX_test.reshape(-1,14,14)[i]), cmap='gray')
plt.show()

# Test set results
print('GENERATED')
plt.figure(figsize=(5,5))
for i in range(9):
    pred = autoencoder.predict(XX_test.reshape(-1,14,14,1)[i:i+1],
verbose=0)
```

```
plt.subplot(331 + i)
plt.imshow(pred[0].reshape(28, 28), cmap='gray')
plt.show()
```

Conclusion

This project was all about building a CNN classifier to classify handwritten digits better than we did in Chapter 16, *Training NN for Prediction Using Regression*, with a multilayer Perceptron.

Our deep convolution neural network classifier with max pooling and dropout hit 99.01% accuracy on a test set of 10,000 images/digits. This is good. This is almost 12% better than our multilayer Perceptron model.

However, there are some implications. What are the implications of this accuracy? It is important that we understand this. Just like we did in Chapter 16, *Training NN for Prediction Using Regression*, let's calculate the incidence of an error occurring that would result in a customer service issue.

Just to refresh our memory, in this hypothetical use case, we assumed that the restaurant has an average of 30 tables at each location, and that those tables turn over two times per night during the rush hour when the system is likely to be used, and finally that the restaurant chain has 35 locations. This means that each day of operation, there are approximately 21,000 handwritten numbers being captured (30 tables x 2 turns/day x 35 locations x 10-digit phone number).

The ultimate goal is to classify all of the digits properly, since even a single-digit misclassification will result in a failure. With the classifier that we have built, it would improperly classify 208 digits per day. If we consider the worst case scenario, out of the 2,100 patrons, 208 phone numbers would be misclassified. That is, even in the worst case, 90.09% $((2,100-208)/2,100)$ of the time, we would be sending the text to the right patron.

The best case scenario would be that if all ten digits were misclassified in each phone number, we would only be improperly classifying 21 phone numbers. This means that we would have a failure rate of $((2,100-21)/2,100)$ 1%. This is as good as it gets.

Unless you aim at reducing that 1% error...

Summary

In this chapter, we understood how to implement a convolution neural network classifier in Keras. You now have a brief understanding of what convolution, average, max pooling, and dropout are, and you also built a deep model. You understood how to reduce overfitting as well as how to generate more/validation in data to build a generalizable model when you have less data than you need. Finally, we assessed the model's performance on test data and determined that we succeeded in achieving our goal. We ended this chapter by introducing you to autoencoders.

20

Object Detection Using

OpenCV and TensorFlow

Welcome to the second chapter focusing on computer vision in *Python Deep Learning Projects* (a data science pun to kick us off!). Let's think about what we accomplished in Chapter 19, *Handwritten Digits Classification Using ConvNets*, where we were able to train an image classifier with a **convolutional neural network (CNN)** to accurately classify handwritten digits in an image. What was a key characteristic of the raw data, and what was our business objective? The data was less complicated than it could have been because each image only had one handwritten digit in it and our goal was to accurately assign a digital label to the image.

What would have happened if each image had multiple handwritten digits in it? What would have happened if we had a video of the digits? What if we want to identify where the digits are in the image? These questions represent challenges that real-world data embodies, and they drive our data science innovation to new models and capabilities.

Let's expand our line of questions and imagination to the next (hypothetical) business use case for our Python deep learning project, where we're looking to build, train, and test an object detection and classification model to be used by an automobile manufacturer in their new line of self-driving cars. Autonomous vehicles need to have fundamental computer vision capabilities that you and I have organically by way of our physiology and experiential learning. We as humans can examine our field of vision and report whether or not a specific item is present and where in relation to other objects that item (if present) is located. So, if I were to ask you if you see a chicken, you'd likely say no, unless you live on a farm and are looking out your window. But if I ask you if you see a keyboard, you'd likely say yes, and could even say that the keyboard is different from other objects and is in front of the wall before you.

This is no trivial task for a computer. As Deep Learning Engineers, you are going to learn the intuition and model architecture that empowers you to build a powerful object detection and classification engine that we can envision being tested for use in autonomous vehicles. The data inputs that we're going to be working with in this chapter will be much more informationally complex than what we've had in previous projects, and the outcomes when we get them right will be that much more impressive.

So, let's get started!

Object detection intuition

When you need your application to find and name things in an image, you need to build a deep neural network for object detection. The visual field is very complex, and a camera for still images and video captures frames with many, many objects in them. Object detection is used in manufacturing for process automation in production lines; autonomous vehicles sensing pedestrians, other cars, the road, and signs, for example; and, of course, facial recognition. Computer vision solutions based on machine learning and deep learning require you, the Data Scientist, to build, train, and evaluate models that can differentiate one object from another and then accurately classify those detected objects.

As you've seen in other projects we've worked on, CNNs are very powerful models for image data. We need to look at expansions on the basic architecture that has performed so well on a single (still) image with simple information to see what works best for complex images and video.

Progress recently has been made with these networks: Faster R-CNN, **region-based fully convolutional network (R-FCN)**, MultiBox, **solid-state drive (SSD)**, and **you only look once (YOLO)**. We've seen the value of these models in common consumer applications such as Google Photos and Pinterest Visual Search. We are even seeing some of these that are lightweight and fast enough to perform well on mobile devices.

Recent progress in the field can be researched with the following list of references:

- PVANET: Deep but Lightweight Neural Networks for Real-time Object Detection, arXiv:1608.08021
- R-CNN: Rich feature hierarchies for accurate object detection and semantic segmentation, CVPR, 2014.
- SPP: Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition, ECCV, 2014.
- Fast R-CNN, arXiv:1504.08083.
- Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, arXiv:1506.01497.
- R-CNN minus R, arXiv:1506.06981.
- End-to-end people detection in crowded scenes, arXiv:1506.04878.
- YOLO – You Only Look Once: Unified, Real-Time Object Detection, arXiv:1506.02640
- Inside-Outside Net: Detecting Objects in Context with Skip Pooling and Recurrent Neural Networks
- Deep Residual Network: Deep Residual Learning for Image Recognition
- R-FCN: Object Detection via Region-based Fully Convolutional Networks
- SSD: Single Shot MultiBox Detector, arXiv:1512.02325

Also, following is the timeline of how the evolution of object detection has developed from 1999–2017:

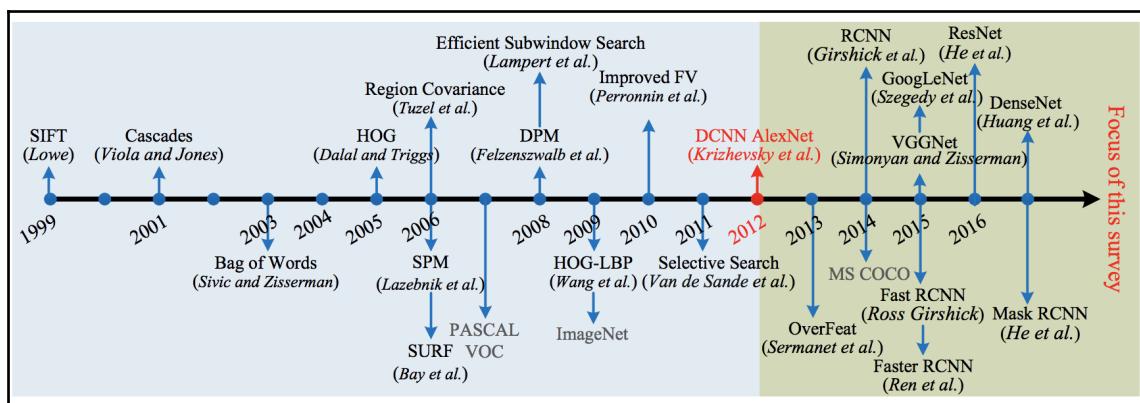


Figure 9.1: The timeline of the evolution of object detection from 1999 to 2017

Improvements in object detection models

Object detection and classification has been the subject of study for quite some time. The models that have been used build on the great success of previous researchers. A brief summary of progress history starts by highlighting the computer vision model called **Histogram of Oriented Gradients (HOG)** features that was developed by Navneet Dalal and Bill Triggs in 2005.

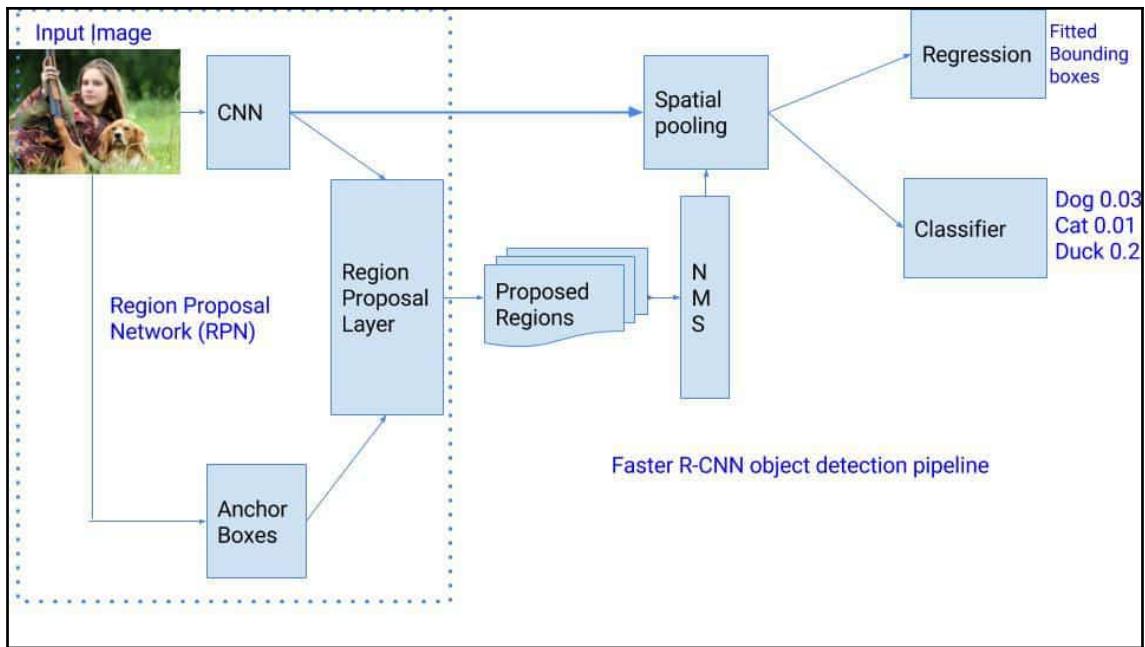
HOG features were fast and performed well. Interest in deep learning and the great success of CNNs that were more accurate classifiers due to their deep networks. But the problem was that the CNNs of the time were too slow in comparison.

The solution was to take advantage of the CNNs, improved classification capabilities and improve their speed with a technique and employ a selective search paradigm in what became known as R-CNN. Reducing the number of bounding boxes did show improvements in speed, but not sufficiently for the expectations.

SPP-net was a proposed solution, wherein a CNN representation for the whole image was calculated and drove CNN-calculated representations for each sub-section generated by selective search. Selective search uses image features to generate all the possible locations for an object by looking at pixel intensity, color, image texture, and a measure of insideness. These identified objects are then fed into the CNN model for classification.

This, in turn, saw improvements in a model named Fast R-CNN that trained end-to-end, and thereby fixed the primary problems with SPP-net and R-CNN. Advancing this technology further with a model named Faster R-CNN, the technique of using small regional proposal CNNs in place of the selective search performed very well.

Here is a quick overview of the Faster R-CNN object detection pipeline:



A quick benchmark comparison of the versions of R-CNN discussed previously shows the following:

	R-CNN	Fast R-CNN	Faster R-CNN
Average response time	~50 sec	~2 sec	~0.2 sec
Speed boost	1x	25x	250x

The performance improvement is impressive, with Faster R-CNN being one of the most accurate and fastest object detection algorithms deployed in real-time use cases. Other recent powerful alternatives include YOLO models, which we will look into in detail later in this chapter.

Object detection using OpenCV

Let's start our project with a basic or traditional implementation of **Open Source Computer Vision (OpenCV)**. This library is primarily targeted at real-time applications that need computer vision capabilities.



OpenCV has its API wrappers in various languages such as C, C++, Python, and so on, and the best way forward is to build a quick prototype using Python wrappers or any other language you are comfortable with, and once you are ready with your code, rewrite it in C/C++ for production.

In this chapter, we will be using the Python wrappers to create our initial object detection module.

So, let's do it.

A handcrafted red object detector

In this section, we will learn how to create a feature extractor that will be able to detect any red object from the provided image using various image processing techniques such as erosion, dilation, blurring, and so on.

Installing dependencies

First, we need to install OpenCV, which we do with this simple pip command:

```
pip install opencv-python
```

Then we will import it along with other modules for visualizations and matrix operations:

```
import cv2
import matplotlib
from matplotlib import colors
from matplotlib import pyplot as plt
import numpy as np
from __future__ import division
```

Also, let's define some helper functions that will help us to plot the images and the contours:

```
# Defining some helper function
def show(image):
    # Figure size in inches
    plt.figure(figsize=(15, 15))
    # Show image, with nearest neighbour interpolation
    plt.imshow(image, interpolation='nearest')
def show_hsv(hsv):
    rgb = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
    show(rgb)
```

```
def show_mask(mask):
    plt.figure(figsize=(10, 10))
    plt.imshow(mask, cmap='gray')
def overlay_mask(mask, image):
    rgb_mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2RGB)
    img = cv2.addWeighted(rgb_mask, 0.5, image, 0.5, 0)
    show(img)

def find_biggest_contour(image):
    image = image.copy()
    im2, contours, hierarchy = cv2.findContours(image, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)

    contour_sizes = [(cv2.contourArea(contour), contour) for contour in
contours]
    biggest_contour = max(contour_sizes, key=lambda x: x[0])[1]

    mask = np.zeros(image.shape, np.uint8)
    cv2.drawContours(mask, [biggest_contour], -1, 255, -1)
    return biggest_contour, mask

def circle_countour(image, countour):
    image_with_ellipse = image.copy()
    ellipse = cv2.fitEllipse(countour)

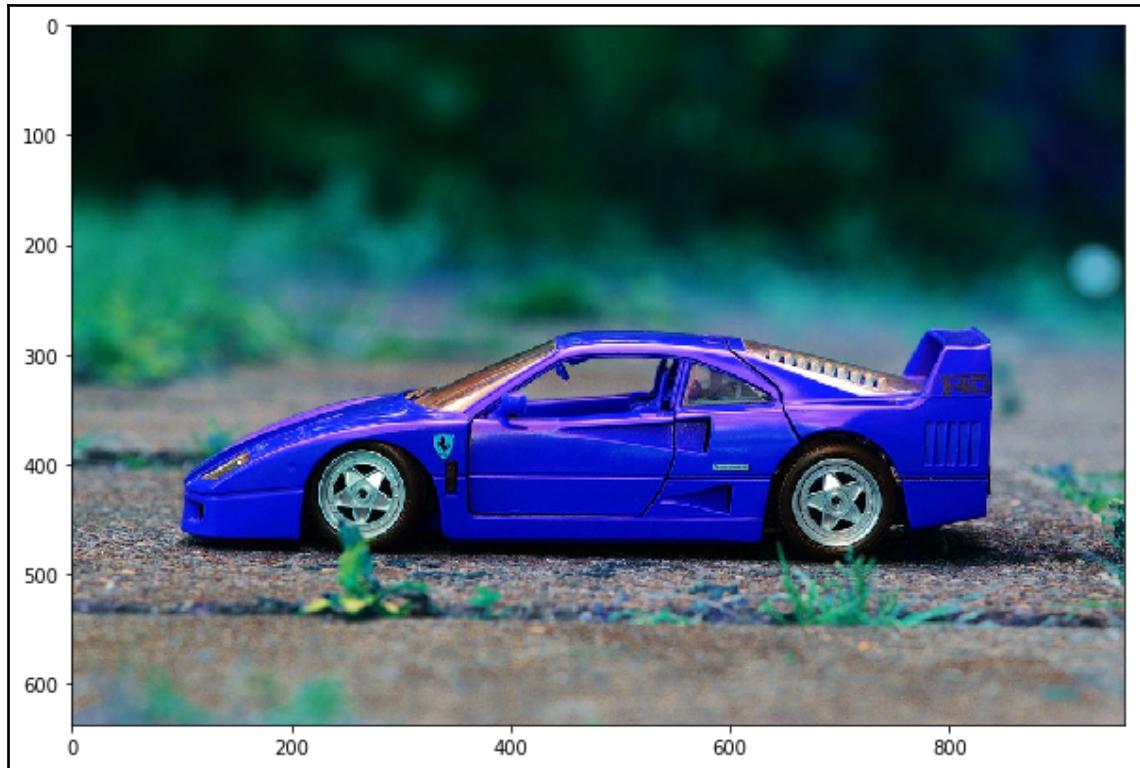
    cv2.ellipse(image_with_ellipse, ellipse, (0,255,0), 2)
    return image_with_ellipse
```

Exploring image data

The first thing in any data science problem is to explore and understand the data. This helps us to make our objective clear. So, let's first load the image and examine the properties of that image, such as the color spectrum and the dimensions:

```
# Loading image and display
image = cv2.imread('./ferrari.png')
show(image)
```

Following is the output:



Since the order of the image stored in the memory is **Blue Green Red (BGR)**, we need to convert it into **Red Green Blue (RGB)**:

```
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
show(image)
```

Following is the output:

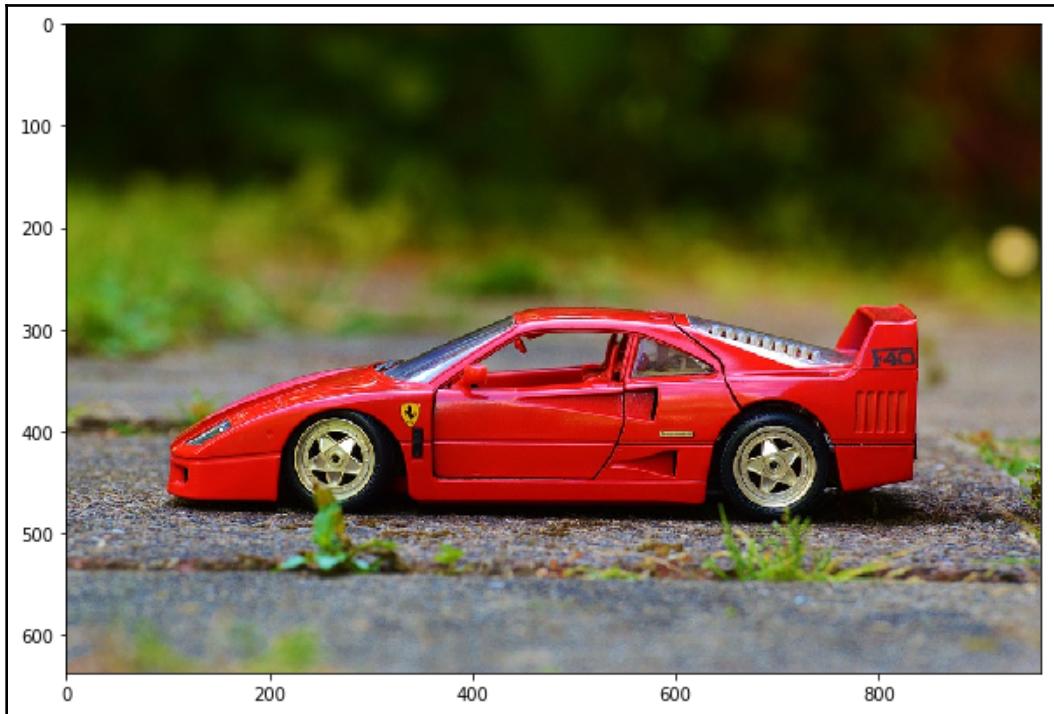


Figure 9.2: The raw input image in RGB color format.

Normalizing the image

We will be scaling down the image dimensions, for which we will be using the `cv2.resize()` function:

```
max_dimension = max(image.shape)
scale = 700/max_dimension
image = cv2.resize(image, None, fx=scale, fy=scale)
```

Now we will perform a blur operation to make the pixels more normalized, for which we will be using the Gaussian kernel. Gaussian filters are very popular in the research field and are used for various operations, one of which is the blurring effect that reduces the noise and balances the image. The following code performs a blur operation:

```
image.blur = cv2.GaussianBlur(image, (7, 7), 0)
```

Then we will convert the RGB-based image into an HSV color spectrum, which will help us to extract other characteristics of the image using color intensity, brightness, and shades:

```
image.blur.hsv = cv2.cvtColor(image.blur, cv2.COLOR_RGB2HSV)
```

Following is the output:

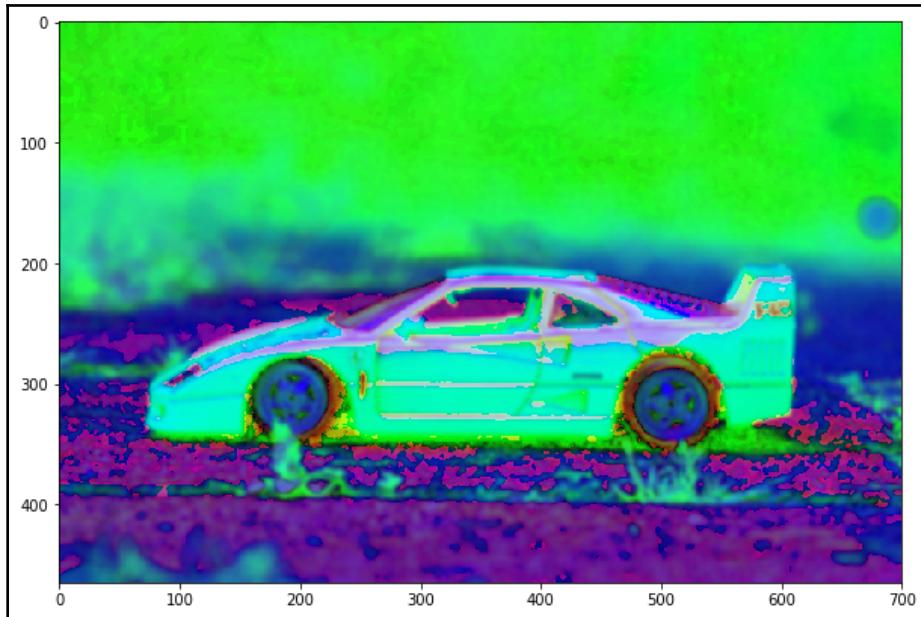


Figure: 9.3: The raw input image in HSV color format.

Preparing a mask

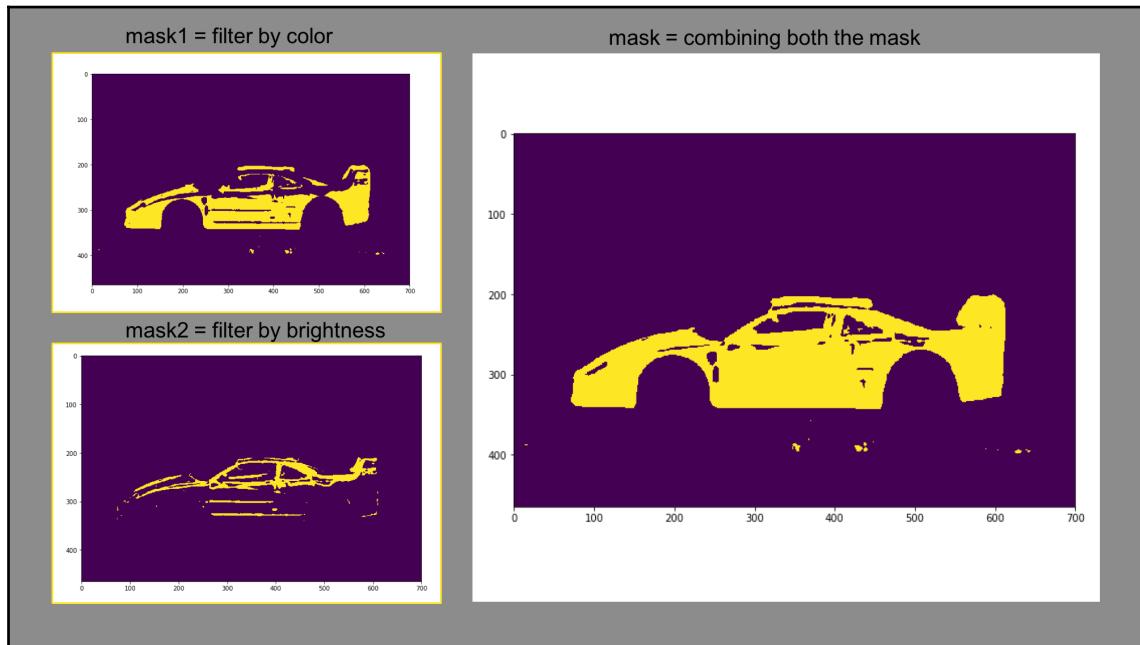
We need to create a mask that can detect the specific color spectrum; let's say red in our case. Now we will create two masks that will be performing feature extraction using the color values and the brightness factors:

```
# filter by color
min_red = np.array([0, 100, 80])
max_red = np.array([10, 256, 256])
mask1 = cv2.inRange(image.blur.hsv, min_red, max_red)

# filter by brightness
min_red = np.array([170, 100, 80])
max_red = np.array([180, 256, 256])
mask2 = cv2.inRange(image.blur.hsv, min_red, max_red)
```

```
# Concatenate both the mask for better feature extraction  
mask = mask1 + mask2
```

Following is how our mask looks:



Post-processing of a mask

Once we are able to create our mask successfully, we need to perform some morphological operations, which are basic image processing operations used for the analysis and processing of geometrical structures.

First, we will create a kernel that will perform various morphological operations over the input image:

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (15, 15))
```

Closing: *Dilation followed by erosion* is helpful to close small pieces inside the foreground objects or small black points on the object.



Now let's perform the close operation over the mask:

```
mask_closed = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

The opening operation *erosion followed by dilation* is used to remove noise.



Then we perform the opening operation:

```
mask_clean = cv2.morphologyEx(mask_closed, cv2.MORPH_OPEN, kernel)
```

Following is the output:

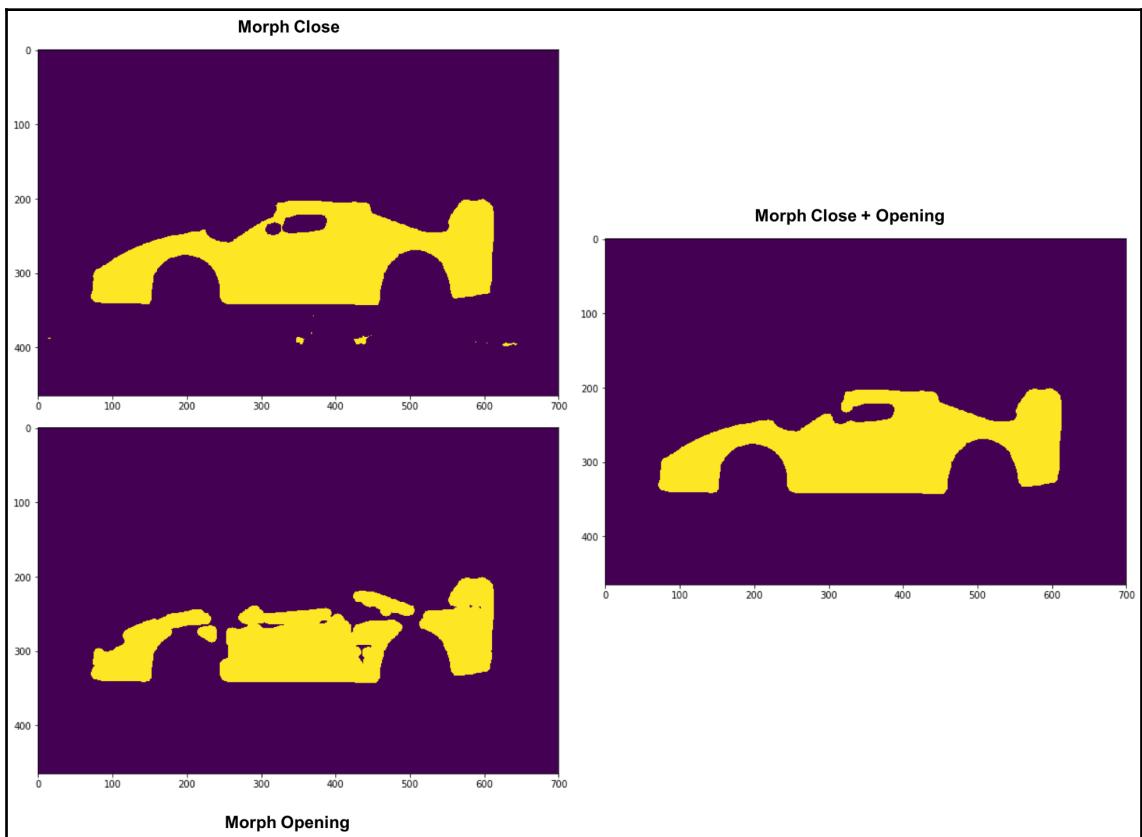


Figure 9.4: This figure illustrated the output of morphological close and open operation (left side) and we combine the both to get the final processed mask(right side).

In the preceding screenshot you can see (in the left part of the screenshot) how the morphological operation changes the structure of the mask and when combining both the operations (in the right side of the screenshot) you get a denoised cleaner structure.

Applying a mask

It's time to use the mask that we created to extract the object from the image. First, we will find the biggest contour using the helper function, which is the largest region of our object that we need to extract. Then apply the mask to the image and draw a circle bounding box on the extracted object:

```
# Extract biggest bounding box
big_contour, red_mask = find_biggest_contour(mask_clean)

# Apply mask
overlay = overlay_mask(red_mask, image)

# Draw bounding box
circled = circle_countour(overlay, big_contour)

show(circled)
```

Following is the output:

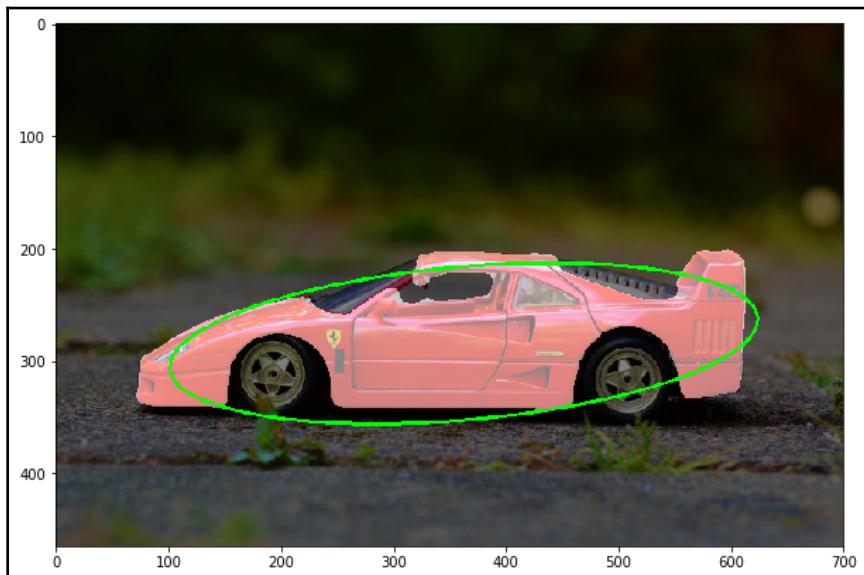


Figure 9.5: This figure shows that we have detected the red region (car body) from the image and plotted an ellipses around it.

Voila! So, we successfully extracted the image and also drew the bounding box around the object using simple image processing techniques.

Object detection using deep learning

In this section, we will learn how to build a world-class object detection module without much use of traditional handcrafting techniques. Here, we will be using the deep learning approach, which is powerful enough to extract features automatically from the raw image and then use those features for classification and detection purposes.

First, we will build an object detector using a pre-baked Python library that can use most of the state-of-the-art pre-trained models, and later on, we will learn how to implement a really fast and accurate object detector using YOLO architecture.

Quick implementation of object detection

Object detection saw an increase in adoption as a result of the industry trend towards deep learning after 2012. Accurate and increasingly fast models such as R-CNN, Fast-RCNN, Faster-RCNN, and RetinaNet, and fast yet highly accurate ones like SSD and YOLO are in production today. In this section, we will use fully-functional pre-baked feature extractors in a Python library that can be used in just a few lines of code. Also, we will touch base regarding the production-grade setup for the same.

So, let's do it.

Installing all the dependencies

This is the same drill that we performed in the previous chapters. First let's install all the dependencies. Here, we are using a Python module called ImageAI (<https://github.com/OlafewaMoses/ImageAI>), which is an effective way to start building your own object detection application from scratch in no time:

```
pip install tensorflow
pip install keras
pip install numpy
pip install scipy
pip install opencv-python
pip install pillow
pip install matplotlib
pip install h5py
```

```
# Here we are installing ImageAI
pip3 install
https://github.com/OlafenwaMoses/ImageAI/releases/download/2.0.2/imageai-2.
0.2-py3-none-any.whl
```

We will be using the Python 3.x environment to run this module.



For this implementation, we are going to use a pre-trained ResNet model that is trained on the COCO dataset (<http://cocodataset.org/#home>) (a large-scale object detection, segmentation, and captioning dataset). You can also use other pre-trained models such as follows:

- DenseNet-BC-121-32.h5 (<https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/DenseNet-BC-121-32.h5>) (31.7 MB)
- inception_v3_weights_tf_dim_ordering_tf_kernels.h5 (https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/inception_v3_weights_tf_dim_ordering_tf_kernels.h5) (91.7 MB)
- resnet50_coco_best_v2.0.1.h5 (https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/resnet50_coco_best_v2.0.1.h5) (146 MB)
- resnet50_weights_tf_dim_ordering_tf_kernels.h5 (https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/resnet50_weights_tf_dim_ordering_tf_kernels.h5) (98.1 MB)
- squeezezenet_weights_tf_dim_ordering_tf_kernels.h5 (https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/squeezezenet_weights_tf_dim_ordering_tf_kernels.h5) (4.83 MB)
- yolo-tiny.h5 (<https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/yolo-tiny.h5>) (33.9 MB)
- yolo.h5 (<https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/yolo.h5>): 237 MB

To get the dataset, use the following command:

```
wget  
https://github.com/OlafenwaMoses/ImageAI/releases/download/1.0/resnet50_coc  
o_best_v2.0.1.h5
```

Implementation

Now that we have all the dependencies and pre-trained models ready, we will implement a state-of-the-art object detection model. We will import the ImageAI's `ObjectDetection` class using the following code:

```
from imageai.Detection import ObjectDetection  
import os  
model_path = os.getcwd()
```

Then we create the instance for the `ObjectDetection` object and set the model type as `RetinaNet()`. Next, we set the part of the ResNet model that we downloaded and call the `loadModel()` function:

```
object_detector = ObjectDetection()  
object_detector.setModelTypeAsRetinaNet()  
object_detector.setModelPath( os.path.join(model_path ,  
"resnet50_coco_best_v2.0.1.h5"))  
object_detector.loadModel()
```

Once the model is loaded into the memory, we can feed a new image to the model, which can be of any popular image format, such as JPEG, PNG, and so on. Also, the function has no constraint on the size of the image, so, you can use any dimensional data and the model will handle it internally. We are using `detectObjectsFromImage()` to feed the input image. This method returns the image with some more information such as the bounding box coordinates of the detected object, the label of the detected object, and the confidence score.

Following are some images that are used as input into the model and to perform the object detection:

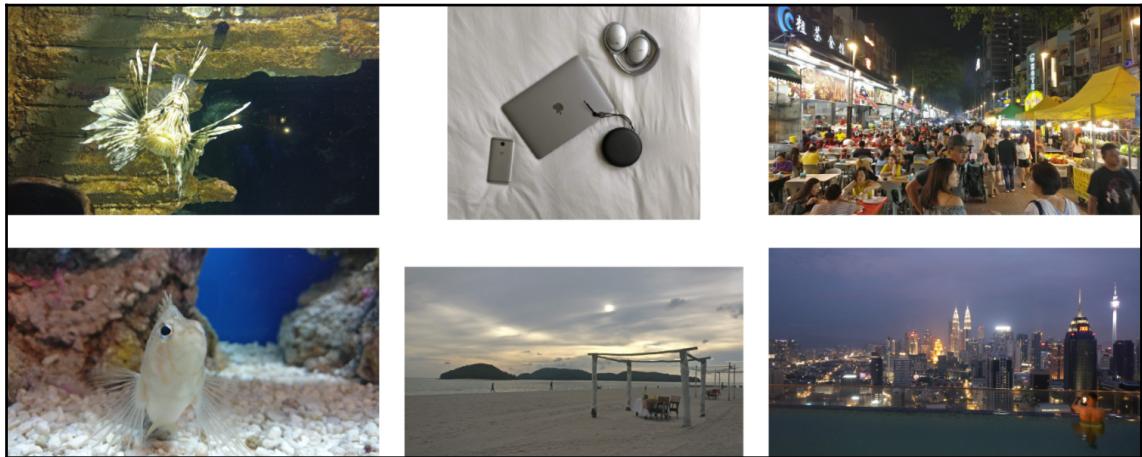


Figure 9.6: Since I was traveling to Asia (Malaysia/Langkawi) while writing this chapter, I decided to give it a shot and use some real images that I captured on the go.

The following code is used for inputting images into the model:

```
object_detections =  
object_detector.detectObjectsFromImage(input_image=os.path.join(model_path  
, "image.jpg"), output_image_path=os.path.join(model_path ,  
"imangenew.jpg"))
```

Further, we iterate over the `object_detection` object to read all the objects that the model predicted with the respective confidence score:

```
for eachObject in object_detections:  
    print(eachObject["name"] , " : " ,  
eachObject["percentage_probability"])
```

Following are how the results look:

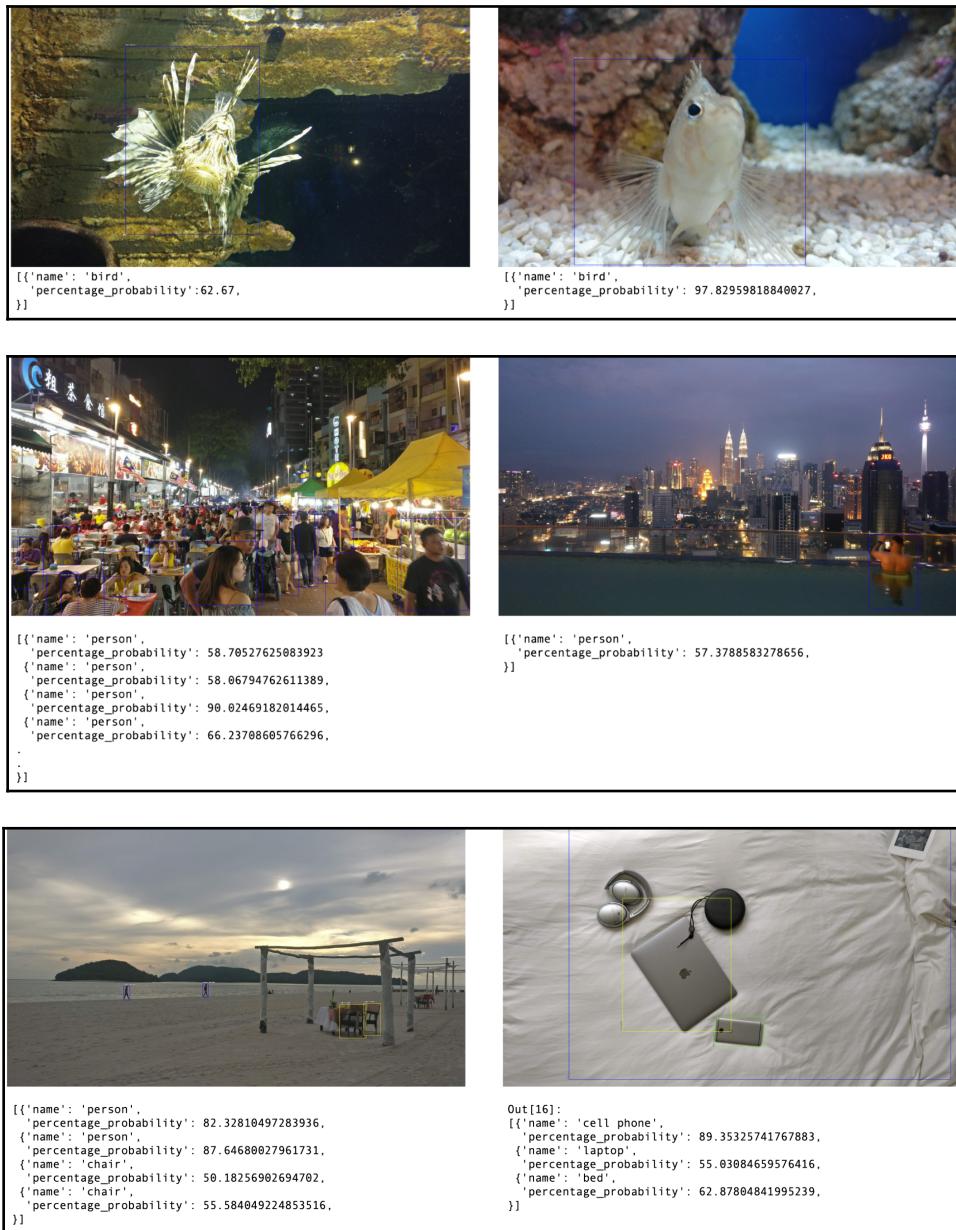


Figure 9.7: The results extracted from the object detection model with the bounding box around the detected object. Results contain the name of the object and the confidence score.

So, we can see that the pre-trained models performed well enough with very few lines of code.

Deployment

Now that we have all base code ready, let's deploy the `ObjectDetection` modules into production. In this section, we will write a RESTful service that will accept the image as an input and returns the detected object as a response.

We will define a `POST` function that accepts the image files with the PNG, JPG, JPEG, and GIF extensions. The uploaded image path is sent to the `ObjectDetection` module, which performs the detection and returns the following JSON results:

```
from flask import Flask, request, jsonify, redirect
import os , json
from imageai.Detection import ObjectDetection

model_path = os.getcwd()

PRE_TRAINED_MODELS = ["resnet50_coco_best_v2.0.1.h5"]

# Creating ImageAI objects and loading models

object_detector = ObjectDetection()
object_detector.setModelTypeAsRetinaNet()
object_detector.setModelPath( os.path.join(model_path ,
PRE_TRAINED_MODELS[0]))
object_detector.loadModel()
object_detections =
object_detector.detectObjectsFromImage(input_image='sample.jpg')

# Define model paths and the allowed file extention
UPLOAD_FOLDER = model_path
ALLOWED_EXTENSIONS = set(['png', 'jpg', 'jpeg', 'gif'])

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/predict', methods=['POST'])
```

```
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            print('No file part')
            return redirect(request.url)
        file = request.files['file']
        # if user does not select file, browser also
        # submit an empty part without filename
        if file.filename == '':
            print('No selected file')
            return redirect(request.url)
        if file and allowed_file(file.filename):
            filename = file.filename
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
            file.save(file_path)

    try:
        object_detections =
object_detector.detectObjectsFromImage(input_image=file_path)
    except Exception as ex:
        return jsonify(str(ex))
    resp = []
    for eachObject in object_detections :
        resp.append([eachObject ["name"],
                    round(eachObject ["percentage_probability"],3)
                    ])
    )

    return json.dumps(dict(enumerate(resp)))
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=4445)
```

Save the file as `object_detection_ImageAI.py` and execute the following command to run the web services:

```
python object_detection_ImageAI.py
```

Following is the output:

```
/anaconda2/envs/book/lib/python3.6/site-packages/h5py/_init_.py:36: FutureWarning: Conversion of the second argument of issubdtype from 'float' to 'np.floating' is deprecated. In future, it will be treated as 'np.float64 == np.dtype(float).type'.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
/anaconda2/envs/book/lib/python3.6/importlib/_bootstrap.py:219: RuntimeWarning: compiletime version 3.5 of module 'tensorflow.python.framework.fast_tensor_util' does not match runtime version 3.6
  return f(*args, **kwdss)
2018-09-26 10:16:33.769031: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
* Serving Flask app "object_detection_ImageAI" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:4445/ (Press CTRL+C to quit)
```

Figure 9.8: Output on the Terminal screen after successful execution of the web service.

In a separate Terminal, you can now try to call the API, as shown in the following command:

```
curl -X POST \
  http://0.0.0.0:4445/predict \
  -H 'content-type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW' \
  -F file=@/Users/rahulkumar/Downloads/IMG_1651.JPG
```

Following will be the response output:

```
{
  "0": ["person", 54.687],
  "1": ["person", 56.77],
  "2": ["person", 55.837],
  "3": ["person", 75.93],
  "4": ["person", 72.956],
  "5": ["bird", 81.139]
}
```

So, this was awesome; with just a few hours' work, you are ready with a production-grade object detection module that is something close to state-of-the-art.

Object Detection In Real-Time Using YOLOv2

A great advancement in object detection and classification was made possible with a process where You Only Look Once (YOLO) at an input image. In this single pass, the goal is to set the coordinates for the corners of the bounding box to be drawn around the detected object and to then classify the object with a regression model. This process is capable of avoiding false positives because it takes into account contextual information from the whole image, and not just a smaller section as in a regional proposal of earlier described methods. The **convolutional neural network (CNN)** as follows can pass over the image once, and therefore be fast enough to function in applications where real-time processing is a requirement.

YOLOv2 predicts an N number of bounding boxes and associates a confidence level for the classification of the object for each individual grid in an S -by- S grid that is established in the immediately preceding step.

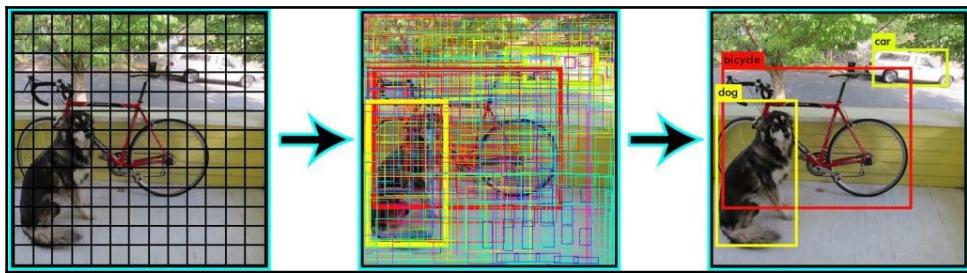


Figure 9.9: The overview of how YOLO works. The input image is divided into grids and then been sent into the detection process which results in lots of bounding boxes which is further been filtered by applying some thresholds.

The outcome of this process is to produce a total of S -by- S by N complement of boxes. For a great percentage of these boxes you'll get confidence scores that are quite low, and by applying a lower threshold (30% in this case), you can eliminate a majority of inaccurately classified objects as shown in the figure.

We will be using a pre-trained YOLOv2 model in this section for object detection and classification.

Preparing the dataset

In this part, we will explore the data preparation using the existing the COCO dataset and a custom dataset. If you want to train the YOLO model with lots of classes, then you can follow the instructions provided in the pre-existing part, or else if you want to build your custom object detector, then follow the instructions provided in the custom build section.

Using the pre-existing COCO dataset

For this implementation, we will be using the COCO dataset. This is a great resource dataset for training YOLOv2 to detect, segment, and caption images on a large scale. Download the dataset from <http://cocodataset.org> and run the following command in the terminal:

1. Get the training dataset:

```
wget http://images.cocodataset.org/zips/train2014.zip
```

2. Get the validation dataset:

```
wget http://images.cocodataset.org/zips/val2014.zip
```

3. Get the train and validation annotations:

```
wget  
http://images.cocodataset.org/annotations/annotations_trainval2014.  
zip
```

Now, let's convert the annotations in the COCO format to VOC format:

1. Install Baker:

```
pip install baker
```

2. Create the folders to store the images and annotations:

```
mkdir images annotations
```

3. Unzip train2014.zip and val2014.zip under the images folder:

```
unzip train2014.zip -d ./images/  
unzip val2014.zip -d ./images/
```

4. Unzip annotations_trainval2014.zip into annotations folder:

```
unzip annotations_trainval2014.zip -d ./annotations/
```

5. Create a folder to store the converted data:

```
mkdir output  
mkdir output/train  
mkdir output/val
```

```
python coco2voc.py create_annotations /TRAIN_DATA_PATH train  
/OUTPUT_FOLDER/train
```

```
python coco2voc.py create_annotations /TRAIN_DATA_PATH val
/OUTPUT_FOLDER/val
```

This is how the folder structure will look after the final transformation:

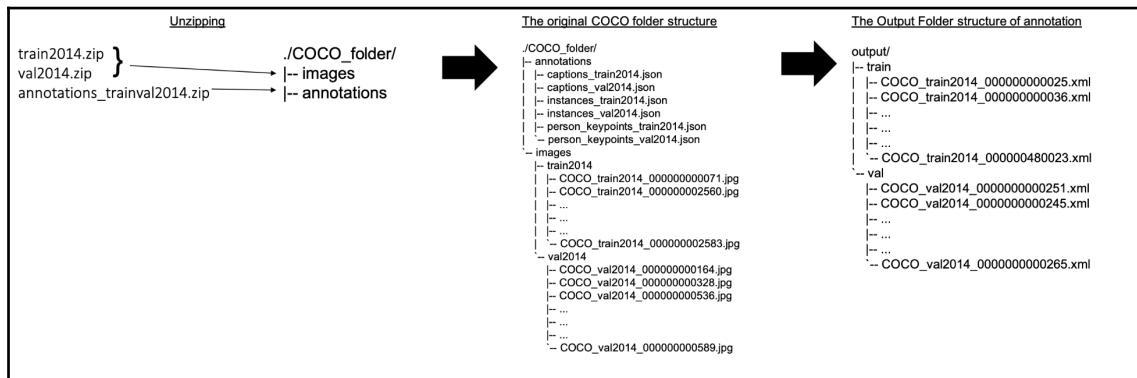


Figure 9.10: The illustration of the COCO data extraction and formatting process



This establishes a perfect correspondence between the image and the annotation. When the validation set is empty, we will use a ratio of eight to automatically split the training and validation sets.

The result is that we will have two folders, `./images` and `./annotation`, for the training purpose.

Using the custom dataset

Now, if you want to build an object detector for your specific use case, then you will need to scrape around 100–200 images from the web and annotate them. There are lots of annotation tools available online, such as LabelImg (<https://github.com/tzutalin/labelImg>) or Fast Image Data Annotation Tool (FIAT) (<https://github.com/christopher5106/FastAnnotationTool>).

For you to play around with the custom object detector, we have provided some sample images with respective annotations. Look into the repository folder called `Chapter09/yolo/new_class/`.

Each image has its respective annotations, as shown in the following picture:

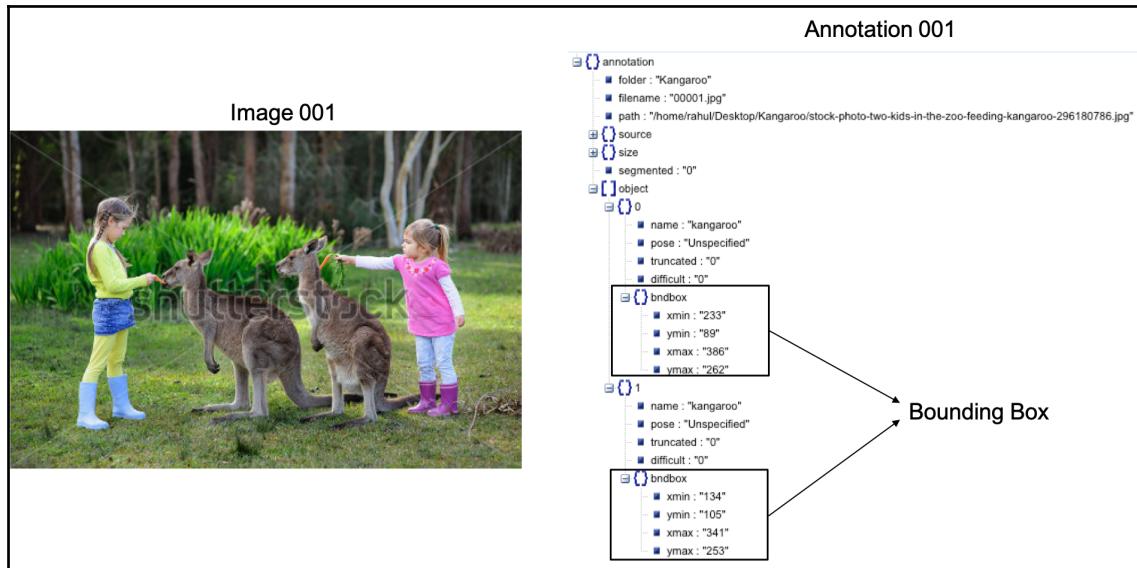


Figure 9.11: The relation between the image and the annotation which is shown here

Also, let's download the pre-trained weights from <https://pjreddie.com/darknet/yolo/>, which we will use to initialize our model, and which will train the custom object detector on top of these pretrained weights:

```
wget https://pjreddie.com/media/files/yolo.weights
```

Installing all the dependencies

We will be using the Keras APIs with a TensorFlow approach to create the YOLOv2 architecture. Let's import all the dependencies:

```
pip install keras tensorflow tqdm numpy cv2 imgaug
```

Following is the code for this:

```
from keras.models import Sequential, Model
from keras.layers import Reshape, Activation, Conv2D, Input, MaxPooling2D,
BatchNormalization, Flatten, Dense, Lambda
from keras.layers.advanced_activations import LeakyReLU
from keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard
from keras.optimizers import SGD, Adam, RMSprop
```

```
from keras.layers.merge import concatenate
import matplotlib.pyplot as plt
import keras.backend as K
import tensorflow as tf
import imgaug as ia
from tqdm import tqdm
from imgaug import augmenters as iaa
import numpy as np
import pickle
import os, cv2
from preprocessing import parse_annotation, BatchGenerator
from utils import WeightReader, decode_netout, draw_boxes

#Setting GPU configs
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = ""
```

It is always recommended to use GPUs to train any YOLO models.



Configuring the YOLO model

YOLO models are designed with the set of hyperparameter and some other configuration. This configuration defines the type of model to construct, as well as other parameters of the model such as the input image size and the list of anchors. You have two options at the moment: tiny YOLO and full YOLO. The following code defines the type of model to construct:

```
# List of object that YOLO model will learn to detect from COCO dataset

#LABELS = ['person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop sign',
'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag',
'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite',
'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis
racket', 'bottle', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog',
'pizza', 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'dining
table', 'toilet', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell
phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'book',
'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
```



```
# Label for the custom curated dataset.
```

```

LABEL = ['kangaroo']
IMAGE_H, IMAGE_W = 416, 416
GRID_H, GRID_W = 13 , 13
BOX = 5
CLASS = len(LABELS)
CLASS_WEIGHTS = np.ones(CLASS, dtype='float32')
OBJ_THRESHOLD = 0.3
NMS_THRESHOLD = 0.3
ANCHORS = [0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434,
7.88282, 3.52778, 9.77052, 9.16828]

NO_OBJECT_SCALE = 1.0
OBJECT_SCALE = 5.0
COORD_SCALE = 1.0
CLASS_SCALE = 1.0

BATCH_SIZE = 16
WARM_UP_BATCHES = 0
TRUE_BOX_BUFFER = 50

```

Configure the path of the pre-trained model and the images, as in the following code:

```

wt_path = 'yolo.weights'
train_image_folder = '/new_class/images/'
train_annot_folder = '/new_class/anno/'
valid_image_folder = '/new_class/images/'
valid_annot_folder = '/new_class/anno/'

```

Defining the YOLO v2 model

Now let's have a look at the model architecture of the YOLOv2 model:

```

# the function to implement the organization layer (thanks to
github.com/allanzelener/YAD2K)
def space_to_depth_x2(x):
    return tf.space_to_depth(x, block_size=2)
input_image = Input(shape=(IMAGE_H, IMAGE_W, 3))
true_boxes = Input(shape=(1, 1, 1, TRUE_BOX_BUFFER , 4))

# Layer 1
x = Conv2D(32, (3,3), strides=(1,1), padding='same', name='conv_1',
use_bias=False) (input_image)
x = BatchNormalization(name='norm_1') (x)
x = LeakyReLU(alpha=0.1) (x)
x = MaxPooling2D(pool_size=(2, 2)) (x)

# Layer 2

```

```

x = Conv2D(64, (3,3), strides=(1,1), padding='same', name='conv_2',
use_bias=False)(x)
x = BatchNormalization(name='norm_2')(x)
x = LeakyReLU(alpha=0.1)(x)
x = MaxPooling2D(pool_size=(2, 2))(x)

# Layer 3
# Layer 4
# Layer 23
# For the entire architecture, please refer to the yolo/Yolo_v2_train.ipynb
notebook here:
https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/blob/master/Module%2002/Chapter09/yolo/Yolo\_v2\_train.ipynb

```

Following is the output:

```

Total params: 50,983,561
Trainable params: 50,962,889
Non-trainable params: 20,672

```

Training the model

Following are the steps to train the model:

1. Load the weights that we downloaded and use them to initialize the model:

```

weight_reader = WeightReader(wt_path)
weight_reader.reset()
nb_conv = 23
for i in range(1, nb_conv+1):
    conv_layer = model.get_layer('conv_' + str(i))
    if i < nb_conv:
        norm_layer = model.get_layer('norm_' + str(i))
        size = np.prod(norm_layer.get_weights()[0].shape)

        beta = weight_reader.read_bytes(size)
        gamma = weight_reader.read_bytes(size)
        mean = weight_reader.read_bytes(size)
        var = weight_reader.read_bytes(size)

        weights = norm_layer.set_weights([gamma, beta, mean, var])
        if len(conv_layer.get_weights()) > 1:
            bias =
    weight_reader.read_bytes(np.prod(conv_layer.get_weights()[1].shape)
)

```

```

        kernel =
weight_reader.read_bytes(np.prod(conv_layer.get_weights()[0].shape)
)
kernel =
kernel.reshape(list(reversed(conv_layer.get_weights()[0].shape)))
kernel = kernel.transpose([2,3,1,0])
conv_layer.set_weights([kernel, bias])
else:
    kernel =
weight_reader.read_bytes(np.prod(conv_layer.get_weights()[0].shape)
)
kernel =
kernel.reshape(list(reversed(conv_layer.get_weights()[0].shape)))
kernel = kernel.transpose([2,3,1,0])
conv_layer.set_weights([kernel])

```

2. Randomize the weights of the last layer:

```

layer = model.layers[-4] # the last convolutional layer
weights = layer.get_weights()

new_kernel =
np.random.normal(size=weights[0].shape) / (GRID_H*GRID_W)
new_bias =
np.random.normal(size=weights[1].shape) / (GRID_H*GRID_W)

layer.set_weights([new_kernel, new_bias])

```

3. Generate the configurations as in the following code:

```

generator_config = {
    'IMAGE_H' : IMAGE_H,
    'IMAGE_W' : IMAGE_W,
    'GRID_H' : GRID_H,
    'GRID_W' : GRID_W,
    'BOX' : BOX,
    'LABELS' : LABELS,
    'CLASS' : len(LABELS),
    'ANCHORS' : ANCHORS,
    'BATCH_SIZE' : BATCH_SIZE,
    'TRUE_BOX_BUFFER' : 50,
}

```

4. Create a training and validation batch:

```
# Training batch data
train_imgs, seen_train_labels =
parse_annotation(train_annot_folder, train_image_folder,
labels=LABELS)
train_batch = BatchGenerator(train_imgs, generator_config,
norm=normalize)

# Validation batch data
valid_imgs, seen_valid_labels =
parse_annotation(valid_annot_folder, valid_image_folder,
labels=LABELS)
valid_batch = BatchGenerator(valid_imgs, generator_config,
norm=normalize, jitter=False)
```

5. Set early stop and checkpoint callbacks:

```
early_stop = EarlyStopping(monitor='val_loss',
                           min_delta=0.001,
                           patience=3,
                           mode='min',
                           verbose=1)

checkpoint = ModelCheckpoint('weights_coco.h5',
                            monitor='val_loss',
                            verbose=1,
                            save_best_only=True,
                            mode='min',
                            period=1)
```

6. Use the following code to train the model:

```
tb_counter = len([log for log in
os.listdir(os.path.expanduser('~/logs/')) if 'coco_' in log]) + 1
tensorboard = TensorBoard(log_dir=os.path.expanduser('~/logs/') +
'coco_' + '_' + str(tb_counter),
histogram_freq=0,
write_graph=True,
write_images=False)
```

```
optimizer = Adam(lr=0.5e-4, beta_1=0.9, beta_2=0.999,
epsilon=1e-08, decay=0.0)
#optimizer = SGD(lr=1e-4, decay=0.0005, momentum=0.9)
#optimizer = RMSprop(lr=1e-4, rho=0.9, epsilon=1e-08, decay=0.0)

model.compile(loss=custom_loss, optimizer=optimizer)

model.fit_generator(generator = train_batch,
                     steps_per_epoch = len(train_batch),
                     epochs = 100,
                     verbose = 1,
                     validation_data = valid_batch,
                     validation_steps = len(valid_batch),
                     callbacks = [early_stop, checkpoint,
                     tensorboard],
                     max_queue_size = 3)
```

Following is the output:

```
Epoch 1/2
11/11 [=====] - 315s 29s/step - loss: 3.6982 -
val_loss: 1.5416

Epoch 00001: val_loss improved from inf to 1.54156, saving model to
weights_coco.h5
Epoch 2/2
11/11 [=====] - 307s 28s/step - loss: 1.4517 -
val_loss: 1.0636

Epoch 00002: val_loss improved from 1.54156 to 1.06359, saving model to
weights_coco.h5
```

Following is the TensorBoard plots output for just two epochs:

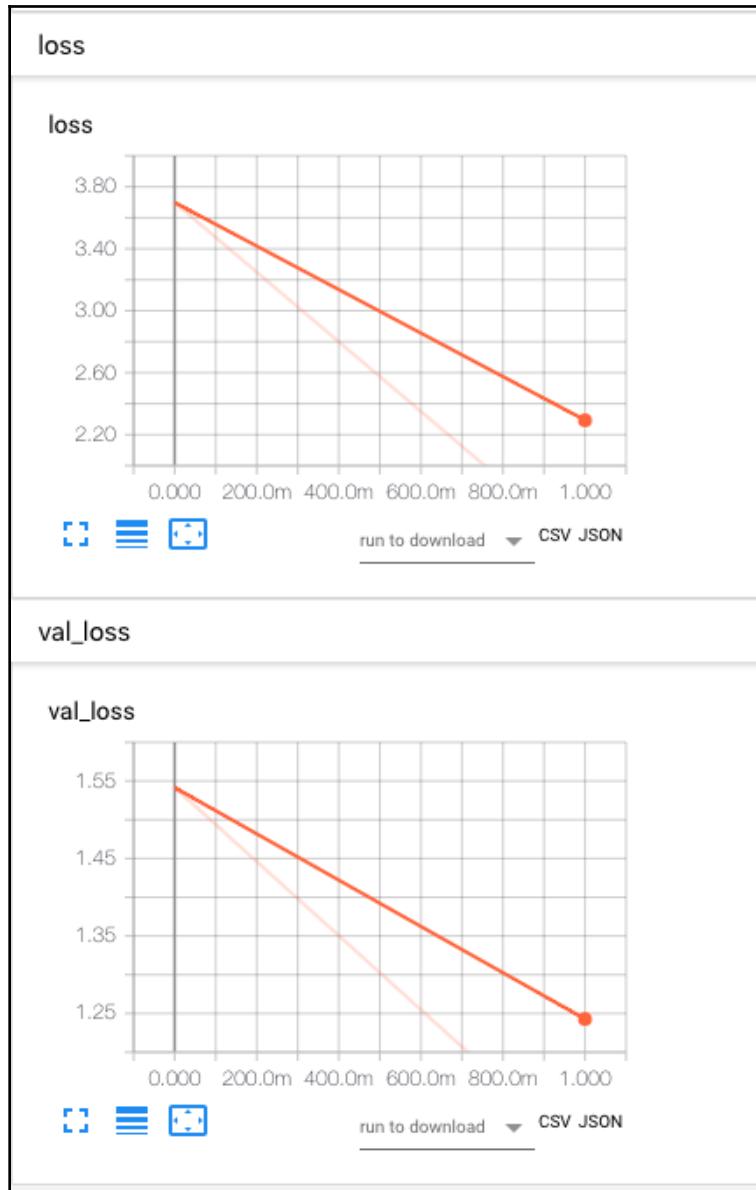


Figure 9.12: The figure represents the loss plots for 2 epochs

Evaluating the model

Once the training is complete, let's perform the prediction by feeding an input image into the model:

1. First we will load the model into the memory:

```
model.load_weights("weights_coco.h5")
```

2. Now set the test image path and read it:

```
input_image_path = "my_test_image.jpg"
image = cv2.imread(input_image_path)
dummy_array = np.zeros((1,1,1,1,TRUE_BOX_BUFFER,4))
plt.figure(figsize=(10,10))
```

3. Normalize the image:

```
input_image = cv2.resize(image, (416, 416))
input_image = input_image / 255.
input_image = input_image[:, :, ::-1]
input_image = np.expand_dims(input_image, 0)
```

4. Make a prediction:

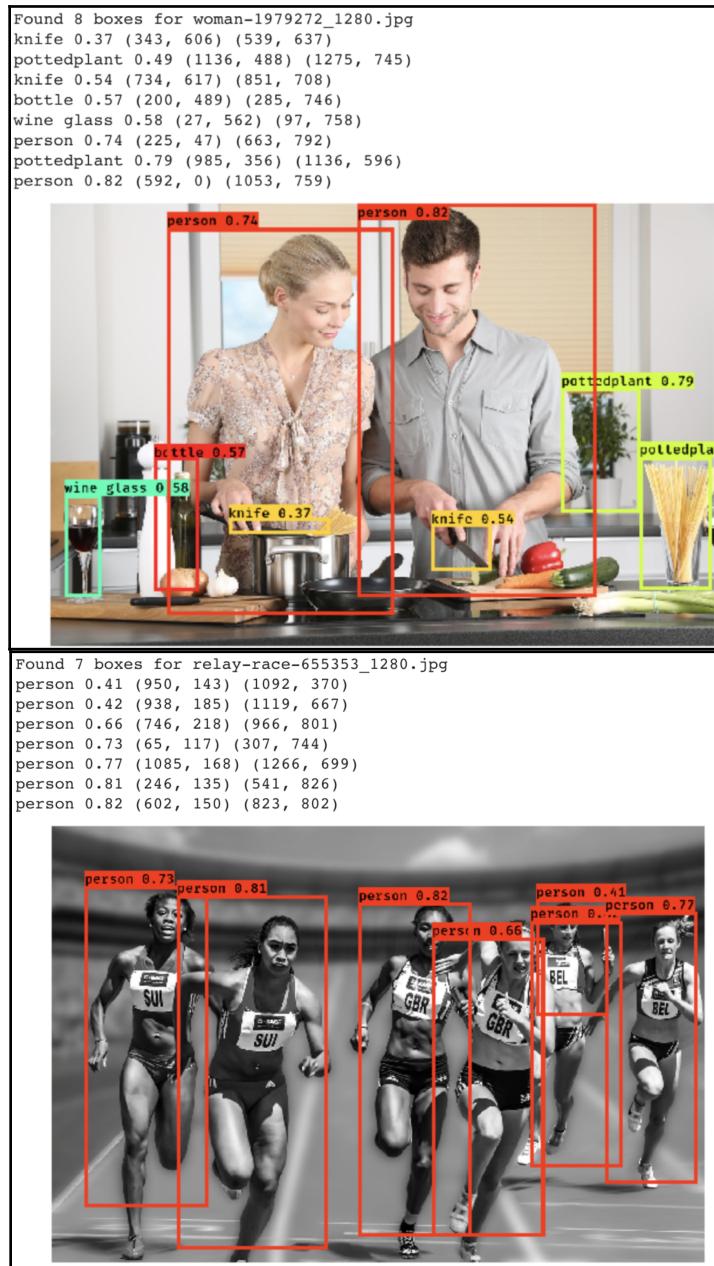
```
netout = model.predict([input_image, dummy_array])

boxes = decode_netout(netout[0],
                      obj_threshold=OBJ_THRESHOLD,
                      nms_threshold=NMS_THRESHOLD,
                      anchors=ANCHORS,
                      nb_class=CLASS)

image = draw_boxes(image, boxes, labels=LABELS)

plt.imshow(image[:, :, ::-1]); plt.show()
```

So, here are some of the results:





Congratulations—you have developed a state-of-the-art object detector that is very fast and reliable.

We learned about building a world class object detection model using YOLO architecture and the results seems to be very promising. Now you can also deploy the same on other mobile devices or Raspberry Pi.

Image segmentation

Image segmentation is the process of categorizing what is in a picture at a pixel level. For example, if you were given a picture with a person in it, separating the person from the image is known as segmentation and is done using pixel-level information.

We will be using the COCO dataset for image segmentation.

Following is what you should do before executing any of the SegNet scripts:

```
cd SegNet
wget http://images.cocodataset.org/zips/train2014.zip
mkdir images
unzip train2014.zip -d images
```

When executing SegNet scripts, make sure that your present working directory is SegNet.

Importing all the dependencies

Make sure to restart the session before proceeding forward.

We will be using numpy, pandas, keras, pylab, skimage, matplotlib, and pycocotools, as in the following code:

```
from __future__ import absolute_import
from __future__ import print_function

import pylab
import numpy as np
import pandas as pd
import skimage.io as io
import matplotlib.pyplot as plt

from pycocotools.coco import COCO
pylab.rcParams['figure.figsize'] = (8.0, 10.0)
import cv2

import keras.models as models, Sequential
from keras.layers import Layer, Dense, Dropout, Activation, Flatten,
Reshape, Permute
from keras.layers import Conv2D, MaxPool2D, UpSampling2D, ZeroPadding2D
from keras.layers import BatchNormalization

from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from keras.optimizers import Adam
```

```
import keras
keras.backend.set_image_dim_ordering('th')

from tqdm import tqdm
import itertools
%matplotlib inline
```

Exploring the data

We will start off by defining the location of the annotation file we will be using for image segmentation, and then we will initialize the COCO API:

```
# set the location of the annotation file associated with the train images
annFile='annotations/instances_train2014.json'

# initialize COCO api with
coco = COCO(annFile)
```

Following should be the output:

```
loading annotations into memory...
Done (t=12.84s)
creating index...
index created!
```

Images

Since we are building a binary segmentation model, let us consider the images from the images/train2014 folder that are only tagged with the person label so that we can segment the person out of the image. The COCO API provides us with easy-to-use methods, two of which are the getCatIds and getImgIds. The following snippet will help us extract the image IDs of all the images with the label person tagged to it:

```
# extract the category ids using the label 'person'
catIds = coco.getCatIds(catNms=['person'])

# extract the image ids using the catIds
imgIds = coco.getImgIds(catIds=catIds )

# print number of images with the tag 'person'
print("Number of images with the tag 'person' :" ,len(imgIds))
```

This should be the output:

```
Number of images with the tag 'person' : 45174
```

Now let us use the following snippet to plot an image:

```
# extract the details of image with the image id
img = coco.loadImgs(imgIds[2])[0]
print(img)

# load the image using the location of the file listed in the image
# variable
I = io.imread('images/train2014/' + img['file_name'])

# display the image
plt.imshow(I)
```

Following should be the output:

```
{'height': 426, 'coco_url':
'http://images.cocodataset.org/train2014/COCO_train2014_000000524291.jpg',
'date_captured': '2013-11-18 09:59:07', 'file_name':
'COCO_train2014_000000524291.jpg', 'flickr_url':
'http://farm2.staticflickr.com/1045/934293170_d1b2cc58ff_z.jpg', 'width':
640, 'id': 524291, 'license': 3}
```

We get the following picture as an output:



Figure 9.13: The plot representation a sample image from the dataset.

In the previous code snippet, we feed in an image ID to the `loadImgs` method of COCO to extract the details of the image it corresponds to. If you look at the output of the `img` variable, one of the keys listed is the `file_name` key. This key holds the name of the image located in the `images/train2014/` folder.

Then we read the image using the `imread` method of the `io` module we have already imported and plot it using `matplotlib.pyplot`.

Annotations

Now let us load the annotation corresponding to the previous picture and plot the annotation on top of the picture. The `coco.getAnnIds()` function helps load the annotation info of an image using its ID. Then, with the help of the `coco.loadAnns()` function, we load the annotations and plot it using the `coco.showAnns()` function. It is important that you first plot the image and then perform the annotation operations as shown in the following code snippet:

```
# display the image
plt.imshow(I)

# extract the annotation id
annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=None)

# load the annotation
anns = coco.loadAnns(annIds)

# plot the annotation on top of the image
coco.showAnns(anns)
```

Following should be the output:

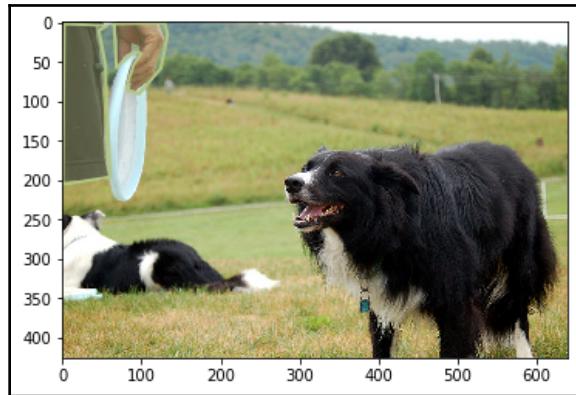


Figure 9.14: Visualizing annotation on an image

To be able to obtain the annotation label array, use the `coco.annToMask()` function as shown in the following code snippet. This array will help us form the segmentation target:

```
# build the mask for display with matplotlib
mask = coco.annToMask(anns[0])

# display the mask
plt.imshow(mask)
```

Following should be the output:

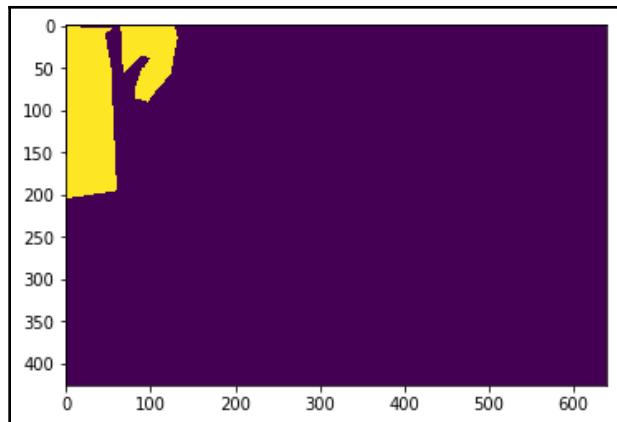


Figure 9.15: Visualizing just the annotation

Preparing the data

Let us now define a `data_list()` function that will automate the process of loading an image and its segmentation array into memory and resize them to the shape of 360*480 using OpenCV. This function returns two lists containing images and segmentation array:

```
def data_list(imgIds, count = 12127, ratio = 0.2):
    """Function to load image and its target into memory."""
    img_lst = []
    lab_lst = []

    for x in tqdm(imgIds[0:count]):
        # load image details
        img = coco.loadImgs(x)[0]
        # read image
        I = io.imread('images/train2014/' + img['file_name'])
        if len(I.shape) < 3:
            continue
        # load annotation information
        annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds,
                               iscrowd=None)
        # load annotation
        anns = coco.loadAnns(annIds)
        # prepare mask
        mask = coco.annToMask(anns[0])
        # This condition makes sure that we select images having only one
person
        if len(np.unique(mask)) == 2:
            # Next condition selects images where ratio of area covered by
the
            # person to the entire image is greater than the ratio
parameter
            # This is done to not have large class imbalance
            if (len(np.where(mask>0)[0])/len(np.where(mask>=0)[0])) > ratio
            :
                # If you check, generated mask will have 2 classes i.e 0
and 2
                # (0 - background/other, 1 - person).
                # to avoid issues with cv2 during the resize operation
                # set label 2 to 1, making label 1 as the person.
                mask[mask==2] = 1
                # resize image and mask to shape (480, 360)
                I= cv2.resize(I, (480,360))
                mask = cv2.resize(mask, (480,360))

                # append mask and image to their lists
                img_lst.append(I)
```

```
        lab_lst.append(mask)
    return (img_lst, lab_lst)

# get images and their labels
img_lst, lab_lst = data_list(imgIds)

print('Sum of images for training, validation and testing :', len(img_lst))
print('Unique values in the labels array :', np.unique(lab_lst[0]))
```

Following should be the output:

```
Sum of images for training, validation and testing : 1997
Unique values in the labels array : [0 1]
```

Normalizing the image

First, let's define the `make_normalize()` function, which accepts an image and performs the histogram normalization operation on it. The return object is a normalized array:

```
def make_normalize(img):
    """Function to histogram normalize images."""
    norm_img = np.zeros((img.shape[0], img.shape[1], 3), np.float32)

    b=img[:, :, 0]
    g=img[:, :, 1]
    r=img[:, :, 2]

    norm_img[:, :, 0]=cv2.equalizeHist(b)
    norm_img[:, :, 1]=cv2.equalizeHist(g)
    norm_img[:, :, 2]=cv2.equalizeHist(r)

    return norm_img

plt.figure(figsize = (14,5))
plt.subplot(1,2,1)
plt.imshow(img_lst[9])
plt.title(' Original Image')
plt.subplot(1,2,2)
plt.imshow(make_normalize(img_lst[9]))
plt.title(' Histogram Normalized Image')
```

Following should be the output:

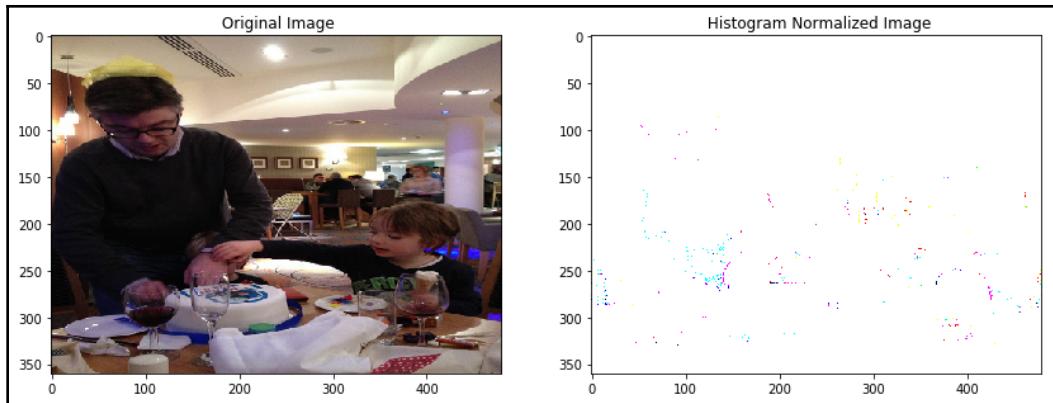


Figure 9.16: Before and After histogram normalization on an image

In the preceding screenshot, we see the original picture on the left, which is very visible, and on the right we see the normalized picture, which is not at all visible.

Encoding

With the `make_normalize()` function defined, let's now define a `make_target` function. This function accepts the segmentation array of shape (360,480) and then returns a segmentation target of shape (360,480,2). In the target, channel 0 represents the background and will have 1 in locations that represent the background in the image and zero elsewhere. Channel 1 represents the person and will have 1 in locations that represent the person in the image and 0 elsewhere. The following code implements the function:

```
def make_target(labels):
    """Function to one hot encode targets."""
    x = np.zeros([360,480,2])
    for i in range(360):
        for j in range(480):
            x[i,j,labels[i][j]]=1
    return x

plt.figure(figsize = (14,5))
plt.subplot(1,2,1)
plt.imshow(make_target(lab_lst[0])[:, :, 0])
plt.title('Background')
plt.subplot(1,2,2)
plt.imshow(make_target(lab_lst[0])[:, :, 1])
plt.title('Person')
```

Following should be the output:

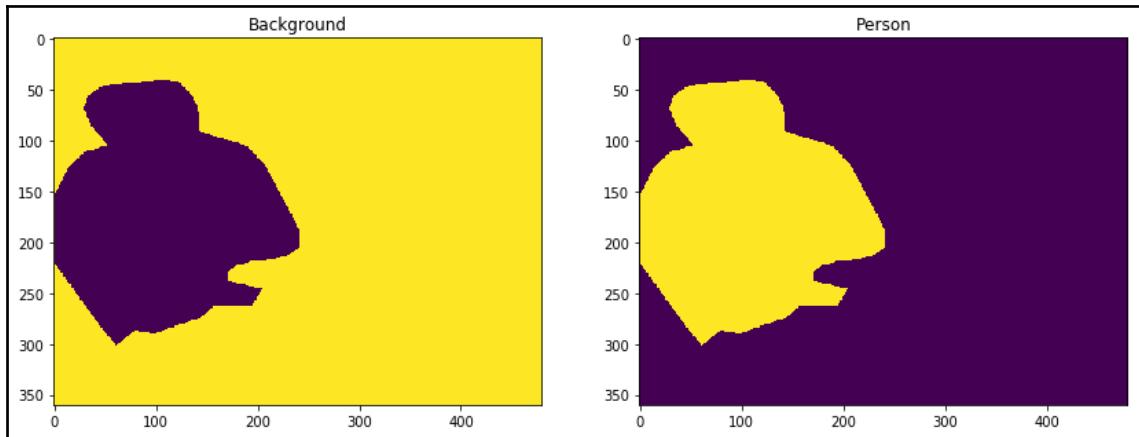


Figure 9.17: Visualizing the encoded target arrays

Model data

We will now define a function called `model_data()` that accepts a list of images and a list of labels. This function will apply the `make_normalize()` function on each image for the purpose of normalizing, and it will apply the `make_encode()` function on each label/segmentation array to obtain the encoded array.

The return of this function is two lists, one containing the normalized images and the other containing the corresponding target arrays:

```
def model_data(images, labels):
    """Function to perform normalize and encode operation on each image."""
    # empty label and image list
    array_lst = []
    label_lst = []
    # apply normalize function on each image and encoding function on each
    label
    for x,y in tqdm(zip(images, labels)):
        array_lst.append(np.rollaxis(normalized(x), 2))
        label_lst.append(make_target(y))
    return np.array(array_lst), np.array(label_lst)

# Get model data
train_data, train_lab = model_data(img_lst, lab_lst)
```

```
flat_image_shape = 360*480

# reshape target array
train_label = np.reshape(train_lab, (-1, flat_image_shape, 2))

# test data
test_data = test_data[1900:]
# validation data
val_data = train_data[1500:1900]
# train data
train_data = train_data[:1500]

# test label
test_label = test_label[1900:]
# validation label
val_label = train_label[1500:1900]
# train label
train_label = train_label[:1500]
```

In the preceding snippet, we have also split the data into train, test, and validation sets, with the train set containing 1500 data points, the validation set containing 400 data points, and the test set containing 97 data points.

Defining hyperparameters

The following are some of the defined hyperparameters that we will be using throughout the code, and they are totally configurable:

```
# define optimizer
optimizer = Adam(lr=0.002)

# input shape to the model
input_shape=(3, 360, 480)

# training batchsize
batch_size = 6

# number of training epochs
nb_epoch = 60
```



To learn more about optimizers and their APIs in Keras, visit <https://keras.io/optimizers/>. Reduce `batch_size` if you get a resource exhaustion error with respect to the GPU.



Experiment with different learning rates, optimizers, and batch_size to see how these factors affect the quality of your model, and if you get better results, show them to the deep learning community.

Define SegNet

For the purpose of image segmentation, we will build a SegNet model, which is very similar to the autoencoder we built in [Chapter 19, Handwritten Digits Classification Using ConvNets](#), as shown:

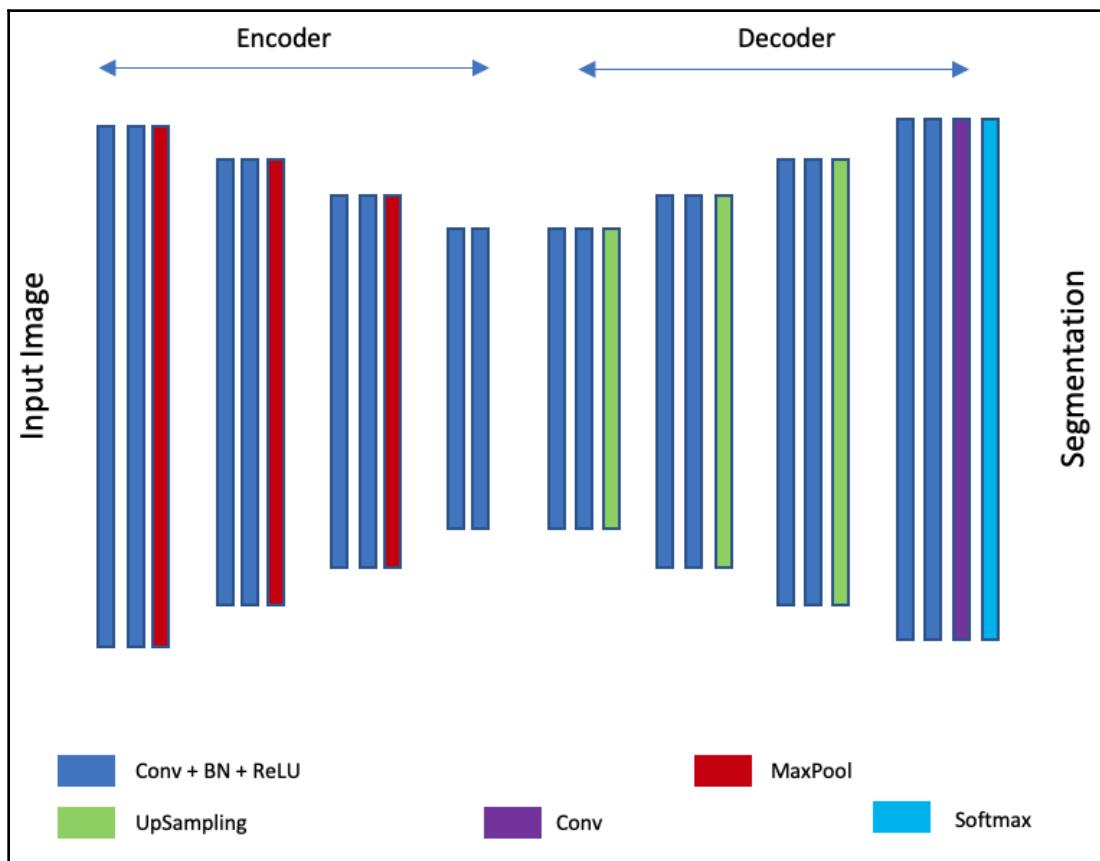


Figure 9.18: SegNet architecture used in this chapter

The SegNet model we'll define will accept $(3,360, 480)$ images as input with $(172800, 2)$ segmentation arrays as the targets, and it will have the following characteristics in the encoder:

- The first layer is a Convolution 2D layer with 64 filters of size $3*3$, with activation as `relu`, followed by batch normalization, followed by downsampling with MaxPooling2D of size $2*2$.
- The second layer is a Convolution 2D layer with 128 filters of size $3*3$, with activation as `relu`, followed by batch normalization, followed by downsampling with MaxPooling2D of size $2*2$.
- The third layer is a Convolution 2D layer with 256 filters of size $3*3$, with activation as `relu`, followed by batch normalization, followed by downsampling with MaxPooling2D of size $2*2$.
- The fourth layer is again a Convolution 2D layer with 512 filters of size $3*3$, with activation as `relu`, followed by batch normalization.

And the model will have the following characteristics in the decoder:

- The first layer is a Convolution 2D layer with 512 filters of size $3*3$, with activation as `relu`, followed by batch normalization, followed by downsampling with UpSampling2D of size $2*2$.
- The second layer is a Convolution 2D layer with 256 filters of size $3*3$, with activation as `relu`, followed by batch normalization, followed by downsampling with UpSampling2D of size $2*2$.
- The third layer is a Convolution 2D layer with 128 filters of size $3*3$, with activation as `relu`, followed by batch normalization, followed by downsampling with UpSampling2D of size $2*2$.
- The fourth layer is a Convolution 2D layer with 64 filters of size $3*3$ with activation as `relu`, followed by batch normalization.
- The fifth layer is a Convolution 2D layer with 2 filters of size $1*1$, followed by Reshape, Permute and a softmax as activation layer for predicting scores.

The model is described with the following code:

```
model = Sequential()
# Encoder
model.add(Layer(input_shape=input_shape))
model.add(ZeroPadding2D())
model.add(Conv2D(filters=64, kernel_size=(3,3), padding='valid',
activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))

model.add(ZeroPadding2D())
model.add(Conv2D(filters=128, kernel_size=(3,3), padding='valid',
activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))

model.add(ZeroPadding2D())
model.add(Conv2D(filters=256, kernel_size=(3,3), padding='valid',
activation='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))

model.add(ZeroPadding2D())
model.add(Conv2D(filters=512, kernel_size=(3,3), padding='valid',
activation='relu'))
model.add(BatchNormalization())

# Decoder
# For the remaining part of this section of the code refer to the
segnet.ipynb file in the SegNet folder. Here is the github link:
https://github.com/PacktPublishing/Python-Beginners-Guide-to-Artificial-Intelligence/tree/master/Module%2002/Chapter09
```

Compiling the model

With the model defined, compile the model with 'categorical_crossentropy' as loss and optimizer as Adam, as defined by the optimizer variable in the hyperparameters section. We will also define ReduceLROnPlateau to reduce the learning rate as needed when training, as follows:

```
# compile model
model.compile(loss="categorical_crossentropy", optimizer=Adam(lr=0.002),
metrics=["accuracy"])

# use ReduceLROnPlateau to adjust the learning rate
reduceLROnPlat = ReduceLROnPlateau(monitor='val_acc', factor=0.75,
patience=5,
min_delta=0.005, mode='max', cooldown=3, verbose=1)

callbacks_list = [reduceLROnPlat]
```

Fitting the model

With the model compiled, we will now fit the model on the data using the `fit` method of the model. Here, since we are training on a small set of data, it is important to set the parameter `shuffle` to `True` so that the images are shuffled after each epoch:

```
# fit the model
history = model.fit(train_data, train_label, callbacks=callbacks_list,
batch_size=batch_size, epochs=nb_epoch,
verbose=1, shuffle = True, validation_data = (val_data,
val_label))
```

This should be the output:

```
Train on 1500 samples, validate on 400 samples
Epoch 1/60
1500/1500 [=====] - 351s 234ms/step - loss: 0.6338 - acc: 0.6610 - val_loss: 0.6201
- val_acc: 0.6631
Epoch 2/60
1500/1500 [=====] - 339s 226ms/step - loss: 0.6016 - acc: 0.6678 - val_loss: 0.6049
- val_acc: 0.6673
Epoch 3/60
1500/1500 [=====] - 337s 225ms/step - loss: 0.5941 - acc: 0.6751 - val_loss: 0.6468
- val_acc: 0.5983
```

Figure 9.19: Training output

The following shows the accuracy and loss plots:

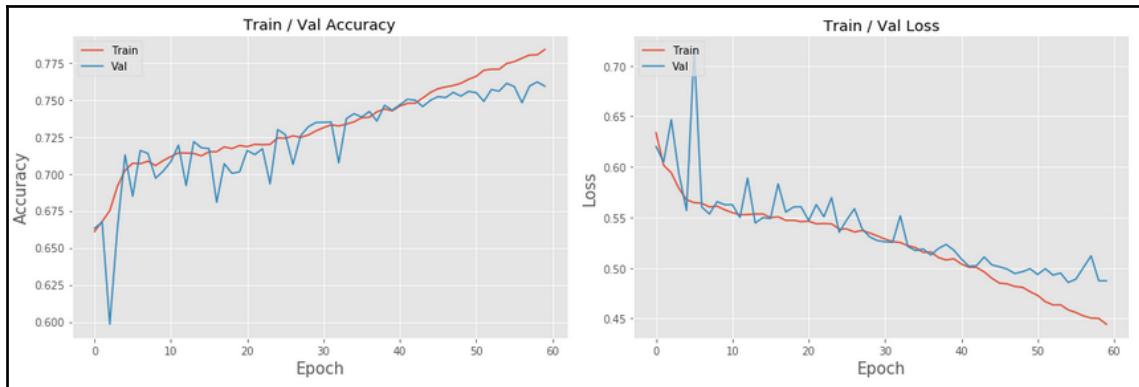


Figure 9.20: Plot showing training progression

Testing the model

With the model trained, evaluate the model on test data, as in the following:

```
loss, acc = model.evaluate(test_data, test_label)
print('Loss :', loss)
print('Accuracy :', acc)
```

This should be the output:

```
97/97 [=====] - 7s 71ms/step
Loss : 0.5390811630131043
Accuracy : 0.7633129960482883
```

We see that the SegNet model we built has a loss of 0.539 and accuracy of 76.33 on test images.

Let's plot the test images and their corresponding generated segmentations to understand model learning:

```
for i in range(3):
    plt.figure(figsize = (10,3))
    plt.subplot(1,2,1)
    plt.imshow(img_lst[1900+i])
    plt.title('Input')
    plt.subplot(1,2,2)
    plt.imshow(model.predict_classes(test_data[i:(i+1)*1]).reshape(360,480))
    plt.title('Segmentation')
```

Following should be the output:

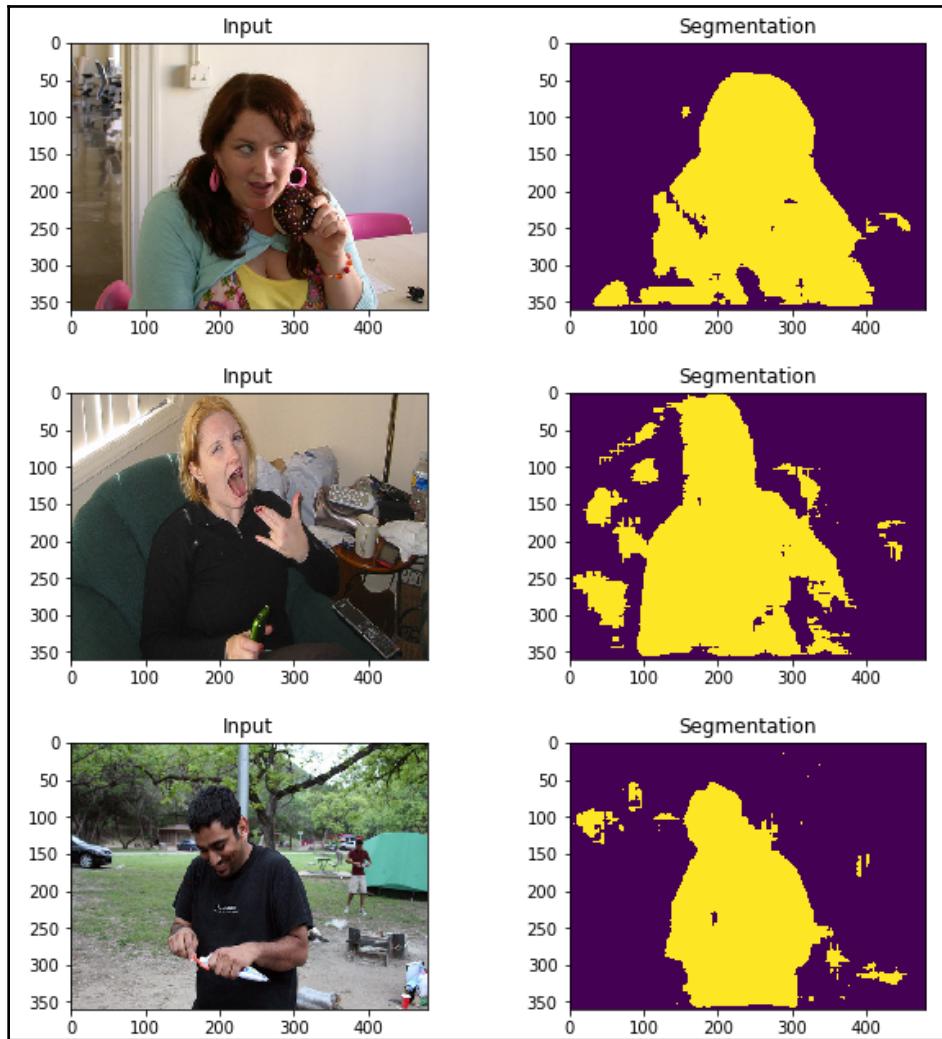


Figure 9.21: Segmentation generated on test images

From the preceding figure, we see that the model was able to segment the person from the image.

Conclusion

The first part of the project was to build an object detection classifier using YOLO architecture in Keras.

The second part of the project was to build a binary image segmentation model on COCO images that contain just a person, aside from the background. The goal was to build a good enough model to segment out the person from the background in the image.

The model we build by training on 1500 images, each of shape 360*480*3, has an accuracy of 79% on train data, and 78% on validation and test data. The model is successfully able to segment the person in the image, but the borders of the segmentations are slightly off from where they should be. This is due to using a small training set. Considering the number of images used for training, the model did a good job of segmenting.

There are more images available in this dataset that can be used for training, and it might take over a day to train on all of them using a Nvidia Tesla K80 GPU, but doing so will give you really good segmentation.

Summary

In the first part of this chapter, we learnt how to build a RESTful service for object detection using an existing classifier, and we also learned to build an accurate object detector using the YOLO architecture object detection classifier using Keras, while also implementing transfer learning. In the second part of the chapter, we understood what image segmentation is and built an image segmentation model on images from the COCO dataset. We also tested the performance of the object detector and the image segmenter on test data, and determined that we succeeded in achieving the goal.

21

Building Face Recognition Using FaceNet

In the previous chapter, we learned how to detect objects in an image. In this chapter, we will look into a specific use case of object detection—face recognition. Face recognition is a combination of two major operations: face detection, followed by face classification.

The (hypothetical) client that provides our business use case for us in this project is a high-performance computing data center Tier III, certified for sustainability. They have designed the facility to meet the very highest standards for protection against natural disasters, with many redundant systems.

The facility currently has ultra-high security protocols in place to prevent malicious, man-made disasters, and they are looking to augment their security profile with facial recognition for access to secure areas throughout the facility.

The stakes are high, as the servers they house and maintain process some of the most sensitive, valuable, and influential data in the world:



This facial recognition system would need to be able to accurately identify not only their own employees, but employees of their clients, who occasionally tour the data center for inspection.

They have asked us to provide a POC for this intelligence-based capability, for review and later inclusion throughout their data center.

So, in this chapter, we will learn how to build a world-class face recognition system. We will define the pipeline as follows:

1. **Face detection:** First, look at an image and find all the possible faces in it
2. **Face extraction:** Second, focus on each face image and understand it, for example if it is turned sideways or badly lit
3. **Feature extraction:** Third, extract unique features from the faces using convolutional neural networks (CNNs)
4. **Classifier training:** Finally, compare the unique features of that face to all the people already known, to determine the person's name

You will learn the main ideas behind each step, and how to build your own facial recognition system in Python using the following deep-learning technologies:

- **dlib** (<http://dlib.net/>): Provides a library that can be used for facial detection and alignment.
- **OpenFace** (<https://cmusatyalab.github.io/openface/>): A deep-learning facial recognition model, developed by Brandon Amos *et al* (<http://bamos.github.io/>). It is able to run on real-time mobile devices as well.
- **FaceNet** (<https://arxiv.org/abs/1503.03832>): A CNN architecture that is used for feature extraction. For a loss function, FaceNet uses triplet loss. Triplet loss relies on minimizing the distance from positive examples, while maximizing the distance from negative examples.

Setup environment

Since setup can get very complicated and take a long time, which is not on the agenda for this chapter, we will be building a Docker image that contains all the dependencies, including dlib, OpenFace, and FaceNet.

Getting the code

Fetch the code that we will use to build face recognition from the repository:

```
git clone https://github.com/PacktPublishing/Python-Deep-Learning-Projects  
cd Chapter10/
```

Building the Docker image

Docker is a container platform that simplifies deployment. It solves the problem of installing software dependencies onto different server environments. If you are new to Docker, you can read more at <https://www.docker.com/>.

To install Docker on Linux machines, run the following command:

```
curl https://get.docker.com | sh
```

For other systems such as macOS and Windows, visit <https://docs.docker.com/install/>. You can skip this step if you already have Docker installed.

Once Docker is installed, you should be able to use the docker command in the Terminal, as follows:

```
[rahulkumar@... ~] $ docker

Usage: docker COMMAND

A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default "/home/rahulkumar/.docker")
  -D, --debug          Enable debug mode
  -H, --host list      Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
  --tls                Use TLS; implied by --tlsv1
  --tlscacert string   Trust certs signed only by this CA (default "/home/rahulkumar/.docker/ca.pem")
  --tlscert string     Path to TLS certificate file (default "/home/rahulkumar/.docker/cert.pem")
  --tlskey string      Path to TLS key file (default "/home/rahulkumar/.docker/key.pem")
  --tlsv1               Use TLS and verify the remote
  -v, --version         Print version information and quit

Management Commands:
  checkpoint  Manage checkpoints
  config       Manage Docker configs
  container    Manage containers
  image        Manage images
  network     Manage networks
  node         Manage Swarm nodes
  plugin       Manage plugins
  secret       Manage Docker secrets
  service      Manage services
  swarm        Manage Swarm
  system       Manage Docker
  trust        Manage trust on Docker images
  volume       Manage volumes

Commands:
  attach        Attach local standard input, output, and error streams to a running container
  build         Build an image from a Dockerfile
  commit        Create a new image from a container's changes
  cp            Copy files/folders between a container and the local filesystem
  create        Create a new container
  deploy        Deploy a new stack or update an existing stack
  diff          Inspect changes to files or directories on a container's filesystem
  events        Get real time events from the server
  exec          Run a command in a running container
  export        Export a container's filesystem as a tar archive
  history      Show the history of an image
  images        List images
  import        Import the contents from a tarball to create a filesystem image
  info          Display system-wide information
  inspect      Return low-level information on Docker objects
  kill          Kill one or more running containers
  load          Load an image from a tar archive or STDIN
  login         Log in to a Docker registry
  logout        Log out from a Docker registry
  logs          Fetch the logs of a container
  pause         Pause all processes within one or more containers
  port          List port mappings or a specific mapping for the container
  ps            List containers
  run           Run a command in a new container
```

Now we will create a `docker` file that will install all the dependencies, including OpenCV, dlib, and TensorFlow.

```
#Dockerfile for our env setup
FROM tensorflow/tensorflow:latest

RUN apt-get update -y --fix-missing
RUN apt-get install -y ffmpeg
RUN apt-get install -y build-essential cmake pkg-config \
    libjpeg8-dev libtiff5-dev libjasper-dev libpng12-dev \
    libavcodec-dev libavformat-dev libswscale-dev libv4l-
dev \
    libxvidcore-dev libx264-dev \
    libgtk-3-dev \
    libatlas-base-dev gfortran \
    libboost-all-dev \
    python3 python3-dev python3-numpy

RUN apt-get install -y wget vim python3-tk python3-pip

WORKDIR /
RUN wget -O opencv.zip https://github.com/Itseez/opencv/archive/3.2.0.zip \
    && unzip opencv.zip \
    && wget -O opencv_contrib.zip \
https://github.com/Itseez/opencv_contrib/archive/3.2.0.zip \
    && unzip opencv_contrib.zip

# install opencv3.2
RUN cd /opencv-3.2.0/ \
    && mkdir build \
    && cd build \
    && cmake -D CMAKE_BUILD_TYPE=RELEASE \
        -D INSTALL_C_EXAMPLES=OFF \
        -D INSTALL_PYTHON_EXAMPLES=ON \
        -D OPENCV_EXTRA_MODULES_PATH=/opencv_contrib-3.2.0/modules \
        -D BUILD_EXAMPLES=OFF \
        -D BUILD_opencv_python2=OFF \
        -D BUILD_NEW_PYTHON_SUPPORT=ON \
        -D CMAKE_INSTALL_PREFIX=$(python3 -c "import sys;
print(sys.prefix)") \
        -D PYTHON_EXECUTABLE=$(which python3) \
        -D WITH_FFMPEG=1 \
        -D WITH_CUDA=0 \
        .. \
    && make -j8 \
    && make install \
    && ldconfig \
    && rm /opencv.zip \
```

```
&& rm /opencv_contrib.zip

# Install dlib 19.4
RUN wget -O dlib-19.4.tar.bz2 http://dlib.net/files/dlib-19.4.tar.bz2 \
&& tar -vxjf dlib-19.4.tar.bz2

RUN cd dlib-19.4 \
&& cd examples \
&& mkdir build \
&& cd build \
&& cmake .. \
&& cmake --build . --config Release \
&& cd /dlib-19.4 \
&& pip3 install setuptools \
&& python3 setup.py install \
&& cd $WORKDIR \
&& rm /dlib-19.4.tar.bz2

ADD $PWD/requirements.txt /requirements.txt
RUN pip3 install -r /requirements.txt

CMD ["/bin/bash"]
```

Now execute the following command to build the image:

```
docker build -t hellorahulk/facerecognition -f Dockerfile
```

It will take approximately 20-30 mins to install all the dependencies and build the Docker image:

```
Successfully installed certifi-2018.10.15 chardet-3.0.4 idna-2.7 numpy-1.13.1 protobuf-3.6.1 requests-2.20.0 scikit-learn-0.19.1 scipy-0.19.1 six-1.11.0 tensorflow-1.13.0 urllib3-1.24 werkzeug-0.14.1
You are using pip version 8.1.1, however version 18.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Removing intermediate container 72eb5f7b562e
-> 82bb36e4238b
Step 10 : CMD ["/bin/bash"]
--> Running in 319b892dd1b7
Removing intermediate container 319b892dd1b7
-> 9ffce647496a
Successfully built 9ffce647496a
Successfully tagged hellorahulk/facerecognition:latest
```

Downloading pre-trained models

We will download a few more artifacts, which we will use and discuss in detail later in this chapter.

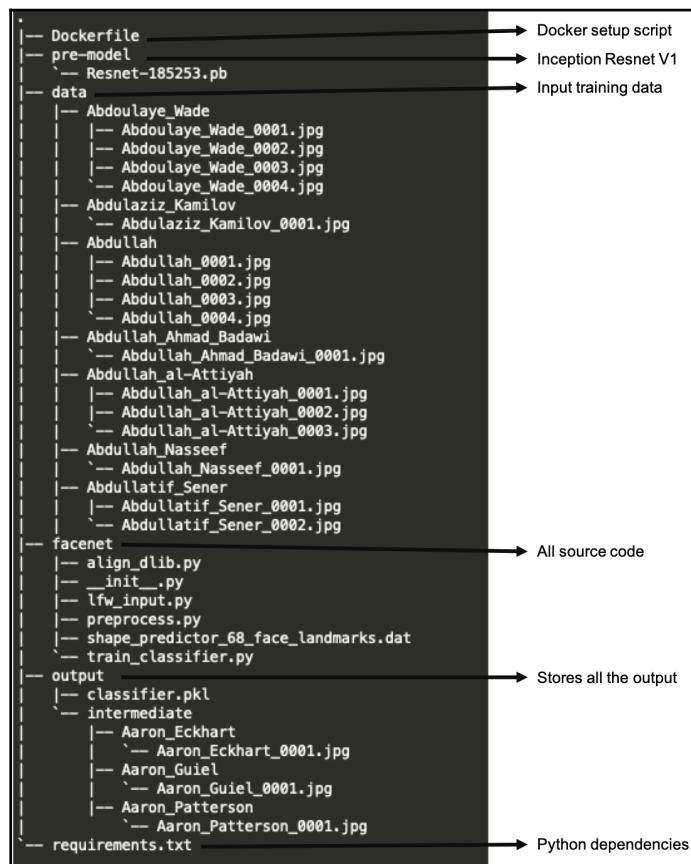
Download dlib's face landmark predictor, using the following commands:

```
curl -O http://dlib.net/
files/shape_predictor_68_face_landmarks.dat.bz2
bzip2 -d shape_predictor_68_face_landmarks.dat.bz2
cp shape_predictor_68_face_landmarks.dat facenet/
```

Download the pre-trained Inception model:

```
curl -L -O https://www.dropbox.com/s/hb75vuur8olyrtw/Resnet-185253.pb
cp Resnet-185253.pb pre-model/
```

Once we have all the components ready, the folder structure should look roughly as follows:



Make sure that you keep the images of the person you want to train the model with in the /data folder, and name the folder as /data/<class_name>/<class_name>_000<count>.jpg.

The /output folder will contain the trained SVM classifier and all preprocessed images inside a subfolder /intermediate, using the same folder nomenclature as in the /data folder.



Pro tip: For better performance in terms of accuracy, always keep more than five samples of images for each class. This will help the model to converge faster and generalize better.

Building the pipeline

Facial recognition is a biometric solution that measures the unique characteristics of faces. To perform facial recognition, you'll need a way to uniquely represent a face.

The main idea behind any face recognition system is to break the face down into unique features, and then use those features to represent identity.

Building a robust pipeline for feature extraction is very important, as it will directly affect the performance and accuracy of our system. In 1960, Woodrow Bledsoe used a technique involving marking the coordinates of prominent features of a face. Among these features were the location of hairline, eyes, and nose.



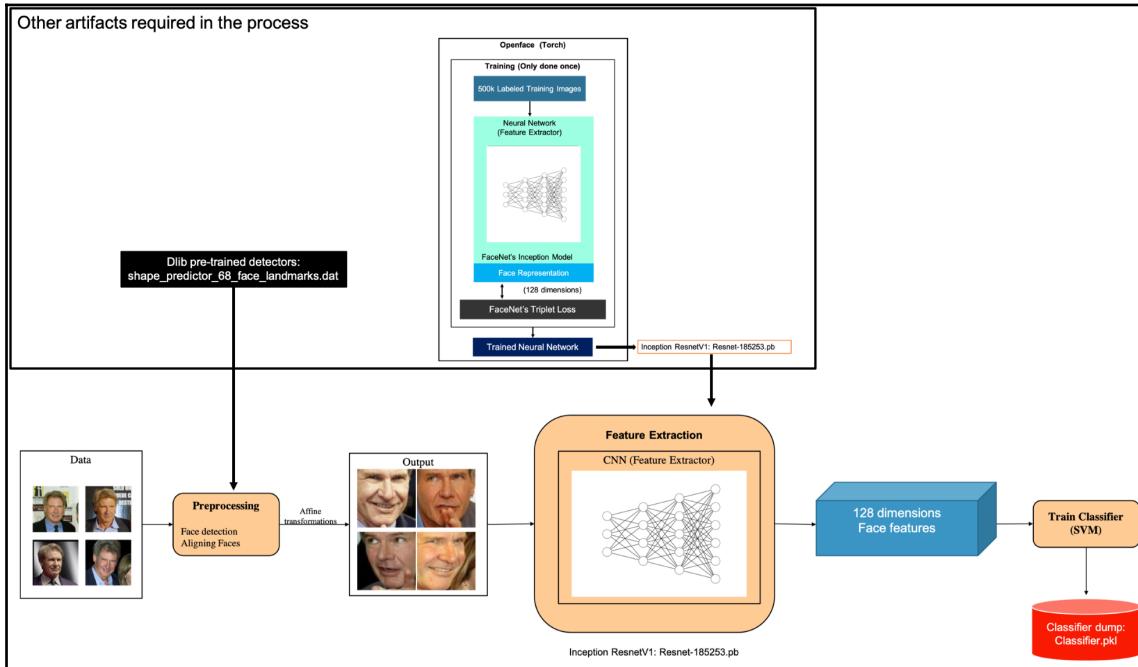
Later, in 2005, a much robust technique was invented, **Histogram of Oriented Gradients (HOG)**. This captured the orientation of the dense pixels in the provided image.

The most advanced technique yet, outperforming all others at the time of writing, uses CNNs. In 2015, researchers from Google released a paper describing their system, FaceNet (<https://arxiv.org/abs/1503.03832>), which uses a CNN relying on image pixels to identify features, rather than extracting them manually.

To build the face recognition pipeline, we will devise the following flow (represented by orange blocks in the diagram):

- **Preprocessing:** Finding all the faces, fixing the orientation of the faces
- **Feature extraction:** Extracting unique features from the processed faces
- **Classifier training:** Training the SVM classifier with 128 dimensional features

The diagram is as follows:



This image illustrates the end to end flow for face recognition pipeline

We will look into each of the steps, and build our world-class face recognition system.

Preprocessing of images

The first step in our pipeline is face detection. We will then align the faces, extract features, and then finalize our preprocessing on Docker.

Face detection

Obviously, it's very important to first locate the faces in the given photograph so that they can be fed into the later part of the pipeline. There are lots of ways to detect faces, such as detecting skin textures, oval/round shape detection, and other statistical methods. We're going to use a method called HOG.



HOG is a feature descriptor that represents the distribution (histograms) of directions of gradients (oriented gradients), which are used as features. Gradients (x and y derivatives) of an image are useful, because the magnitude of gradients is large around edges and corners (regions of abrupt intensity changes), which are excellent features in a given image.

To find faces in an image, we'll convert the image into greyscale. Then we'll look at every single pixel in our image, one at a time, and try to extract the orientation of the pixels using the HOG detector. We'll be using `dlib.get_frontal_face_detector()` to create our face detector.

The following small snippet demonstrates the HOG-based face detector being used in the implementation:

```
import sys
import dlib
from skimage import io

# Create a HOG face detector using the built-in dlib class
face_detector = dlib.get_frontal_face_detector()

# Load the image into an array
file_name = 'sample_face_image.jpeg'
image = io.imread(file_name)

# Run the HOG face detector on the image data.
# The result will be the bounding boxes of the faces in our image.
detected_faces = face_detector(image, 1)

print("Found {} faces.".format(len(detected_faces)))
```

```
# Loop through each face we found in the image
for i, face_rect in enumerate(detected_faces):
    # Detected faces are returned as an object with the coordinates
    # of the top, left, right and bottom edges
    print("- Face #{} found at Left: {} Top: {} Right: {} Bottom:
    {}".format(i+1, face_rect.left(), face_rect.top(), face_rect.right(),
    face_rect.bottom()))
```

The output is as follows:

```
Found 1 faces.
-Face #1 found at Left: 365 Top: 365 Right: 588 Bottom: 588
```

Aligning faces

Once we know the region in which the face is located, we can perform various kinds of isolation techniques to extract the face from the overall image.

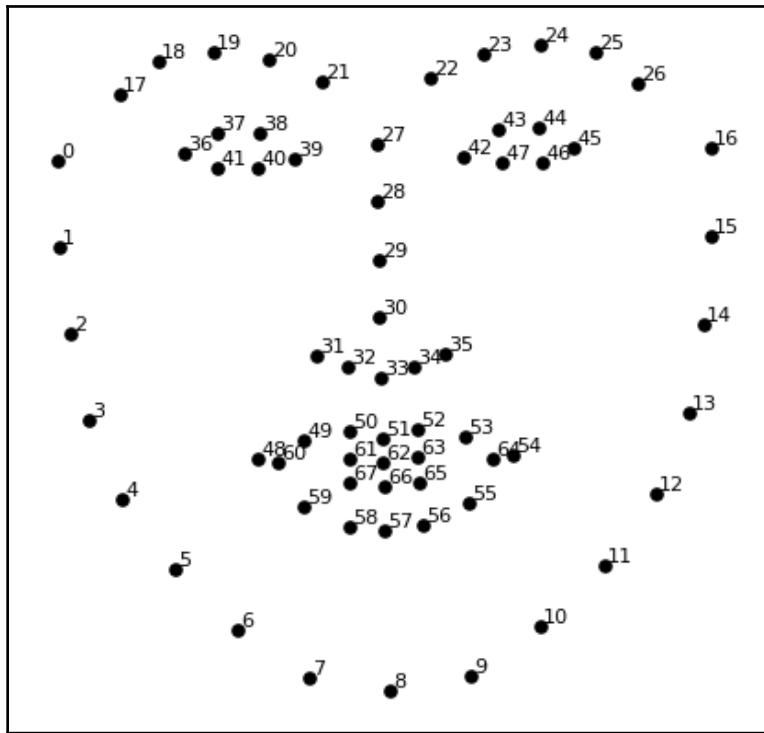
One challenge to deal with is that faces in images may be turned in different directions, making them look different to the machine.

To solve this issue, we will warp each image so that the eyes and lips are always in the sample place in the provided images. This will make it a lot easier for us to compare faces in the next steps. To do so, we are going to use an algorithm called **face landmark estimation**.



The basic idea is we will come up with 68 specific points (called *landmarks*) that exist on every face—the top of the chin, the outside edge of each eye, the inner edge of each eyebrow, and so on. Then we will train a machine learning algorithm to be able to find these 68 specific points on any face.

The 68 landmarks we will locate on every face are shown in the following diagram:



This image was created by Brandon Amos (<http://bamos.github.io/>), who works on OpenFace (<https://github.com/cmusatyalab/openface>).

Here is a small snippet demonstrating how to use face landmarks, which we downloaded in the *Setup environment* section:

```
import sys
import dlib
import cv2
import openface

predictor_model = "shape_predictor_68_face_landmarks.dat"

# Create a HOG face detector , Shape Predictor and Aligner
face_detector = dlib.get_frontal_face_detector()
face_pose_predictor = dlib.shape_predictor(predictor_model)
face_aligner = openface.AlignDlib(predictor_model)

# Take the image file name from the command line
```

```
file_name = 'sample_face_image.jpeg'

# Load the image
image = cv2.imread(file_name)

# Run the HOG face detector on the image data
detected_faces = face_detector(image, 1)

print("Found {} faces.".format(len(detected_faces)))

# Loop through each face we found in the image
for i, face_rect in enumerate(detected_faces):

    # Detected faces are returned as an object with the coordinates
    # of the top, left, right and bottom edges
    print("- Face #{} found at Left: {} Top: {} Right: {} Bottom: {}".
          format(i, face_rect.left(), face_rect.top(), face_rect.right(),
                 face_rect.bottom()))

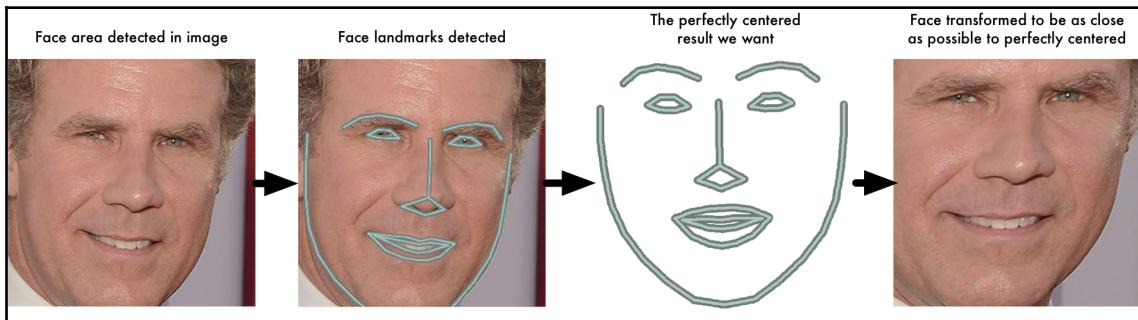
    # Get the the face's pose
    pose_landmarks = face_pose_predictor(image, face_rect)

    # Use openface to calculate and perform the face alignment
    alignedFace = face_aligner.align(534, image, face_rect,
                                    landmarkIndices=openface.AlignDlib.OUTER_EYES_AND_NOSE)

    # Save the aligned image to a file
    cv2.imwrite("aligned_face_{}.jpg".format(i), alignedFace)
```

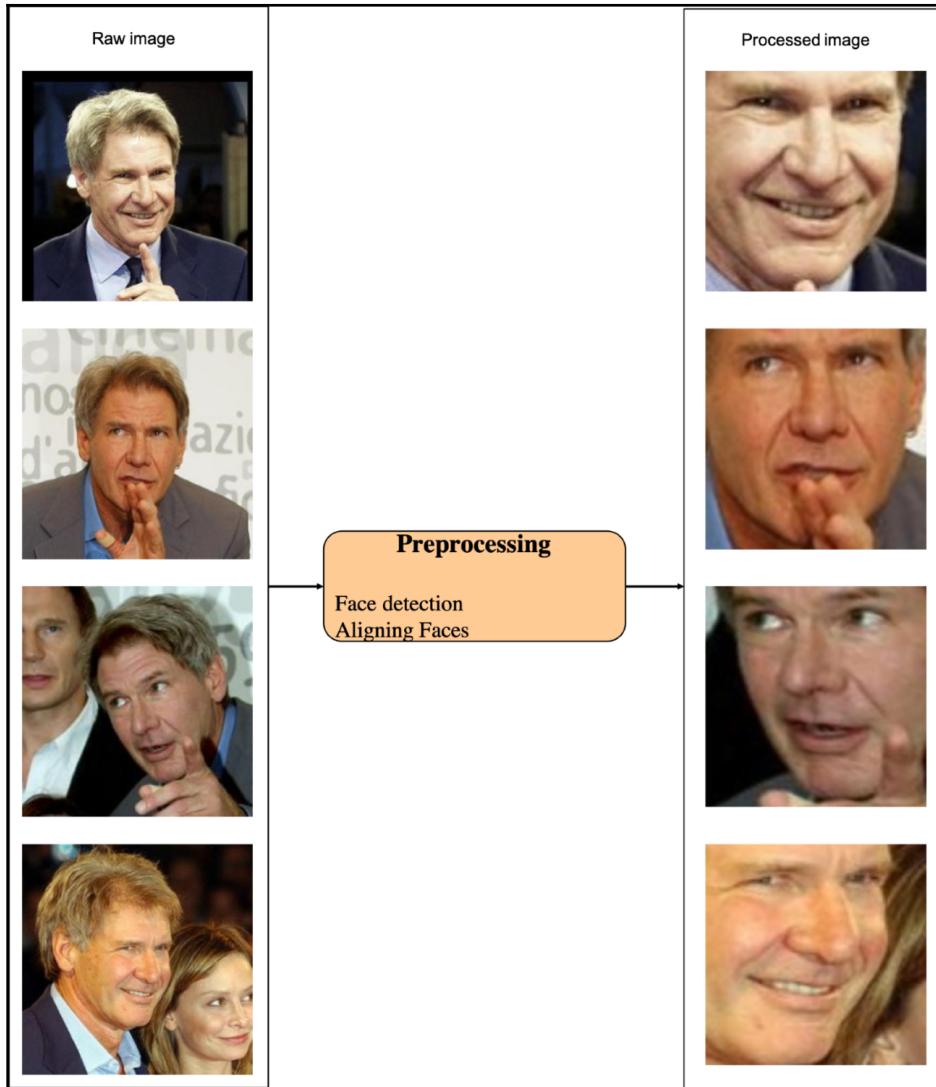
Using this, we can perform various basic image transformations such as rotation and scaling while preserving parallel lines. These are also known as affine transformations (https://en.wikipedia.org/wiki/Affine_transformation).

The output is as follows:



With segmentation, we solved finding the largest face in an image, and with alignment, we standardized the input image to be in the center based on the location of eyes and bottom lip.

Here is a sample from our dataset, showing the raw image and processed image:



Feature extraction

Now that we've segmented and aligned the data, we'll generate vector embeddings of each identity. These embeddings can then be used as input to a classification, regression, or clustering task.

This process of training a CNN to output face embeddings requires a lot of data and computer power. However, once the network has been trained, it can generate measurements for any face, even ones it has never seen before! So this step only needs to be done once.

For convenience, we have provided a model that has been pre-trained on Inception-Resnet-v1, which you can run over any face image to get the 128 dimension feature vectors. We downloaded this file in the *Setup environment* section, and it's located in the `/pre-model/Resnet-185253.pb` directory.



If you want to try this step yourself, OpenFace provides a Lua script (<https://github.com/cmusatyalab/openface/blob/master/batch-represent/batch-represent.lua>) that will generate embeddings for all images in a folder and write them to a CSV file.

The code to create the embeddings for the input images can be found further after the paragraph.

In the process, we are loading trained components from the Resnet model such as `embedding_layer`, `images_placeholder`, and `phase_train_placeholder`, along with the images and the labels:

```
def _create_embeddings(embedding_layer, images, labels, images_placeholder,
phase_train_placeholder, sess):
    """
    Uses model to generate embeddings from :param images.
    :param embedding_layer:
    :param images:
    :param labels:
    :param images_placeholder:
    :param phase_train_placeholder:
    :param sess:
    :return: (tuple): image embeddings and labels
    """
    emb_array = None
    label_array = None
    try:
```

```

i = 0
while True:
    batch_images, batch_labels = sess.run([images, labels])
    logger.info('Processing iteration {} batch of size:
    {}'.format(i, len(batch_labels)))
    emb = sess.run(embedding_layer,
                   feed_dict={images_placeholder: batch_images,
phase_train_placeholder: False})

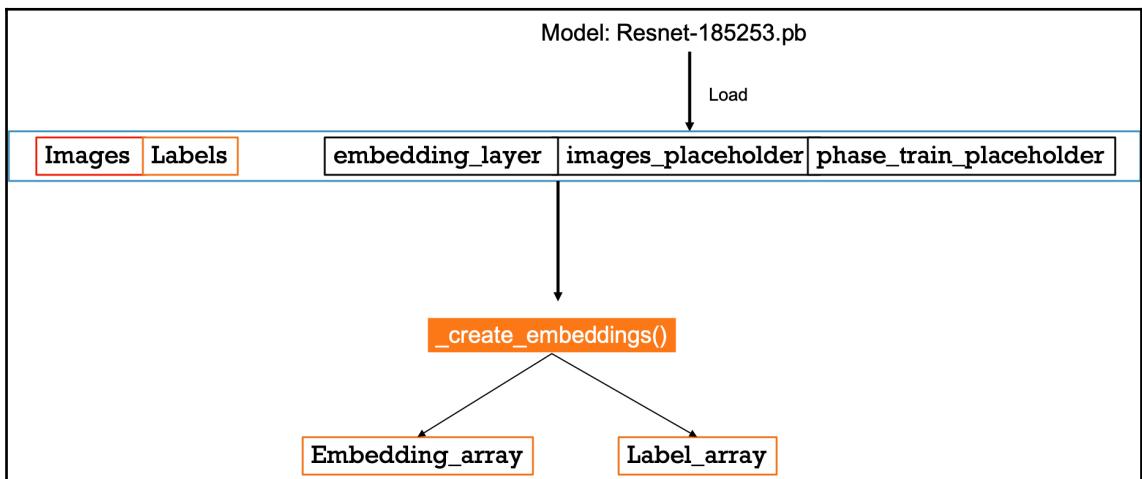
    emb_array = np.concatenate([emb_array, emb]) if emb_array is
not None else emb
    label_array = np.concatenate([label_array, batch_labels]) if
label_array is not None else batch_labels
    i += 1

except tf.errors.OutOfRangeError:
    pass

return emb_array, label_array

```

Here is a quick view of the embedding creating process. We fed the image and the label data along with few components from the pre-trained model:



The output of the process will be a vector of 128 dimensions, representing the facial image.

Execution on Docker

We will implement preprocessing on our Docker image. We'll mount the project directory as a volume inside the Docker container (using a `-v` flag), and run the preprocessing script on the input data. The results will be written to a directory specified with command-line arguments.

The `align_dlib.py` file is sourced from CMU. It provides methods for detecting a face in an image, finding facial landmarks, and aligning these landmarks:

```
docker run -v $PWD:/facerecognition \
-e PYTHONPATH=$PYTHONPATH:/facerecognition \
-it hellorahulk/facerecognition python3
/facerecognition/facenet/preprocess.py \
--input-dir /facerecognition/data \
--output-dir /facerecognition/output/intermediate \
--crop-dim 180
```

In the preceding command we are setting the input data path using a `--input-dir` flag. This directory should contain the images that we want to process.

We are also setting the output path using a `--output-dir` flag, which will store the segmented aligned images. We will be using these output images as input for training.

The `--crop-dim` flag is to define the output dimensions of the image. In this case, all images will be stored at 180×180 .

The outcome of this process will be an `/intermediate` folder being created inside the `/output` folder, containing all the preprocessed images.

Training the classifier

First, we'll load the segmented and aligned images from the `input` directory `--input-dir` flag. While training, we'll apply preprocessing to the image. This preprocessing will add random transformations to the image, creating more images to train on.

These images will be fed in a batch size of 128 into the pre-trained model. This model will return a 128-dimensional embedding for each image, returning a 128×128 matrix for each batch.

After these embeddings are created, we'll use them as feature inputs into a scikit-learn SVM classifier to train on each identity.

The following command will start the process, and train the classifier. The classifier will be dumped as a pickle file in the path defined in the `--classifier-path` argument:

```
docker run -v $PWD:/facerecognition \
-e PYTHONPATH=$PYTHONPATH:/facerecognition \
-it hellorahulk/facerecognition \
python3 /facerecognition/facenet/train_classifier.py \
--input-dir /facerecognition/output/intermediate \
--model-path /facerecognition/pre-model/Resnet-185253.pb \
--classifier-path /facerecognition/output/classifier.pkl \
--num-threads 16 \
--num-epochs 25 \
--min-num-images-per-class 10 \
--is-train
```

A few custom arguments are tunable:

- `--num-threads`: Modify according to the CPU/GPU config
- `--num-epochs`: Change according to your dataset
- `--min-num-images-per-class`: Change according to your dataset
- `--is-train`: Set the True flag for training

This process will take a while, depending on the number of images you are training on. Once the process is completed, you will find a `classifier.pkl` file inside the `/output` folder.

Now you can use the `classifier.pkl` file to make predictions, and deploy it on production.

Evaluation

We will evaluate the performance of the trained model. To do that, we will execute the following command:

```
docker run -v $PWD:/facerecognition \
-e PYTHONPATH=$PYTHONPATH:/facerecognition \
-it hellorahulk/facerecognition \
python3 /facerecognition/facenet/train_classifier.py \
--input-dir /facerecognition/output/intermediate \
--model-path /facerecognition/pre-model/Resnet-185253.pb \
```

```
--classifier-path /facerecognition/output/classifier.pkl \
--num-threads 16 \
--num-epochs 2 \
--min-num-images-per-class 10 \
```

Once the execution is completed, you will see predictions with a confidence score, as shown in the following screenshot:

```
INFO:__main__:Evaluating classifier on 22 images
  0 Abdullah_Gul: 0.997
  1 Abdullah_Gul: 1.000
  2 Abdullah_Gul: 0.999
  3 Abdullah_Gul: 1.000
  4 Abdullah_Gul: 0.998
  5 Adrien_Brody: 0.997
  6 Adrien_Brody: 0.963
  7 Adrien_Brody: 0.989
  8 Alejandro_Toledo: 1.000
  9 Alejandro_Toledo: 1.000
 10 Alejandro_Toledo: 0.993
 11 Alejandro_Toledo: 1.000
 12 Alejandro_Toledo: 1.000
 13 Alejandro_Toledo: 1.000
 14 Alejandro_Toledo: 0.505
 15 Alejandro_Toledo: 1.000
 16 Alejandro_Toledo: 0.995
 17 Alejandro_Toledo: 1.000
 18 Harrison_Ford: 0.945
 19 Alejandro_Toledo: 0.998
 20 Abdullah_Gul: 0.762
 21 Harrison_Ford: 0.987
Accuracy: 0.955
INFO:__main__:Completed in 7.507126092910767 seconds
```

We can see that the model is able to predict with 99.5% accuracy. It is also relatively fast.

Summary

We have successfully completed a world-class facial recognition POC for our hypothetical high-performance data center, utilizing the deep-learning technologies of OpenFace, dlib, and FaceNet.

We built a pipeline that included:

- **Face detection:** To examine an image and find all the faces it contains
- **Face extraction:** To focus on each face and understand its general qualities
- **Feature extraction:** To pull out unique features from the faces using CNNs
- **Classifier training:** To compare those unique features to all the people already known, and determine the person's name

The added security level of a robust facial recognition system for access control is in keeping with the high standards demanded by this Tier III facility. This project is a great example of the power of deep learning to produce solutions that make a meaningful impact on the business operations of our clients.

22

Generative Adversarial Networks

This chapter will help us understand the concepts of **generative adversarial networks (GANs)**. To understand the basic concepts, we will use a quirky example. Let's suppose I have some news for you, both good and bad. The good news is, there's a secret party in my neighborhood tonight. The bad news is that entry to this party is restricted. Luckily for you, I have a trick that will get you an entry pass.

How do you get an entry pass? The answer is simple: I will generate a fake one for you. However, the guards will probably recognize that this is a fake entry pass. Most likely, you'll then come back to me and berate me for the botched entry pass. I will then generate another one for you, only this time, it will be based on the guards' feedback. If the guards are still able to recognize that this is a fake, I will continue to generate tickets based on the feedback you get from the guards. I guarantee that after a period of time, I will be able to generate an exact replica of the original ticket.

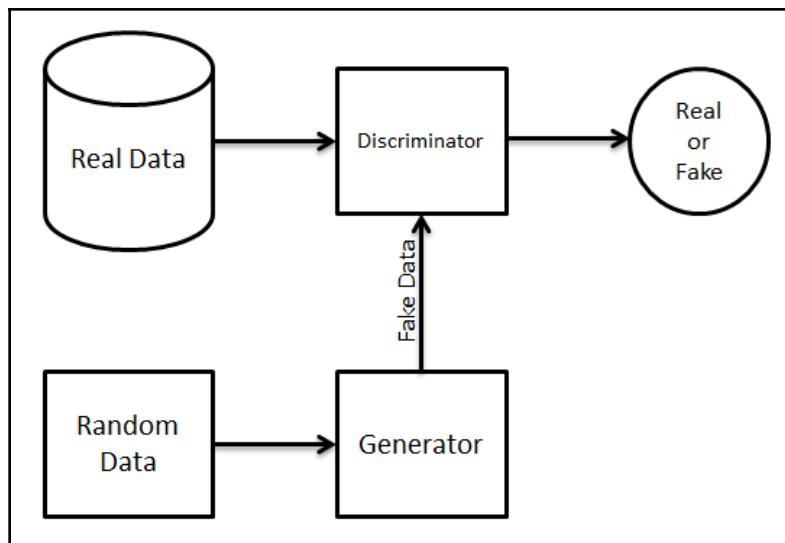
In this chapter, we will going to talk about the following topics:

- What are GANs?
- How do GANs work?
- Where can GANs be used?
- How can we implement a GAN?

We will start with a general introduction of GANs and then we will learn about the concepts of the technology by implementing 1D and 2D variations of GANs.

GANs

The previous example is a typical case of a GAN. It is a logical architecture with a fake data **generator** (me, in the example), fake data (our party tickets), genuine data (real entry passes), and a **discriminator** (the guards). Let's see what the architecture of a GAN looks like:



GANs are generative models that were devised by Goodfellow et al. in 2014. A GAN setup consists of two differentiable functions, represented by individual neural networks (in most cases), which are locked in a game. The two players (the generator and the discriminator) have different roles within this framework.

The generator tries to produce data that comes from a probability distribution. This is you trying to reproduce the party tickets (to continue with the allegory of the previous example).

The discriminator acts as a judge. It gets to decide whether the input comes from the generator or from the true training set. These are the party's security guards comparing your fake ticket with the true ticket to find flaws in your design.

In summary, the game is as follows:

- The generator tries to maximize the probability of making the discriminator mistake its input as genuine
- The discriminator guides the generator to produce more realistic images

In a perfect equilibrium, the generator would capture the distribution of the general training data. As a result, the discriminator would always be unsure of whether its inputs are real or not.

So far, GANs have primarily been applied to modeling natural images. They are now producing excellent results for image-generation tasks such as generating images that are significantly sharper than those trained using other leading generative methods based on maximum likelihood training objectives. Here are some examples of handwritten digit images generated by GANs:



GAN can also be used to convert intensity images (gray images) into color images, as in the following image:



These examples depict the use cases of GANs and the power they wield. The next section will describe the implementation of a simple GAN for 1D data generation. We will use TensorFlow to implement all our networks. Later in the chapter, we will see more intuitive applications of GANs. Let's begin by taking a look at how GANs are implemented practically.

Implementation of GANs

To further understand the underlying concepts of GANs, we will first implement a simple GAN architecture for 1D data approximation. Only then shall we proceed to enhance the capability of the network for 2D data, such as images.

In this example, we will try to fit a Gaussian distribution through our GAN architecture. This Gaussian distribution will have a mean and sigma value defined by the user. We will consider a distribution with a mean value of 0 and a sigma value of 1.

We will require the following elements:

- Real data generator
- Random data (input) generation for the generator network
- Linear layer

- Generator network for the generation of fake data
- Discriminator network to distinguish between fake and real data
- Optimizer to train our generator and discriminator

The linear layer will be used to create the generator and discriminator model.

The workflow for creating a GAN will be as follows:

1. We begin with the generation of real data samples x from distribution P_d for discriminator network D to learn what real data looks like. This will be a Gaussian distribution.
2. We will generate some random data z from a uniform distribution P_z . This will be the input to the discriminator D as well as to our generator network G . From this, our discriminator will learn to distinguish fake data from real data.
3. We will create two discriminator networks D_1 and D_2 . One will handle real data and the other will handle fake data that has been generated by our generator. Afterwards, we will train both networks with a combined loss. The important thing here is that both networks will share the same parameters.
4. D_1 and D_2 are copies of D (they share the same parameters, so $D_1(x)=D_2(x)$). The input to D_1 is a single sample of the legitimate data distribution, $x \sim P_d$, so when optimizing the discriminator, we want the quantity $D_1(x)$ to be maximized. D_2 takes as input x' (the fake data generated by G), so when optimizing D , we want $D_2(x')$ to be *minimized*. The value function for D is $\log(D_1(x)) + \log(1 - D_2(G(z)))$.
5. As you can see, we have made the copies D_1 and D_2 because in TensorFlow we need one copy of D to process the input x and another copy to process the input $G(z)$. The same section of the computational graph can't be reused for different inputs. When optimizing G , we want the quantity $D_2(X')$ to be maximized (successfully fooling D). The value function for G is $\log(D_2(G(z)))$.
6. Once we have defined our loss function, our next task will be optimizing this loss so that our networks can learn the underlying distributions. We will use a stochastic gradient descent algorithm for optimization but with an adaptive momentum algorithm.

We are now ready to implement our very first GAN model. We will create a Python file where we will use the TensorFlow library to create network graphs and train them. We will also use a few supportive libraries for algebraic operations as well as for output visualization.

The following libraries will be used in our code:

- NumPy: For matrix algebra and simple numeric calculations
- TensorFlow: For network building, training, and testing
- Matplotlib: For visualization of data by drawing plots

We will begin by creating a Python file with the name `GAN_1D.py` and importing the necessary packages that we discussed earlier:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

We have imported `os` for setting a CUDA device variable, which is only in case you have multiple GPUs on your board. In the next section, we will initialize a random number generator to make the results reproducible from NumPy as well as TensorFlow:

```
seed = 42
np.random.seed(seed)
tf.set_random_seed(seed)
```

We will now proceed to implement these features to generate data.

Real data generator

This module will help us generate real Gaussian distribution data with a mean of 0 and a sigma value (standard deviation) of 1. Our module will look as follows:

```
def RealInput(N, mu=0, sigma=1):
    p = np.random.normal(mu, sigma, N)
    return np.sort(p)
```

Whenever this class is called, it will generate the sample points from a Gaussian distribution of the defined mean and variance. Let's test it out:

```
# Create a main() function to call our defined method
def main():
    # We will generate a Gaussian distribution with 10000 points
    npoints = 10000
    # Let's call our real data generator
    x = RealInput(npoints)
    # Once we get the sample points we will calculate the histogram of the
    sample
```

```

#space;
#this is also known as the probability distribution function. It will
require
#a number of
#bins in which we want to distribute our sample points.
#We will evaluate our histogram for 100 bins in the range of -10 to 10
nbins = 100
nrangle = 10
#Linspace from numpy will help us to create linear spacing between the
points
bins = np.linspace(-nrangle, nrangle, nbins)
#We will use numpy's histogram function for creating a PDF for the
desired
#bins
pd, _ = np.histogram(x, bins=bins, density=True)
p_x = np.linspace(-nrangle, nrangle, len(pd))
#Following lines will help us out in the visualization of histogram
plt.plot(p_x, pd, label='real data')
plt.legend()
plt.show()

```

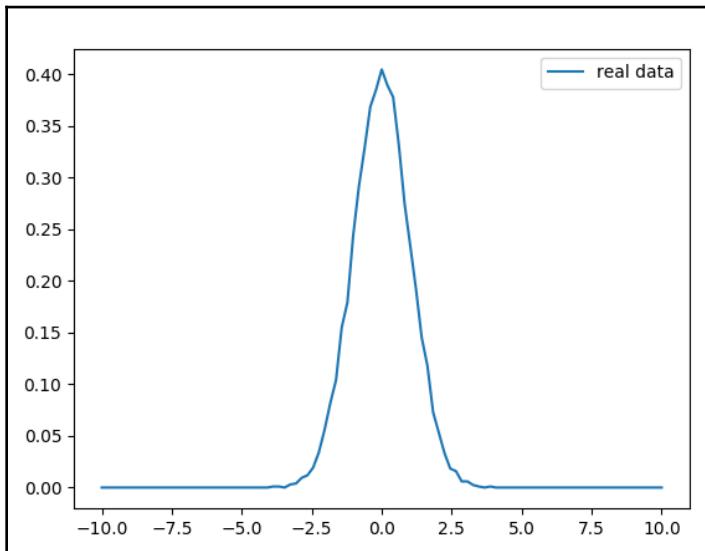
Once this is done, we will call our `main()` function and test our generated data distribution using the following line of code.

```

if __name__ == '__main__':
    main()

```

The output of our function looks as follows:



Once we have generated our input data, it's time to add a random data generator to our framework.

Random data generator

We will add the following function to our code for random data generation. This will generate samples from a uniform distribution. These samples will be generated using simple linear spacing in a user-defined range as follows:

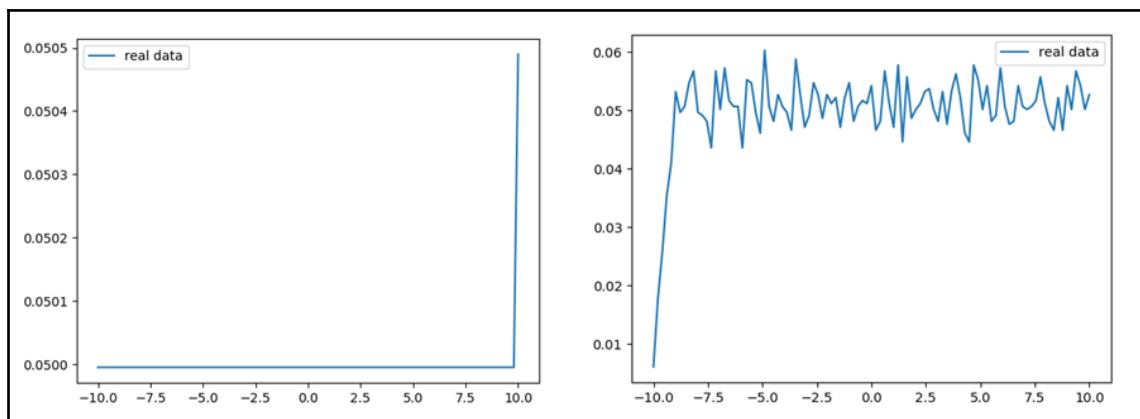
```
def RandomInput (nrange, N) :  
    return np.linspace (-nrange, nrange, N)+np.random (N)
```

In the preceding function, `nrange` is defined as the range in which we want our sample points and `N` is the number of samples to be drawn.

As you can see in the previous snippet, we have added some random noise to the data. This will help us achieve some variability in our samples. Let's observe how this works.

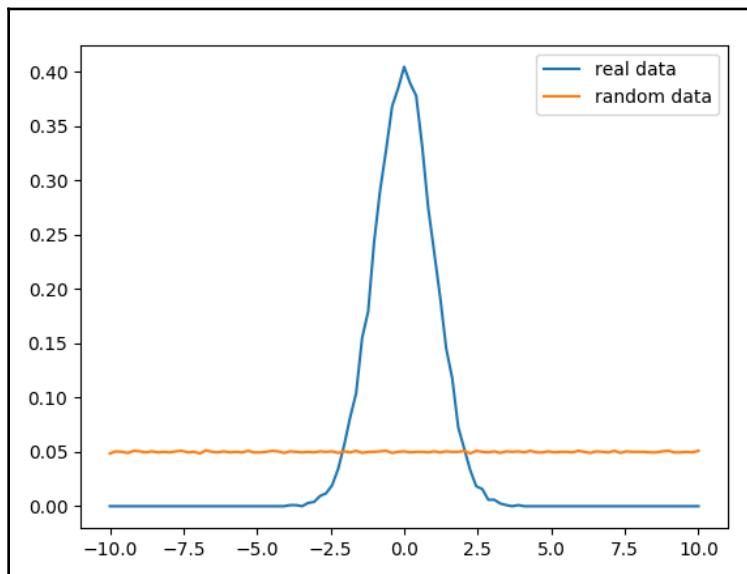
We will use the same `main()` file we used for real sample generation with a single modification. In this case, we will replace the real input function line with `RandomInput()`. We will draw a distribution with the same number of samples (10,000) and the same number of bins (100).

Let's visualize our sample's distribution, with and without the addition of noise. It should look something like this:



As can be observed in the preceding diagram, the addition of random signals helps us to get variable samples. This kind of sampling is also known as **Stratified Sampling**. It is generally used to increase the representativeness of the entire training space. In other words, it increases the variability.

If we put both of the signals on one plot, we can see the difference, as follows:



We now have our two data sample generator. The next building block in our architecture is to create a linear layer.

Linear layer

As the generator and discriminator are nothing but multilayer perceptron models, they have stacked layers made up of a linear combination of weights and biases. Before implementing a generator or discriminator network, we will write a function for a single layer perceptron and then combine these layers to create a generator or discriminator architecture:

```
#The following function is an implementation of a Perceptron layer
def Layer(inp,out_dim,scope,stddev=1.0):

    #This function will have the input sample INP;
    #The user needs to mention the dimension of the output;
    #STDDEV will be used to initialize the weight variable.
```

```

with tf.variable_scope(scope or 'Layer'):

#First we will create a tensorflow variable for weights.
#Its dimension will be calculated on the basis of INP and OUT_DIM
#We will initialize the weights using a gaussian distribution
w = tf.get_variable('w',
    [inp.get_shape()[1],out_dim],
    initializer=
        tf.random_normal_initializer(stddev=stddev))
#Then we will create a tf variable for bias.
b = tf.get_variable('b',[out_dim],
    initializer=tf.constant_initializer(0.0))
#Finally we will multiply thr weight matrix by the INP and add the
bias Term.
return tf.matmul(inp,w)+b

```

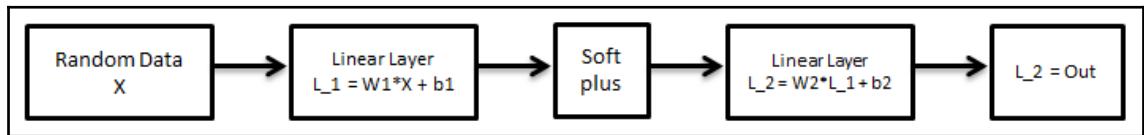
We will now create a generator network by combining multiple linear layers in cascade. We will simply place these layers, one-by-one, so that the output of the previous layer will be the input of the next layer. At the time of optimization (training), we will update the weight and bias parameters simultaneously for all the layers.

Generator

Our generator and discriminator networks are quite simple. The generator is a linear transformation passed through a non-linearity (a soft-plus function), followed by another linear transformation.

A generative model tries to learn the joint probability of the input data and labels simultaneously, namely $P(x,y)$. This can be converted to $P(y|x)$ for classification by using the Bayes rule (https://en.wikipedia.org/wiki/Bayes%27_theorem). The generative ability of the model can also be used for other purposes, such as creating likely new (x, y) samples.

The following diagram shows the internal architecture of the generative model:



We will add the following function with the name `Generator()` to create our generator model:

```
#This is our fake sample generator
def Generator(inp,out_dim):

    #Input to this network are samples generated by RandomInput function.
    #This function will create a two layer generator model.
    #We will add a Linear layer, the output of which will
    #be to send a soft-plus function to create non-linearity
    L_0 = tf.nn.softplus(tf.layers.dense(inp, out_dim, 'G_L_0'))
    #At the output end we will just add a linear layer without
    #any transformation function.
    out = tf.layers.dense(L_0, 1,'G_L_1')
    return out
```

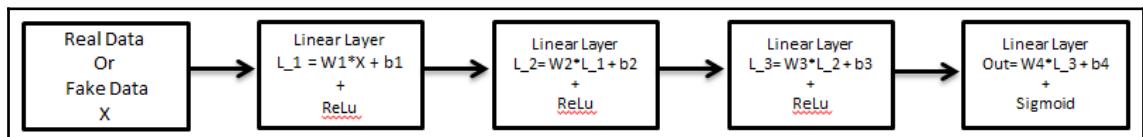
Now, it's time to add a discriminator network to our architecture.

Discriminator

Our discriminator model will have more linear layers than the generator model because it is a classifier model, so it is more complex than the generator model. We will add four (you can set the number of the layers as this depending on the complexity of your dataset) linear layers to our discriminator. The transformation function will be **rectifier linear units (ReLU)** for all layers except the output. We will use the sigmoid function in the last layer.

A discriminative model learns a function that maps the input data (x) to some desired output class label (y). In probabilistic terms, they directly learn the conditional distribution $P(y|x)$.

The following diagram shows the architecture that we are going to create for our discriminator:



The function to create the generator is as follows:

```
#Following function is implementation of discriminator
def Discriminator(inp,hlayer_dim):
    #INP is input to the function
    #HLAYER_DIM is number of weights required in the layer
    #This is our first linear layer with the ReLU transformation
    L_0 = tf.nn.relu(tf.layers.dense(inp, hlayer_dim, 'D_L_0'))
    #Second layer
    L_1 = tf.nn.relu(tf.layers.dense(L_0, hlayer_dim, 'D_L_1'))
    #Third layer
    L_2 = tf.nn.relu(tf.layers.dense(L_1, hlayer_dim, 'D_L_2'))
    #Final layer of the network will have a Sigmoid transformation
    #we have used Sigmoid to limit our output in the range of 0 and 1
    out = tf.sigmoid(tf.layers.dense(L_2, 1, 'D_L_3'))
    #Finally function will return the output of the final layer
    return out
```

We are done with creating the individual blocks of our GAN architecture. Let's now combine all of these together.

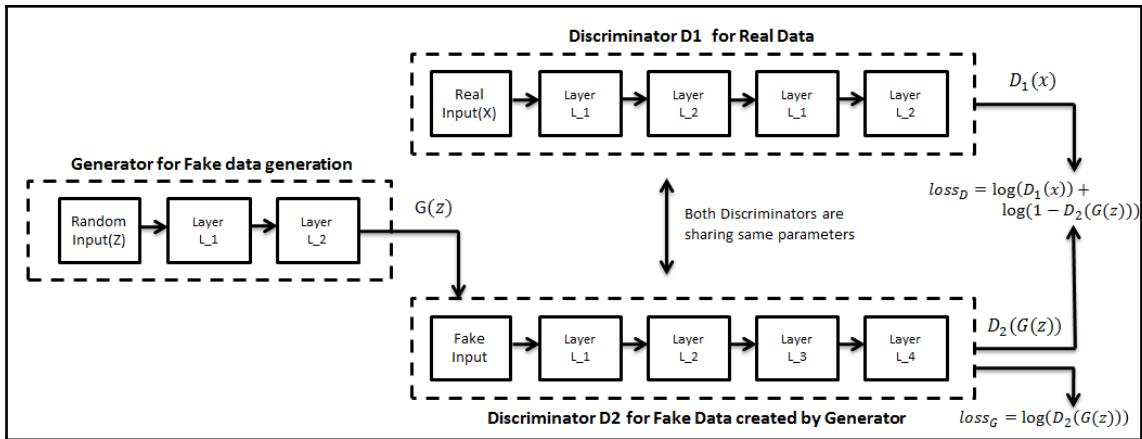
GAN

As we have already discussed, the main idea behind a GAN is to have two competing neural network models. The first model, the generator, considers noise as input and generates samples. The other model, the discriminator, receives samples from both the generator and the training data and has to be able to distinguish between the two sources.

These two networks play a continuous game, where the generator is learning to produce more and more realistic samples, and the discriminator is learning to distinguish the generated data from the real data more accurately. These two networks are trained simultaneously in the hope that the competition will drive the generated samples to be indistinguishable from the real data.

An important thing to note with the GAN is that we can back-propagate gradient information from the discriminator to the generator network. This means that the generator knows how to adapt its parameters in order to produce output data that can fool the discriminator.

The following diagram shows the architecture of the complete GAN model that we shall use to learn our distribution:



As we can see in the preceding diagram, we have created two copies of discriminators, and both of the discriminators share the same parameters. We have created these two copies because of the programming limitations of TensorFlow: we cannot simply reuse the same network with different inputs at the same time. During optimization, because both the discriminators have common parameters, we will optimize them together.

We then use the output of discriminator D2 in the loss calculation for the discriminator as well as for the generator. In this way, the generator learns how to generate more real-looking samples for the discriminator in the next iteration.

We will implement these concepts by using the following function:

```
#The following is the class written to create a GAN
class GAN():
    #We will write a class for GAN for simplicity
    #in variable management
    #You can also write a simple function for the task.
    #Currently we will put a default batch size of 8 samples.
    def __init__(self, inp_size=8):
        #We will create two place holders to store Real samples
        #and random samples with the input size defined by the user.
        self.z = tf.placeholder(tf.float32, shape=(inp_size, 1))
        self.x = tf.placeholder(tf.float32, shape=(inp_size, 1))
        #Whenever we are working with a generator,
        #we want all the parameters to
        #have same prefix 'G' so that we can identify them.
        with tf.variable_scope('G'):
            #Here we will create our Generator with the number of
            #hidden weights defined by user.
            self.G = Generator(self.z, 4)
        #Whenever we are working with a discriminator,
```

```

#we want all the parameters, have same prefix 'D'
#so that we can identify them.
with tf.variable_scope('D'):
    #Here we will create our Discriminator 1 with number of
    #hidden weights defined by user,
    #it will handle the real data samples.
    self.D_1 = Discriminator(self.x, 4)
#Here we will set flag reuse=True, it will help us to use same
#parameters from Discriminator 1
with tf.variable_scope('D', reuse=True):
    #Here we will create our Discriminator 2
    #with number of hidden weights
    #defined by user, it will handle the
    #output of generator(Fake data).
    self.D_2 = Discriminator(self.G, 4)
#Once we got outputs from both the discriminator we will use loss,
#function for checking the performance of our classifiers
#First we will check combined performance to update Discriminator.
self.loss_d = tf.reduce_mean(-log(self.D_1)-log(1-self.D_2))
#Then we will measure the loss for Generator optimization.
self.loss_g = tf.reduce_mean(-log(self.D_2))
#Here we will create a list of all
#trainable parameters from discriminator
#as well as generator
var = tf.trainable_variables()
#Here is the reason we have defined different variable scopes,
#for generator and discriminator, we will separate the
#parameters of both the networks to optimize them individually
self.d_params = [v for v in var if v.name.startswith('D/')]
self.g_params = [v for v in var if v.name.startswith('G/')]
#Here our optimization will take place
self.opt_d = optimizer(self.loss_d, self.d_params)
self.opt_g = optimizer(self.loss_g, self.g_params)

```

We have used **log loss** for loss calculation. We cannot use the log function from NumPy as we are working with tensors in our code, so we need to write a simple function to get the logarithm of the tensors. This can be written as follows:

```

#This function will return logarithmic of tensor
def log(x):
    #We will use tensorflow's log function for our task
    return tf.log(tf.maximum(x,1e-5))

```

If you look back at the previous code snippet showing the GAN class, we used the `optimizer` function in the last two lines of the function. This optimizer is actually our final building block; it helps to tune the network's weights so that both of our networks can attain the lowest possible loss.

We will use stochastic gradient descent to optimize the networks. We will also use the adaptive momentum method to train our networks so that we don't need to worry about the optimal hyperparameters for the SGD algorithm:

```
#Following is the function for optimizers
from tensorflow.train import AdamOptimizer
def optimizer(loss,var_list):
    #We will use initial learning rate as lr
    lr = 0.001
    #Update the weights after each iteration
    step = tf.Variable(0, trainable=False)
    #We will use adaptive momentum to tune the weights
    #It will change the learning rate according to speed
    #of convergence of the loss function.
    optimizer = AdamOptimizer(learning_rate=lr).minimize(loss,
                                                       global_step=step,
                                                       var_list=var_list)
    return optimizer
```

That's it—we have managed to combine all of the building blocks and now have a working GAN architecture. However, there are some questions that need to be answered. What about the working architecture? How do we test it? How do we train the model? How do we know if it is working or not?

Keep calm and train the GAN

We can now proceed to train our very first GAN model so that we can learn about Gaussian distribution. We will write the training module by using the following steps:

1. Define the sample size (batch size) for training the networks. The batch size is the number of training samples passed to the network in one iteration.
2. Create a TensorFlow session and initialize all local and global variables.
3. Generate the real and random samples in every iteration from the Gaussian and uniform distributions respectively.
4. Optimize the weights of the discriminators using the combined loss for the real and fake data points.
5. Optimize the weights for the generator using the loss made by discriminator 2, because discriminator 2 is dealing with the fake data generated by the generator.
6. Repeat this procedure for a defined number of times.

Let's now add a function that can do all of these things for us. Before doing so, however, we need to create the object of a GAN class so that we can pass it to the training module. This can be written as follows:

```
#Create a main() function to call our defined method
def main():
    #Create object of GAN class
    model = GAN()
    #Train out networks for 5000 iteration
    train(model, 5000)
```

The following is the function we will use to train our models:

```
def train(model,steps):
    #Let's choose a sample size of 8 samples per batch
    N = 8
    #We will start with creating a tensorflow session
    with tf.Session() as sess:
        #Initialize all local and global variables
        tf.local_variables_initializer().run()
        tf.global_variables_initializer().run()
        #Start training for user defined number of iterations
        for step in range(steps):
            #Get real samples
            x = RealInput(N)
            #Get random samples
            z = RandomInput(N,N)
            #Start training for both the discriminators
            loss_d,_, = sess.run([model.loss_d,model.opt_d],
                                {model.x: np.reshape(x, (N,1)),
                                 model.z: np.reshape(z, (N,1))})
            #Start training for Generator for random sample inputs
            z = RandomInput(N,N)
            loss_g,_, = sess.run([model.loss_g,model.opt_g],
                                {model.z: np.reshape(z, (N,1))})
            #Here we will print the loss values for each iteration
            print('{}: {:.4f}\t{:.4f}'.format(step, loss_d, loss_g))
```

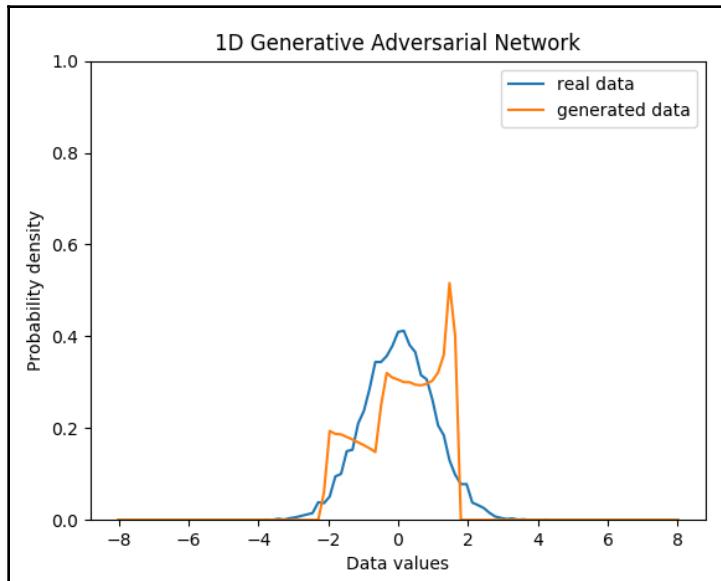
Once the training of our model is completed, we need to visualize the output of the trained generator. For this, we need to write a function to help us see the performance of our generator:

```
def visualize(npoints,session,model):
    #NPOINTS are the number of sample points
    #SESSION is tensorflow session
    #MODEL of Generator
    #Generate real samples for reference
    x = RealInput(npoints)
```

```
#Generate Random samples to input the generator in range of -8 to 8
z = RandomInput(8, npoints)
#We will process 8 points in one pass
batch_size = 8
#We will create a histogram with hundred bins in range of -8 to 8
nbins = 100
bins = np.linspace(-8, 8, nbins)
#First create the histogram for real data points
pd, _ = np.histogram(x, bins=bins, density=True)
#In following loop we will calculate generator's response
#for each batch of 8 points.
#Initialize a variable for store the response
g = np.zeros(npoints, 1)
#Following loop will run for number of batches
for i in range(npoints // batch_size):
    #Here we will calculate the response of G for each batch
    g[batch_size * i:batch_size * (i + 1)] = session.run(
        model.G,
        {
            model.z: np.reshape(
                z[batch_size * i:batch_size * (i + 1)],
                (batch_size, 1)
            )
        }
    )
#Once we got the response of G for all sample points
#create the histogram for generators output
pg, _ = np.histogram(g, bins=bins, density=True)
#Following lines will plot the Real distribution and fake
#distribution on the same plot.
p_x = np.linspace(-8, 8, len(pd))
f, ax = plt.subplots(1)
ax.set_ylim(0, 1)
plt.plot(p_x, pd, label='real data')
plt.plot(p_x, pg, label='generated data')
plt.title('1D Generative Adversarial Network')
plt.xlabel('Data values')
plt.ylabel('Probability density')
plt.legend()
plt.show()
```

Now, let's see how the generator deals with our random dataset after training it for 5,000 iterations. We will choose 1% of the random noise to perturb our linearly-spaced input dataset.

We will choose four neurons for both the discriminators as well as for the generator. After training, we get the following results that compare the data generated by the generator with the real data distribution:



Our input for the generator was almost a straight line and we got the preceding result, which clearly shows that our discriminators and generator are learning quite well. Although we have a very small generator network (of only two layers), the results are promising. We can conclude that we are on the right track.

We have learned to create a simple GAN architecture using a few layers to regenerate a 1D signal distribution using very few points (only eight). Now, we can work on enhancing the power of our GAN architecture to learn the distributions for 2D samples: images.

In the next section, we will build a GAN for reproducing image samples.

GAN for 2D data generation

Moving forward, let's bring back the example of fake ticket generation. We are still nowhere near building that application. You might wonder whether it really is possible to generate such data using GANs. I wouldn't blame you for having these doubts; the field of machine learning initially made quite strong promises, but failed to keep them when it comes to practical applications.

In this section, we will work toward generating real-looking fake images from our GAN architecture! This could lead us to generating those party tickets so that we can attend the party. Let's get started.

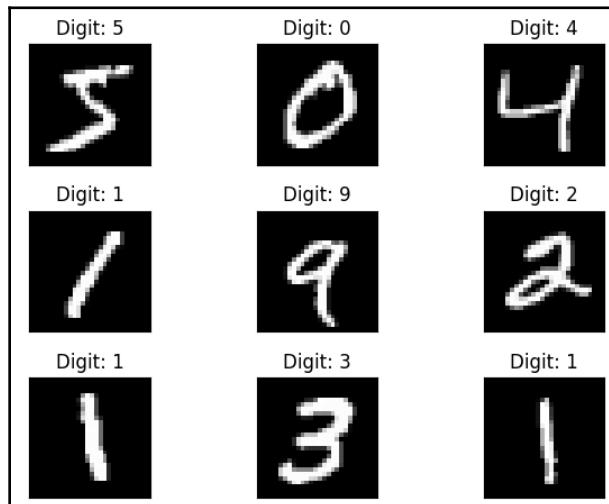
In the previous section, we successfully generated 1D data with the help of a basic generator and discriminator model. It is now time to move toward some more complex aspects of GANs. In this section, we will talk about the implementation of GANs for more complex data generation, such as images.

We will start with a very basic GAN architecture for generating handwritten digits. This dataset was originally created for experimental purposes, but later became a benchmark test for machine learning algorithms. The reason for this is quite obvious: this dataset has around 60,000 samples from the train and test set combined. It covers a huge range of variables and, most importantly, has a very large sample size. It has sample images with a size of 28x28 pixels, so anyone can train any kind of machine learning model on a very basic computer. For this reason, we'll start our discussion with digit recognition problem.

MNIST digit dataset

As this data has mainly been used to check the performance of the classification frameworks, it has been widely used for image recognition framework benchmarking. It has a decent sample size (28x28) for image recognition problems. You can find this dataset at <http://yann.lecun.com/exdb/mnist/>.

The following image depicts some of the images from the dataset:



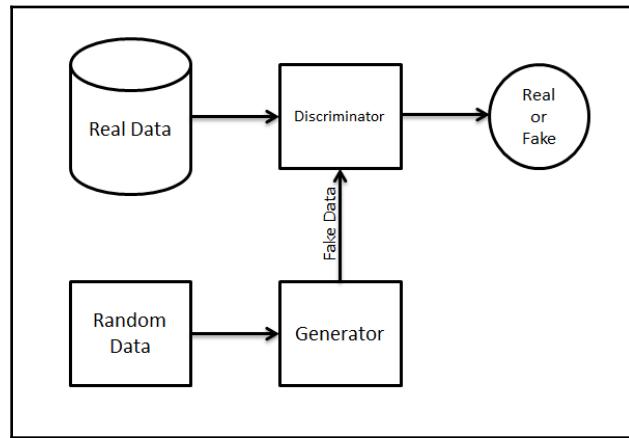
We will try to implement a GAN model to recreate these images from the input noise signal. Before implementing the network, let's see what challenges this creates for our previous architecture:

- This dataset contains 2D image samples
- As the sample size is quite big (784 pixels), it is difficult to handle with a basic network
- As this is image data, there will be translational variability present, which is very difficult to handle with plain neural networks

To resolve these issues, we need to make some serious changes to our GAN architecture. This time, we will use a deep convolutional GAN, which is also known as the **DCGAN architecture**. A CNN is the obvious choice to work with images. It's now time to implement our very first DCGAN.

DCGAN

As we have already discussed, CNNs are much more capable of handling image data than a plain neural network because of spatial context learning (such as weight sharing). We will try to incorporate these same fundamental elements to create a DCGAN for an image reconstruction-related task. The architecture of our GAN model will be similar to the one we used in 1D data generation. It will look as follows:



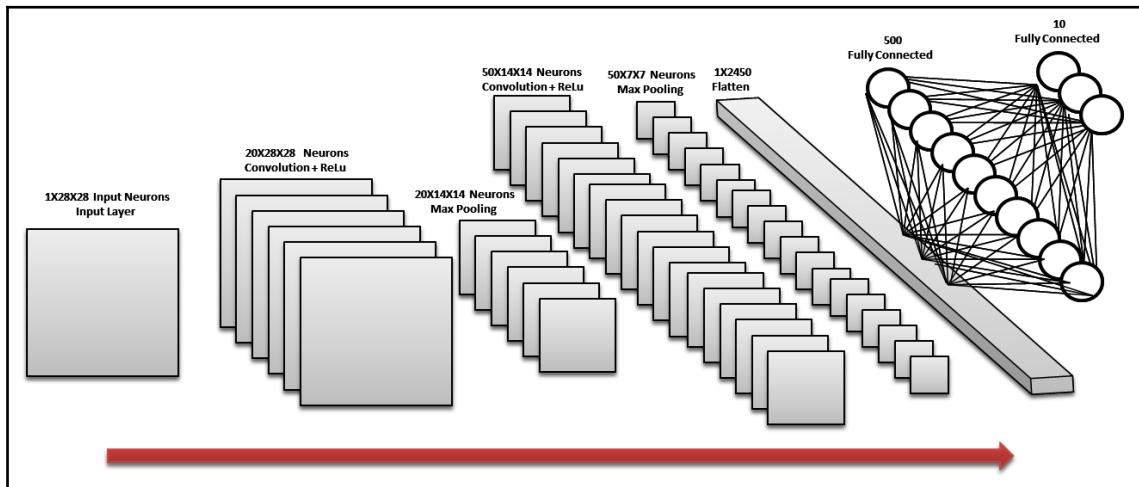
The major architectural changes will occur in the generator and discriminator networks. We will replace the plain linear layers (in both networks) with convolutional layers, where we will use shared weights and biases.

The procedure to train our DCGAN model will be the same as the one we discussed earlier. We will train our discriminator with real data and test it against both the real data and the fake data generated by the generator. We will then use the error (log loss) made by the discriminator as feedback in order to optimize the weights of the discriminator and the generator.

As we have already discussed, our discriminator is nothing but a binary classification network, which is designed to classify whether the input is real or fake. Therefore, the architecture of our discriminator will be similar to a typical CNN model.

Discriminator

Our model will have some sets of convolutional layers (with a different number of filters) with an activation function and one pooling layer. At the end of the network, there will be a sigmoid layer to get the class probabilities. The following diagram shows the architecture for a CNN classification network:

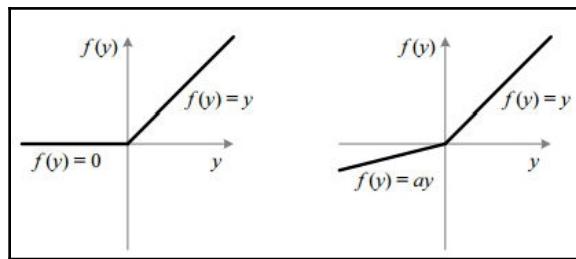


We started with an input with a single channel image. In each subset (convolutional layer, activation, and pooling), we increase the number of filters. In the final layers, there are the flattened and dense layers.

Let's start by implementing our discriminator model in Python. We will create a Python file called `GAN_MNIST.py` and put all the classes and functions for this problem in a single file.

We will use TensorFlow's core functions to create the convolutional and max pooling layers, but we will implement our own activation layer. This is because we do not want to use a plain **rectified linear unit (ReLU)**. Instead of ReLU, we will use another variant of it: Leaky ReLU. This is because ReLU always generates a non-negative ($>=0$) output, which makes it fragile during training and unable to cope with high learning rates.

The following diagram shows the difference between the two methods:



The left-hand side of the diagram shows the normal ReLU, where the value of function $f(y)$ is 0 for all values less than 0. The right-hand side of the diagram has some negative values. Here, we need to decide about the negative number limit. We will achieve this by multiplying the value by a coefficient. The equation for the leaky ReLU is as follows:

$$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

So, before implementing the discriminator function, we will implement a tiny function for our Leaky ReLU activation. The code snippet will be as follows:

```
#Following function is for leaky-ReLU
def LeakyRelu(x, alpha=0.2):
    #X: is input tensor
    #ALPHA: is leaky value
    #We need to just multiply the alpha value to input tensor
    value = tf.multiply(x, alpha)
    #Return the maximum of two values
    return tf.maximum(x, value)
```

We are now ready to implement the function for our discriminator network. As we have already discussed, it will be a CNN with a repeated set containing a convolutional layer and ReLU. We will not use a pooling layer here, but instead we will use strided convolutions. This will allow us to reduce the size of the input image.

The following is the code snippet for the discriminator network:

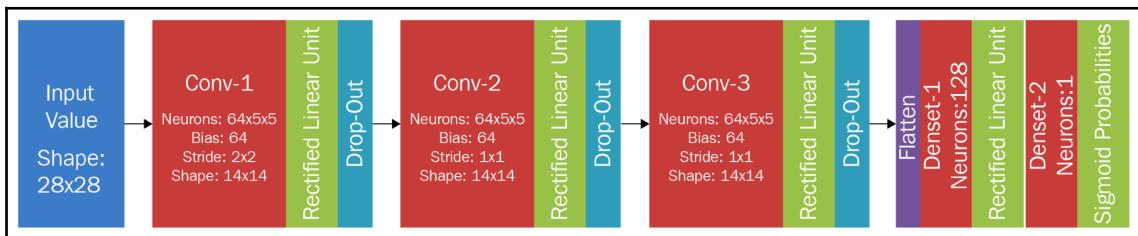
```
def Discriminator(img_in, reuse=None, prob=keep_prob, ksize = 5):
    #IMG_IN: is input image tensor.
    #REUSE: is for reusing same variables, as we need to create-
    #two copies of discriminator network it will necessary to define.
    #PROB: is drop-out probability.
    #KSIZE: is convolution kernel size we will choose 5X5 this time.
    #We will start with defining activation function for our model
    activation = LeakyRelu
    #We will set the variable scope for the discriminator network.
    with tf.variable_scope("discriminator", reuse=reuse):
        #We will start reshaping our input data into 1X28X28 from 28X28X1
        x = tf.reshape(img_in, shape=[-1, 28, 28, 1])
        #Here we will add first Convolution + ReLU with 64 filters,
        #kernel size will be 5X5 and a stride of 2,
        #it will reduce the size of image by half of its original size.
        #we will use 'same' padding which will retain the size after
        #convolution
        x = tf.layers.conv2d(x,kernel_size=ksize, filters=64, strides=2,
                            padding='same', activation=activation)
        #Now add some drop-out to the next layer
        #It will randomly remove some neurons from the training in each
        #epoch
        x = tf.layers.dropout(x, prob)
        #It will be our second Conv+ReLU again with the same configuration
        #as first
        #here we will not use strides so size of image will not reduce
        x = tf.layers.conv2d(x,kernel_size=ksize, filters=64, strides=1,
                            padding='same', activation=activation)
        #Again add some drop-outs
```

```

x = tf.layers.dropout(x, prob)
#This is our third Conv+ReLU with same configuration as previous
#layer
x = tf.layers.conv2d(x,kernel_size=ksize, filters=64, strides=1,
                     padding='same', activation=activation)
#Some more drop-outs
x = tf.layers.dropout(x, prob)
#Now we will convert 2D data into 1D tensor.
x = tf.contrib.layers.flatten(x)
#Here we will implement first dens layer with 128 neurons
x = tf.layers.dense(x,units=128, activation=activation)
#It will be our second dens layer with 1 neuron as it is a binary
#classifier, we will use sigmoid activation function here.
x = tf.layers.dense(x,units=1, activation=tf.nn.sigmoid)
#Return the model
return x

```

Our architecture will look as follows:



Let's talk about our architecture a little bit. The following operations take place:

1. We will start with an input layer, and our input size is 28x28.
2. In the convolutional layer, we will use 64 filters, each having a size of 5x5, and we will use a stride of 2. This means that during convolution, our filter will perform convolution operations on every alternate pixel. In other words, we will perform convolution operations on only half of the pixels. This reduces the size of the image by half and, as we are using same padding, there will not be any effect on the size of the image. Therefore, the output size of the image tensor will be 14x14x64. We will apply a Leaky ReLU in here, followed by a dropout layer. The probability of dropout is 20%, which means that 20% of the pixels will be randomly dropped during the training process in the current epoch. Dropouts prevent redundancy in the network by limiting (Dropping) the neurons to learn features for all the epochs.

3. In the second convolution layer, we will use the exact same configuration as the previous layer, except for the stride size. Here, we will use a stride of 1, so there will not be any change in the size of the image tensor. This means that the output tensor size will be $14 \times 14 \times 64$. Again, we will use the same activation and dropout probability as in the previous layer.
4. The third convolution layer is an exact replica of the previous layer, so there will not be any change in the hyperparameters, and the output tensor size will remain unaffected, staying at $14 \times 14 \times 64$.
5. We will now flatten the input of size $14 \times 14 \times 64$ to 12,544 by using the flatten layer. This will help us to create fully-connected dense layers. No trainable parameters will be used here.
6. The first dense layer is made up of 128 neurons. It is a fully-connected layer, so each neuron will be connected to each input element to the layer. We will again use Leaky ReLU. No dropout will be used here.
7. Finally, we will use a final dense layer or an output layer with only one neuron that has a sigmoid activation function. This will generate an output probability in-between 0 and 1. Ideally, it should be closer to 1 for true input samples, and closer to 0 for samples generated by the generator network.

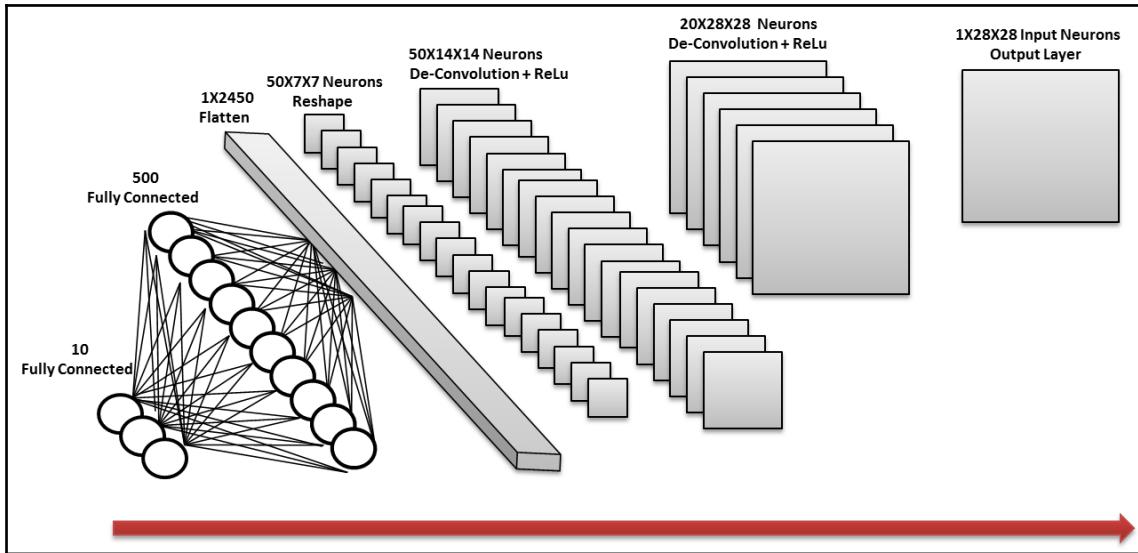
It's now time to talk about the generator network.

Generator

To design our generator network, we will use a convolutional network that is opposite in nature to the network used in the discriminator. We will start with some random input samples to a dense layer and then try to reconstruct this random sequence into a digital image using transpose convolutions.

These kinds of networks are also known as **decoder networks** and are widely used in CNNs for pixel-based predictions (**segmentation**). The difference between a decoder and a generator is that a decoder gets its input from an encoder, which is again a convolutional network such as our discriminator. Our generator, on the other hand, will not have an encoder; instead, it will try to reconstruct a random input sample into an image. We will adapt the architecture of our generator from a decoder network.

The following diagram shows the typical architecture of a decoder:

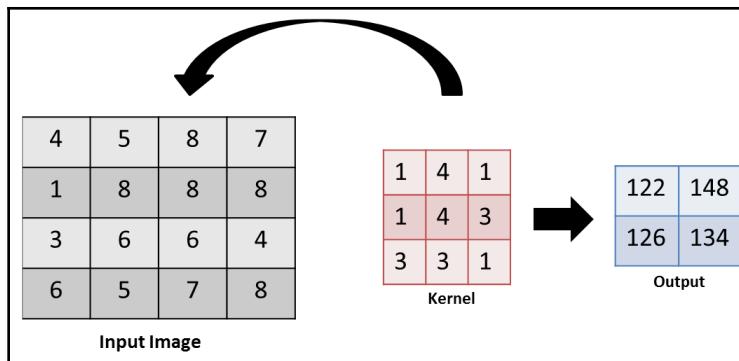


The preceding diagram shows a typical decoder model in which we start with dense layers that are fully connected to convolutional layers. These convolutional layers, however, are of a kind as they are transposed convolutions or deconvolution layers. We will talk about this in the following section.

Transposed convolutions

Transposed convolutions (TCs) are also known as **deconvolutions** in some literature. TCs have similar operations as convolution layers but differ in terms of their output. First, let's see how convolutions happen.

Let's use a simple example to understand this. Suppose we have a 4×4 matrix and apply a convolution operation on it with a 3×3 kernel, with no padding and with a stride of 1. As shown in the following diagram, the output is a 2×2 matrix:

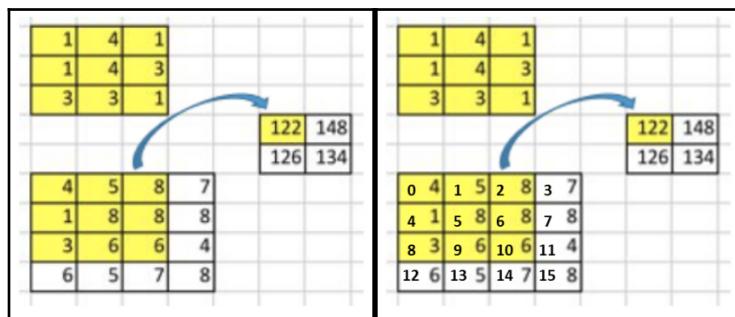


The convolution operation calculates the sum of the element-wise multiplication between the input matrix and kernel matrix. Since we have no padding and the stride is 1, we can only do this four times. Hence, the output matrix is 2x2.

Let's now change how we look at this operation. Suppose we do not want to do these multiple passes of convolution kernels over the input image. In that case, we can convert this convolution operation into a simple matrix multiplication operation. For this we need to observe the convolution operation closely. Let's look at the convolution example we discussed previously.

As we are not using any padding in here, our convolution will not start with the first image coordinate as our kernel cannot overlap the neighborhood completely. The following five operations will take place there:

- At first, our kernel will overlap with the yellow region marked in the following diagram, and the results of the sum of the product will be the first element of the output matrix. To understand the process, let's visualize the operation with linear indices of matrix elements rather than 2D Cartesian coordinates:



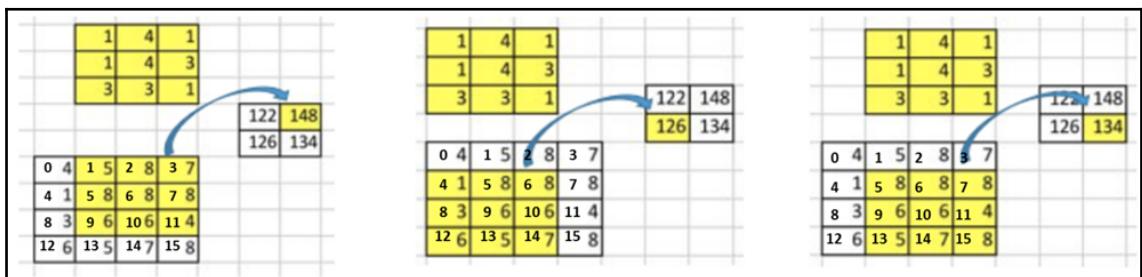
- Now, if we want to convert this first operation into matrix multiplication form, we just need to convert both matrices into 1D array forms. Let's convert the kernel matrix into a 1D array of 16 elements so that it will have nonzero values on the overlapping indices, and zero otherwise. In this case, we will have 0 values at the indices 3, 7, 11, 12, 13, 14, and 15. Our 1D array will look as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	1	0	1	4	3	0	3	3	1	0	0	0	0	0

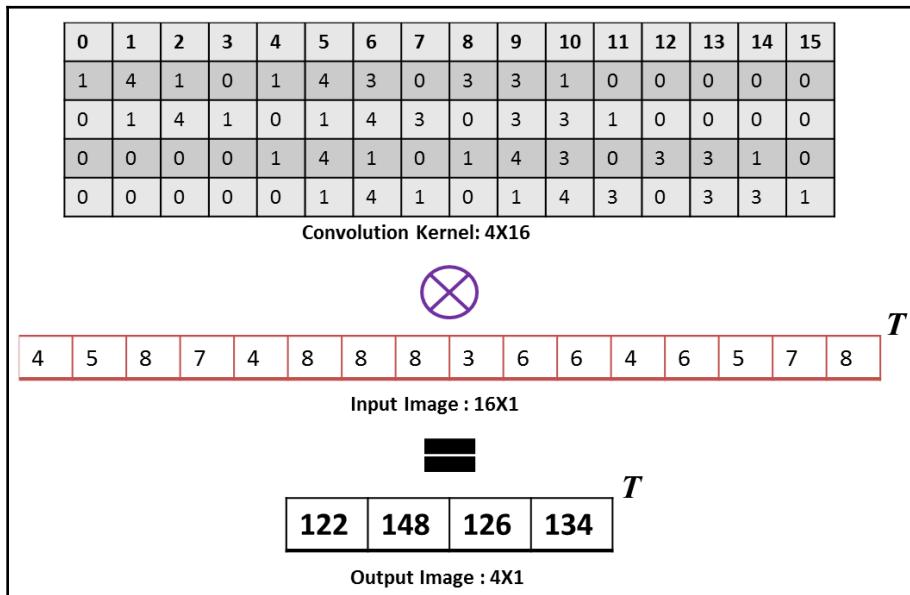
- You can see in the preceding diagram that we have kernel values on all the indices that overlap with kernel values and 0 on non-overlapping indices. Multiplying this array by our input image will give us the first element of our output matrix. The following diagram shows the multiplication operation for the first element:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	1	0	1	4	3	0	3	3	1	0	0	0	0	0
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4	5	8	7	1	8	8	8	3	6	6	4	6	5	7	8
4	20	8	0	1	32	24	0	9	18	6	0	0	0	0	0
122															

- Similarly, we can perform this multiplication for all other elements, which give us the other three output values:



- In matrix multiplication form, this can be written as follows:



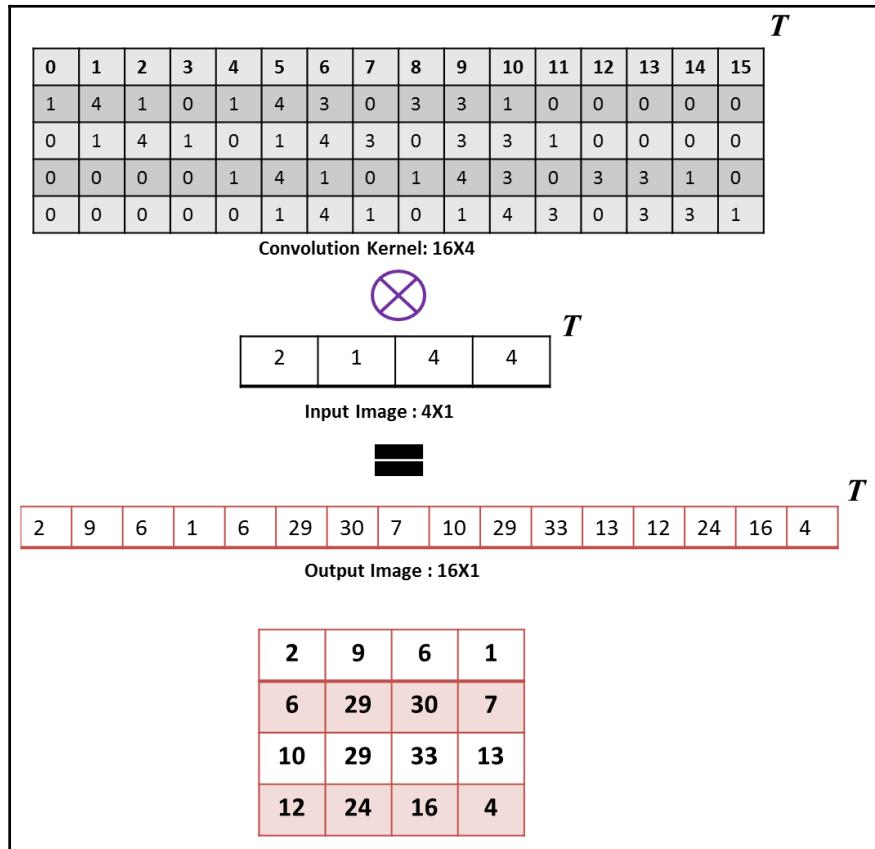
In the preceding diagram, the subscript T shows the transposition of row vectors to column vectors. As you can see, we have a kernel matrix with 4 rows and 16 columns, and each row of this matrix shows one individual multiplication with 16 inputs. By the matrix algebra, we have a kernel matrix of size 4×16 and an input image with a size of 16×1 . When we multiply both the matrices, we will get an output matrix of 4×1 , which will have similar values as in the case of the convolution operation.

But what is the relationship between the transposed convolutions and the matrix multiplication that we have discussed? Until you know how we have reduced the output image size using simple matrix multiplication, it is very difficult to understand the reverse process.

Now, suppose we want to go in the other direction. We want to associate one value in one matrix to nine values in another matrix. It's a **one-to-many relationship**. This is such as reversing the convolution operation, and this is the main idea behind transposed convolution.

Let's say we want to go from a 2×2 matrix to a 4×4 matrix. In this case, we would use a 16×4 matrix. In addition to this, we want to maintain the one to nine relationship.

Suppose we transpose the convolution matrix C (4x16) to C.T (16x4). We can matrix-multiply C.T (16x4) with a column vector (4x1) to generate an output matrix (16x1). The transposed matrix connects one value to nine values in the output. The following image depicts the transposed convolution as matrix multiplication:



The preceding diagram shows a completed transposed convolution. Even though it is called the **transposed convolution**, this does not mean that we take some existing convolution matrix and use the transposed version. The main point is that the association between the input and the output is handled in a backward fashion, compared to a standard convolution matrix (one-to-many rather than many-to-one association).

We will perform this transposed convolution operation using TensorFlow's transposed convolution layer. Before going into the code implementation for the generator, we need to discuss one more key aspect of the generator model: **batch normalization**.

Batch normalization

When we propagate our input through a layer of a neural network, we get different weight values in the same layer. This means that a single layer can generate a random range of outputs. If the activation is a ReLU function, the output values may range from 0 to ∞ , which creates complexity in convergence for the optimizer. To limit the output values from the activation function, we use batch normalization.

The functionality of the batch normalization layer is quite simple. It calculates the batch mean and the batch standard deviation of the input. Then, the batch mean is subtracted from the batch and divided by the standard deviation. This centralizes the tensor values around the batch mean with regular space.

During back-propagation, to remove the effect of normalization, this layer reverses its operation: it first multiplies the standard deviation (calculated during the forward pass) and then adds the batch mean to the input.

This layer has two trainable parameters: one is the batch mean and the other is the batch standard deviation. These two will be calculated for each layer wherever batch normalization has been used. For batch normalization in our code, we will use TensorFlow's batchnorm layer.

We have now learned both of the new concepts that will be included in our implementation. Now is the time to write the code for our generator. Let's get started:

```
def Generator(z, prob=keep_prob, is_training=is_training, ksize=5):
    #Z: is the random input vector.
    #PROB: is dropout probability
    #IS_TRAINING: flag is for batch norm layer
    #KSIZE: is convolution kernel size
    #We will use leaky ReLU activation
    activation = LeakyRelu
    #Add a regularization parameter for batch norm layer
    momentum = 0.99
    #Define the variable scope for generator
    with tf.variable_scope("generator", reuse=None):
        x = z
        #we will define neurons for the dense layer,
        #there are two value d1 and d2 it will be used,
        #for reshaping flatten output of dense layer into 2D matrix
        d1 = 4
        d2 = 1
        #Define our first dense layer
        x = tf.layers.dense(x, units=d1 * d1 * d2, activation=activation)
        #Add a dropout with defined probability
        x = tf.layers.dropout(x, prob)
```

```

#Here we add our batch normalization layer with regularization
x = tf.contrib.layers.batch_norm(x, is_training=is_training,
                                decay=momentum)
#Now we will reshape our flatten tensor into 2D tensor
x = tf.reshape(x, shape=[-1, d1, d1, d2])
#Resize the matrix with height and width = 7
x = tf.image.resize_images(x, size=[7, 7])
#Now we will perform transposed convolution here with 64 filters of
#5X5,Stride will be 2 with 'same' padding so upsampling will be
#done here tensor of 7X7 will be resized to 64X14X14
x = tf.layers.conv2d_transpose(x, kernel_size=ksize, filters=64,
                               strides=2, padding='same',
                               activation=activation)

#Add some dropout again
x = tf.layers.dropout(x, prob)
#Here will be our second batch norm layer
x = tf.contrib.layers.batch_norm(x, is_training=is_training,
                                decay=momentum)
#Add another convolution layer with exact same hyper parameter
#as previous
#So one more upsampling will happen here,
#output will have size of 64X28X28

x = tf.layers.conv2d_transpose(x, kernel_size=ksize, filters=64,
                               strides=2, padding='same',
                               activation=activation)

#Add another dropout layer
x = tf.layers.dropout(x, prob)
#Here will be our third batch norm layer
x = tf.contrib.layers.batch_norm(x, is_training=is_training,
                                decay=momentum)
#This will be our third transposed convolution layer,
#this time stride size=1 so no upsampling will happen here
#output size of the tensor will be 64X28X28
x = tf.layers.conv2d_transpose(x, kernel_size=ksize, filters=64,
                               strides=1, padding='same',
                               activation=activation)

#Our final dropout layer
x = tf.layers.dropout(x, prob)
#Final batch norm layer
x = tf.contrib.layers.batch_norm(x, is_training=is_training,
                                decay=momentum)
#Final convolution with single filter of 5X5
#stride size =1 so no upsampling,
#output tensor size will be 1X28X28
#Sigmoid activation will be used here
x = tf.layers.conv2d_transpose(x, kernel_size=ksize,
                               filters=1, strides=1,

```

```
padding='same',
activation=tf.nn.sigmoid)
#Return the Generator model
return x
```

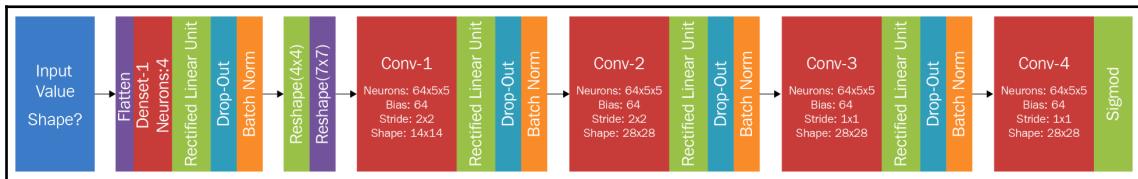
If you go through the complete code listing, it will be easy to understand the process because we have left a lot of comments in the code. We can summarize the whole process of the generator in the following steps.

We will use Leaky ReLU for all of the activations except in the last layer:

1. As the generator receives its input as a random 1D vector, we need to reshape it into the desired size using weight multiplication. For this, we will choose a number of neurons in the dense layer according to our application. Here, we will start with four neurons.
2. Next, there is a dropout layer with a dropout probability of 20% to reduce the redundancy from the network in a manner similar to what was done while training the discriminator.
3. Then, we will normalize the input tensor using the batch normalization layer. This will distribute the input points around the batch mean with the batch's standard deviation.
4. Next, we will convert our 1D vector into a 2D image matrix to apply the convolution operation. We will do this by simply reshaping the 1D vector into a 4x4 matrix.
5. We are now ready to apply the transposed convolution operation, but before that, we will resize our matrix of 4x4 and convert it into a 7x7 matrix. The number of filters will be 64 with a size of 5x5. The convolution stride will be two here, which will increase the size of input tensor by a factor of two, which gives us a 64x14x14 tensor at the output.
6. We will now add one more pair of dropout and batch norm layers.
7. As our tensor is still smaller (14x14) than the actual image (28x28), we need at least one more up sampling here. For this reason, we will use the same convolution configuration as the previous convolutional layer, which generates an output of 64x28x28 in size.
8. We will now add our third convolution layer with a filter size of 64x5x5. This time, however, the time stride will be one, so no up sampling takes place here. The output size of the tensor will be 64x28x28.
9. At this point, we are ready for our final convolutional layer. This converts our 64x28x28 tensor into 1x28x28. Before this, however, we will add one more pair of dropout and batch norm layers.

10. Lastly, we will add a final convolutional layer with one filter of 5x5. This will give us an output matrix of 1x28x28, which is exactly the same size as our actual input. We will use the sigmoid activation here to give output values in the range of 0 to 1.

The output of the final layer will be fed as the input of our discriminator. Our discriminator will decide whether this is a real image or a fake image. The following diagram shows the generator model implemented in our code:



GAN-2D

It's now time to mix all the parts in one class to create a GAN model. This class will be almost the same as the one we saw in our 1D GAN architecture, apart from a few slight differences. This class will contain all the parameters required to train the networks. The implementation is as follows:

```
from tensorflow.contrib.layers import apply_regularization
from tensorflow.contrib.layers import*
class GAN():
    def __init__(self):
        #Lets start with defining the batch size.
        #it will same for noise and real data
        self.batch_size = 196
        self.n_noise = 196
        #We need to create 2 place holders to hold the data values for
        #noise and data
        self.X_in = tf.placeholder(dtype=tf.float32,
                                  shape=[None, 28, 28],
                                  name='X')
        self.noise = tf.placeholder(dtype=tf.float32,
                                   shape=[None, self.n_noise])
        #Here we call our generator to generate false data
        #we need to define dropout probability and training mode.
        self.g = Generator(self.noise, keep_prob, is_training)
        #Here we will create two discriminator models.
        #these two will share same parameters(weights and biases).
        #one will operate on real data while other is on fake one.
```

```

self.d_real = Discriminator(self.X_in)
self.d_fake = Discriminator(self.g, reuse=True)
#Separate the trainable variables for both generator as well as
#discriminator
self.vars_g = [var for var in tf.trainable_variables()
               if var.name.startswith("generator")]
self.vars_d = [var for var in tf.trainable_variables()
               if var.name.startswith("discriminator")]
#Here we will apply some regularization on weights of
#generator and discriminator
self.d_reg = apply_regularization(l2_regularizer(1e-6), self.vars_d)
self.g_reg = apply_regularization(l2_regularizer(1e-6), self.vars_g)
#We will use binary cross entropy loss to measure the performance
#of our discriminators, output label will be zero for fake data
#and one for real data.
self.loss_d_real = binary_crossentropy(tf.ones_like(self.d_real),
                                       self.d_real)
self.loss_d_fake = binary_crossentropy(tf.zeros_like(self.d_fake),
                                       self.d_fake)
#Here we will calculate the loss for both networks
self.loss_g =
    tf.reduce_mean(binary_crossentropy(tf.ones_like(self.d_fake),
                                      self.d_fake))
self.loss_d = tf.reduce_mean(0.5 * (self.loss_d_real +
                                    self.loss_d_fake))
#Let's update the graphs
self.update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
#Now its time to train the networks
with tf.control_dependencies(self.update_ops):
    #Total loss of Discriminator with regularization
    total_loss_d = self.loss_d + self.d_reg
    #Total loss of Generator with regularization
    total_loss_g = self.loss_g + self.g_reg
    #Set the learning rate
    lr = 0.00015
    #We will use RMSprop with SGD for training the
    #networks
    self.optimizer_d = tf.train.RMSPropOptimizer(learning_rate=
        lr).minimize(total_loss_d, var_list=self.vars_d)
    self.optimizer_g = tf.train.RMSPropOptimizer(learning_rate=
        lr).minimize(total_loss_g, var_list=self.vars_g)

```

If you look carefully at the code, you will find an unknown function we have not yet discussed. If you missed it, go back and look closely. The unknown function is a binary cross entropy function.

Here is the code listing for binary cross entropy:

```
def binary_cross_entropy(x, z):
    #X: Expected output of network
    #Z: Actual output of the network
    #Define a small value epsilon prevent log(0) error
    eps = 1e-12
    #Calculate the loss
    return (-(x * tf.log(z + eps) + (1. - x) * tf.log(1. - z + eps)))
```

So, after creating a GAN architecture, our next step will be to *train the GAN*.

Training the GAN model for 2D data generation

It is now time to train our GAN model and see whether or not we can learn the handwritten digits. We will use the following steps to train our model:

1. Create a TensorFlow session and initialize all the variables created.
2. Train a batch of 196 images per iteration.
3. As we are currently in training mode, we will set the training flags for both of the models as we need to train the parameters for the batch normalization layer.
4. We will use NumPy's random number generator to generate a uniform distribution vector, which will be the input of our generator. Similarly, we will fetch the real images from the **Modified National Institute of Standards and Technology (MNIST)** at https://en.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology database dataset that comes with the TensorFlow package.
5. Again, we will create two discriminators that share the same parameters. One will get real samples and the other will get the output of the generator.
6. We will use binary cross entropy loss for performance evaluation and then pass this loss to the RMSprop optimizer for parameter optimization.
7. During training, we will take care that no part of the network becomes more powerful than another. This balance will be maintained using conditions. We will continuously monitor the loss values of the generator and the discriminator, and if the generator's loss is less than the discriminator's, we will stop the training of the generator until the loss is balanced. We will do the same thing for discriminator training.
8. We will monitor the loss values after 50 iterations and then test the generator on a batch of images.
9. We then will use an image montage to visualize the reconstruction generated by the generator.

The following is the code snippet for the whole training process:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data')
def train(model):
    #We will start with creating a tensorflow session
    sess = tf.Session()
    #Initialize all the variables
    sess.run(tf.global_variables_initializer())
    #And hit the training loop
    for i in range(60000):
        #Set training flag on for batch normalization
        train_d = True
        train_g = True
        #Set Drop out probability
        keep_prob_train = 0.6 # 0.5
        #Here we will create random data to input generator
        n = np.random.uniform(0.0, 1.0,
                              [model.batch_size,
                               model.n_noise]).astype(np.float32)
        #And here we will fetch input data from tensorflow
        batch = [np.reshape(b, [28, 28])
                 for b in mnist.train.next_batch(batch_size=
                                                 model.batch_size)[0]]
        #Lets start training by executing the session
        d_real_ls, d_fake_ls, g_ls, d_ls = sess.run([model.loss_d_real,
                                                       model.loss_d_fake,
                                                       model.loss_g,
                                                       model.loss_d],
                                                    feed_dict={model.X_in:
                                                               batch,
                                                               model.noise:
                                                               n,
                                                               keep_prob:
                                                               keep_prob_train,
                                                               is_training:True})

        #Each epoch will give us loss by discriminator for real
        #and fake data take mean of it
        d_real_ls = np.mean(d_real_ls)
        d_fake_ls = np.mean(d_fake_ls)
        g_ls = g_ls
        d_ls = d_ls
        #Here we will set some conditions so our discriminator
        #can not be strong than generator or vice versa
        #We will use the flags to stop training the discriminator or
        #generator
        if g_ls * 1.5 < d_ls:
```

```

        train_g = False
        pass
    if d_ls * 2 < g_ls:
        train_d = False
        pass
    #Here we will limit the training
    if train_d:
        sess.run(model.optimizer_d,
                 feed_dict={model.noise: n,
                            model.X_in: batch,
                            keep_prob: keep_prob_train,
                            is_training:True})

    if train_g:
        sess.run(model.optimizer_g,
                 feed_dict={model.noise: n,
                            keep_prob: keep_prob_train,
                            is_training:True})
    #Lets print out the results here
    #we will print all the losses after every 50 epochs
    if not i % 50:
        print (i, d_ls, g_ls, d_real_ls, d_fake_ls)
    if not train_g:
        print("not training generator")
    if not train_d:
        print("not training discriminator")
    #Let's test the trained generator after every 50 iterations
    gen_img = sess.run(model.g,
                       feed_dict = {model.noise: n,
                                    keep_prob: 1.0,
                                    is_training:False})

    #Here we will plot all the images of a single batch in montage
    #form
    imgs = [img[:, :, 0] for img in gen_img]
    m = montage(imgs)
    gen_img = m
    plt.axis('off')
    plt.imshow(gen_img, cmap='gray')
    plt.draw()
    plt.pause(0.00001)

```

Now, let's write the code to create the image montage to visualize the output of our network:

```

def montage(images):
    #Lets start by checking whether we have got an array or list
    if isinstance(images, list):
        #If list is there convert it into an array

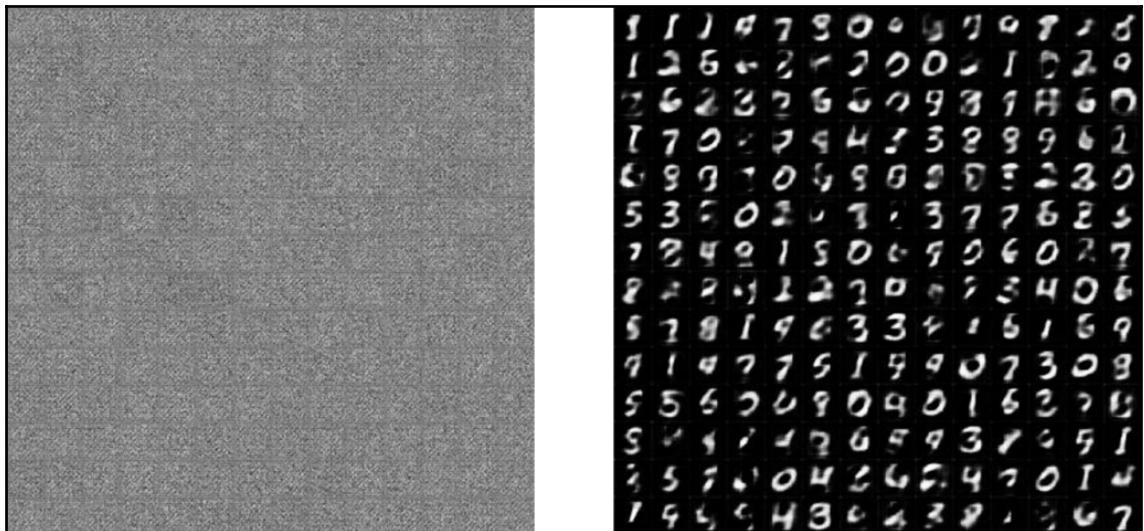
```

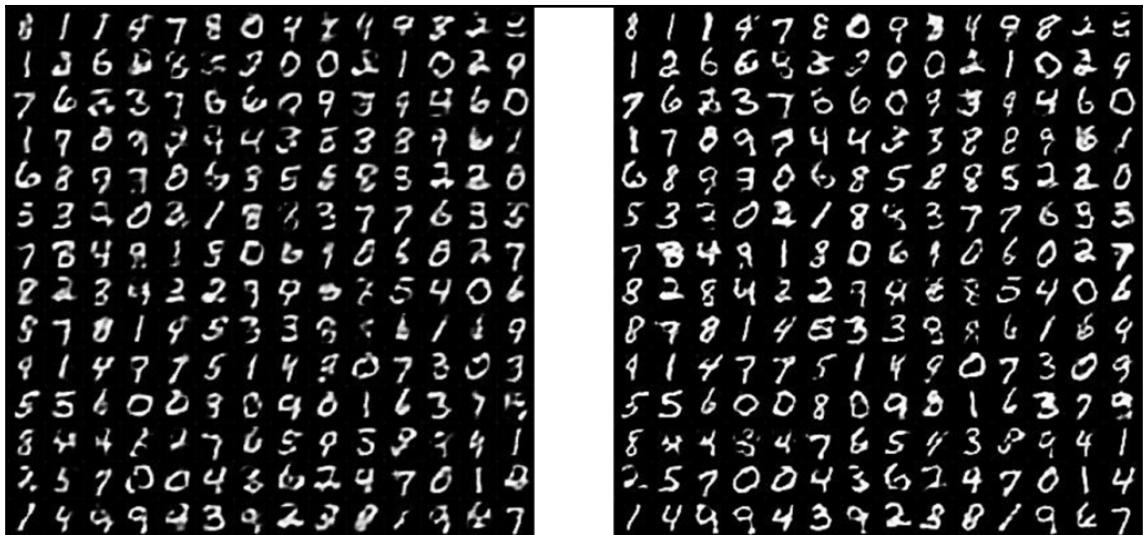
```
images = np.array(images)

#Get shape of the input array
img_h = images.shape[1]
img_w = images.shape[2]
#Calculate grid size
n_plots = int(np.ceil(np.sqrt(images.shape[0])))
#Create an array of ones from calculated grid size
m = np.ones((images.shape[1] * n_plots + n_plots + 1, images.shape[2] *
             n_plots + n_plots + 1)) * 0.5

#Start plotting images on the grid one by one
for i in range(n_plots):
    for j in range(n_plots):
        this_filter = i * n_plots + j
        if this_filter < images.shape[0]:
            this_img = images[this_filter]
            m[1 + i + i * img_h:1 + i + (i + 1) * img_h,
              1 + j + j * img_w:1 + j + (j + 1) * img_w] = this_img
return m
```

Let's visualize the results generated by our generator by using different epochs:





Outputs generated by the generator at different numbers of iterations (iterations are increasing from left to right and top to bottom)

Voila! We have successfully created a GAN for an image dataset. Even though our network is too small to be a deep neural network, it still has a lot of capabilities, and we are still getting very good results from it.

So what do you think—can we generate fake tickets using a GAN? I think we can! A GAN to generate fake tickets will need a more complex architecture, including more convolution layers with more filters and a lot of training samples, but it is still possible! There are many more applications of GANs, but in terms of the implementation of this particular problem, we will stop here for now.

In the next section, we will talk briefly about the different types of GANs that are used in the industry for different tasks.

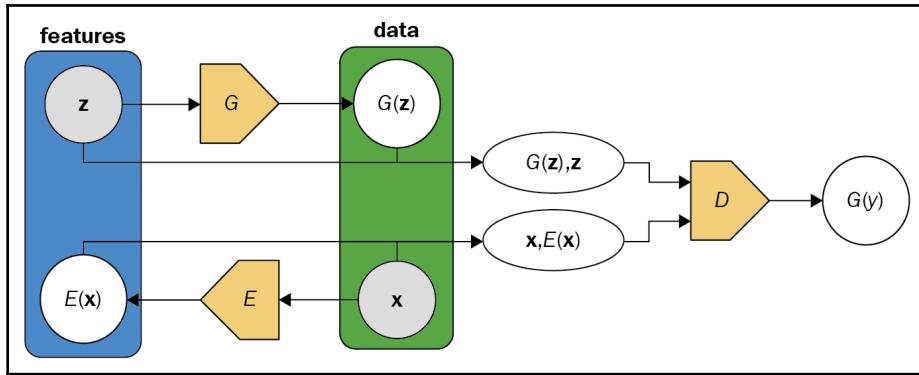
GAN Zoo

Currently, there is a huge amount of literature about GANs. In this section, we will pick a few of the best pieces so that you can learn how to apply GANs in the real world.

BiGAN – bidirectional generative adversarial networks

Bidirectional generative adversarial networks (BiGANs) were proposed by Jeff Donahue et al. (2017). The goal of BiGANs is to learn inverse mapping, which allows us to project the data into latent space. The traditional GAN only learns to generate a new data sample that looks like the training data, even though we know that GAN has the ability to learn the underlying structure of the data. This means that we cannot use the traditional GAN to extract useful feature representations of the data. BiGANs focus on solving this exact problem.

The structure of a BiGAN is shown in the following diagram:



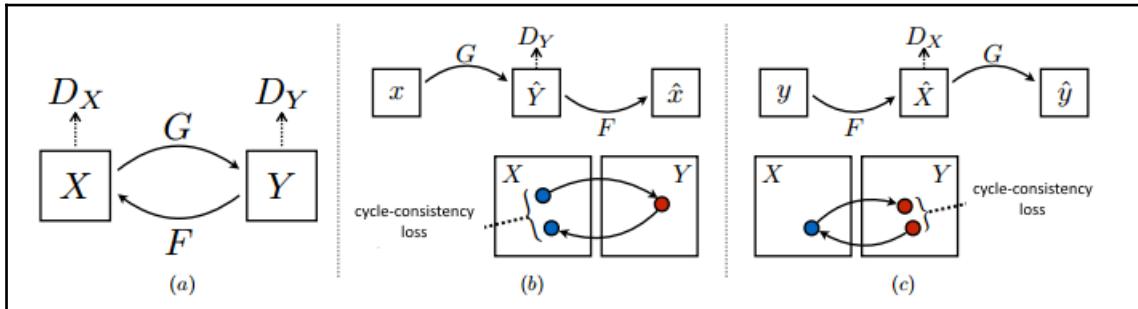
In BiGANs, an encoder E is added to learn how revert from the data to the feature space, $E(x)$. The discriminator D will need to discriminate not only the data space (x or $G(z)$), but also the data and the latent space ($x, E(x)$ versus $G(z), z$). In the literature of BiGANs, the authors show that the encoder and the generator must learn together to fool the discriminator, even though there isn't a direct connection between the generator G and the encoder E .

The most interesting thing about the BiGANs framework is that this is an unsupervised approach and it can still learn the feature representations of the data. The authors of BiGANs also show many examples of classification, detection, and segmentation in their literature to demonstrate how BiGANs can provide a representation that is capable of competing with other existing supervised approaches for visual feature learning.

CycleGAN

CycleGAN is a novel GAN variant that learns to automatically perform image-to-image translation. Given any two sets of image collections, X and Y , CycleGAN allows us to translate an image in set X to set Y so that the images from $G(X)$ are indistinguishable from the images in Y . An interesting idea is that this is unsupervised learning and we don't need a training set of aligned image pairs. For example, CycleGAN can learn to translate an image of a horse to an image of a zebra and vice versa.

The structure of the CycleGAN from Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks is shown in the following diagram:

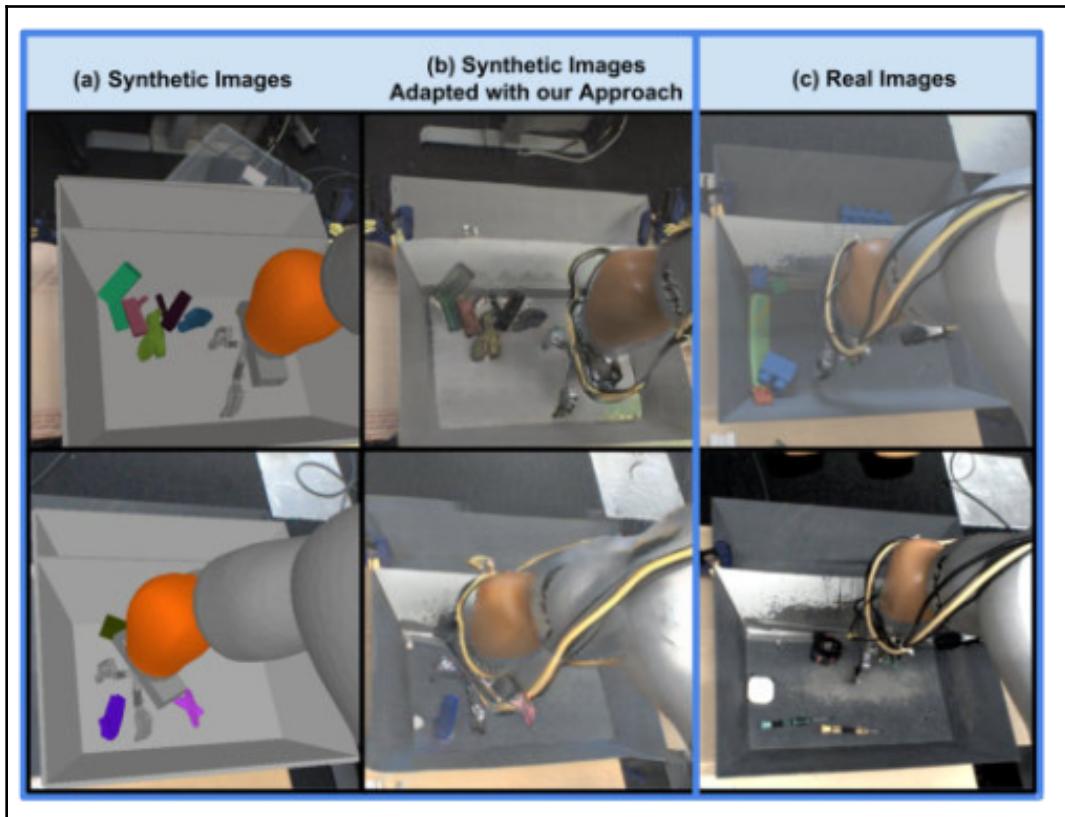


In CycleGAN, the authors introduced two mapping functions, $G: X \rightarrow Y$ and $F: Y \rightarrow X$. There are also two discriminators, D_X and D_Y . Besides that, they also added a cycle-consistency loss so that the image that we translate to the other domain and back again should be the same as the original image. You can learn more about CycleGAN in a paper called *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* by Jun-Yan Zhu et al. (Submitted on 30 Mar 2017 (v1 at <https://arxiv.org/abs/1703.10593v1>), last revised 19 Feb 2018).

GraspGAN – for deep robotic grasping

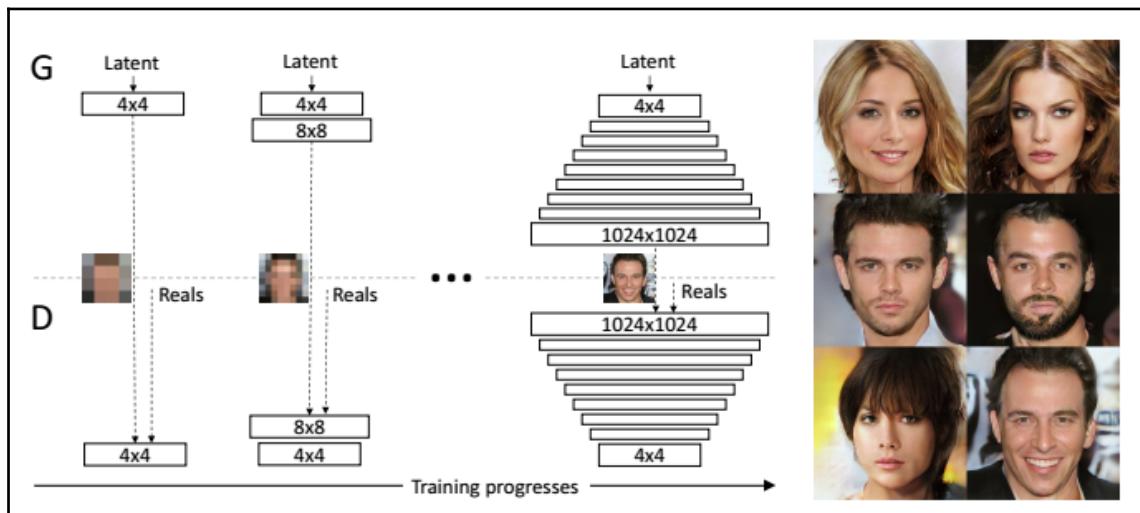
GraspGAN is another novel GAN variant in which GAN is applied to generate synthetic data of an environment based on a simulated environment.

The goal of this problem is to train a robot to grasp an object. One challenge this presents is that the large labeled dataset for training is too expensive to create. Therefore, the authors created a simulator and generated synthetic data that give as real an experience as possible. The following diagram, from a paper named *Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping*, shows synthetic images that look almost identical to the real images:



Progressive growth of GANs for improved quality

This is a new training approach for GANs that was proposed by Tero Keras et al. in 2018. This approach can generate realistic images with a size of 1024x1024, which is much larger than the previous approach. In this approach, instead of training both the generator and the discriminator directly to the 1024x1024 size, they start with a low spatial resolution of 4x4 pixels. They then incrementally add new layers to the generator and discriminator and start the training process. All the existing layers are still trainable throughout the process. The following diagram shows the training process as well as some realistic generated images on the right:



Summary

This is it for GANs for the time being, but there is much more to be discussed later on. We started our GAN journey by generating fake party tickets. To convince you guys that this was a tough task, we first looked at how GAN can help us generate fake 1D samples. We then went into detail by implementing step-by-step codes in our 1D GAN file.

We later dived into the world of images, where we unlocked the potential of CNNs. We used a CNN as a discriminator as well as a generator. As part of this discussion, we also went through the important concepts of transposed convolution and batch normalization. We used these concepts to create a very effective generator model, which was quite capable of generating MNIST digits by itself. Finally, we saw some unique varieties of GAN architecture that are quite popular within the field.

23

From GPUs to Quantum computing - AI Hardware

So far, we have talked a lot about different machine learning techniques. We have seen how AI has applications across various different domains, including FinTech, healthcare, and computer vision. For each of these applications, we have built several kinds of neural networks, starting from a very basic model to a much more complex one.

The complexity of our networks always depends on the type of problem we are trying to solve; for vision-related tasks, we can't get highly accurate outputs by using a simple neural network. We built a normal feed-forward neural network classifier for a MNIST digit classification problem and saw how a convolutional neural network beat its performance by quite a margin. For Bitcoin price prediction, we used an LSTM-based recurrent neural network. This shows that for a specific problem that we need to design a specific network.

In most cases, the complexity of the data is the main factor that affects the kind of network architecture we choose. However, if you don't have a graphical processing unit (GPU) or graphics card installed on your system, and instead opt to train a convolutional neural network on your PC, be ready to spend a long time training it – a normal VGG-16 convolutional neural network on a normal CPU might take three weeks before you can get any performance out of it! Don't be too hasty to dump your current system, however; you can rent a computer instance on Google Cloud to train your networks over Tensor Processing Units (TPUs).

If you haven't heard these terms before, this chapter will help you to learn about them. We will talk about the infrastructures used to solve ML-based problems. We will start with CPUs and then learn about different platforms, too.

As AI applications are becoming increasingly well known, there is a growing demand for faster processing systems, especially to process terabytes of data. Google, for example, processes around 40,000 searches worldwide per second. This is equivalent to around 3.5 billion searches per day and about 1.2 trillion searches per year, or several terabytes worth of data per day. You simply cannot process this much data using normal GPUs.

As software technology is advancing, hardware needs to be updated at the same pace or there will be a huge gap between a hypothesis and a practical solution. To overcome such gaps, industries are investing billions of dollars into research to do with hardware. Their results have been surprising. **Quantum Processor Units (QPUs)**, which can process data more than 100 million times faster than your PC or laptop, have now arrived at the market. We will talk about these in this chapter too, as well as the following topics:

- A brief history of computers.
- What CPUs and GPUs are
- What ASICs and TPUs are?

Computers – an ordinary tale

Since the invention of the computer, researchers have been working hard to make them more compact and faster than ever. The first computer was built by Charles Babbage somewhere between 1831 and 1871. The project to build a non-human computer was initiated by Napoleon Bonaparte – the same Frenchman who played a very important role in European history in the 1800s.

A brief history

The whole thing started in 1790, when Napoleon replaced the old measurement system with a new imperial measurement system. At that time, computers were not machines, but humans instead. Their work was to perform various calculations for different kinds of mathematical tasks, such as tax calculations. They used to record the calculations in the form of tables, which could be used for the reference of others. To convert from the old to the new measurement system, Napoleon used human computers for almost 10 years to convert the system into tabular form. However, he never got a chance to publish these tables; they later became a part of the French Academy of Science.

In 1819, Charles Babbage got a chance to view the unpublished tables. He came up with the idea of using machines to do these kinds of calculations. He created the difference engine, which was able to perform complex calculations with repeated additions. In 1832, he presented his table-making machines to the public. After this, he never looked back.

After some time, Babbage came up with an idea of creating a more complex computing system, the analytical engine. This was a revolutionary idea, because it is quite similar to modern computer systems. The analytical engine had two important components: a **central processing unit** and a memory element. At that time, the CPU was known as the **mill**, while memory was referred as the **store**. The third device that was added to this system to receive inputs was known as the **reader**.

Unfortunately, this system was only partially developed, because of its heavy weight and complexity, but the idea made Babbage the **father of computers**. In 1991, the Science Museum in London tried to convert this idea into a fully working form. The system was 11 feet long and 7 feet tall. It had 8,000 moving parts and a weight of 15 tons.

Alan Turing was a scientist and a mathematician of British origin. He is well known for his contributions in the field of computer science, which helped the Allied forces to win the Second World War. He invented a machine that could decode the various symbols hidden in German messages.

This allowed the Allied forces to know in advance about the next planned attack of their opponents. The Germans were using a machine that could encode a message into various symbols. The messages could only be decoded by another similar machine, known as the Enigma. To break these codes, Turing invented an electrical system that applied a lot of permutations and combinations to decipher the messages, otherwise known as a brute force attack. He named this machine Bombe.

Alan Turing is known as the father of modern computer science due to his concept that helped scientists to work in the direction of inventing a true computer. This hypothesis is often referred to as a Turing machine. The Turing machine was very simple and able to simulate any complex algorithm using just three symbols: 0, 1, and " " (a space). This is quite similar to modern binary systems. It therefore represents a crucial step towards making an electronic system that can receive information in the form of binary codes and that can process that information using simple instructions. This was considered a major breakthrough towards modern day computers and programming.

As you know, all computers designed up until the 1940s were actually various kinds of mechanical or electromechanical systems. There then came a major breakthrough by American physicist John Mauchly. He was the first person who came up with an idea of a digital computer. While his proposal was in progress, he met another brilliant man, John Eckert, who was able to design a hardware system for Mauchly's prototype. Together, they invented the first general purpose computer: the **Electronic Numerical Integrator and Computer (ENIAC)**. The ENIAC was used by the US army for their ballistic tests in 1947.

After the success of this computer system, Mauchly and Eckert started a computer manufacturing firm. Through this firm, they designed two more very popular computer systems, known as the **Binary Automatic Computer (BINAC)** and **Universal Automatic Computer (UNIVAC)**, which was the first commercial computer.

Central Processing Unit

CPUs are made of transistors, which work on very basic principles of current and voltage. You can use a transistor as a switch to block the signal for one voltage stage and pass it to another voltage stage. This property led scientists to use transistors in the creation of logic gates such as AND, OR, and NAND. It can be said that these logic gates are the heart and soul of our computer system because we can use them to create any kind of function.

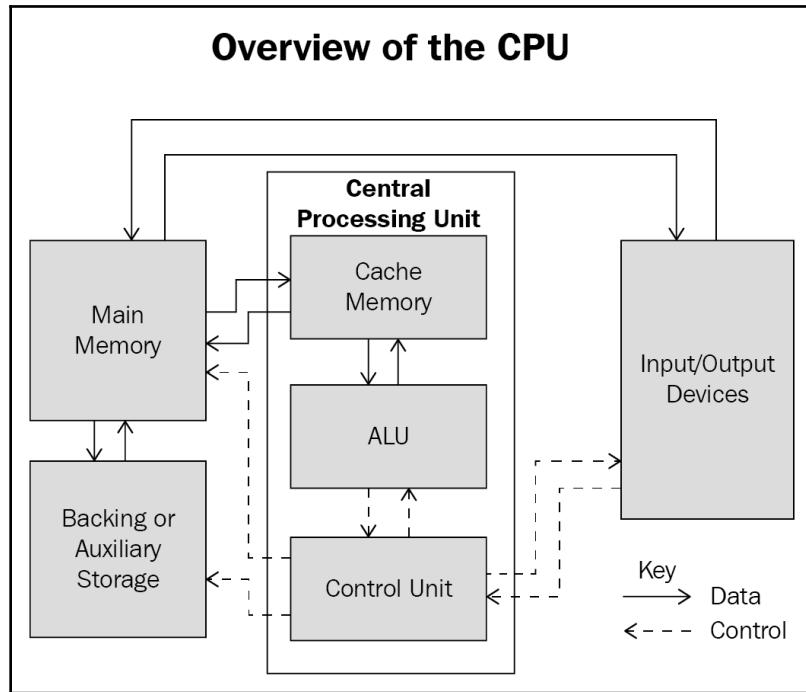
There are billions of logic gates being used in modern day computers. When CPUs were initially developed, the size of the transistors was quite large, which often made the CPUs larger than a room. They also consumed too much power. With the advancement of silicon wafer technology, the size of the transistors has become so tiny that you can put a billion transistors in an area that is less than a square centimeter. This eventually solved the problem of power and space consumption.

The number of transistors and the performance of the processor is proportional: if you want a high-performance processor, you need a lot of transistors. A computer uses a pulse generator to signal to the transistors whether they should be high voltage (1) or low voltage (0).

The processor performs operations that are proportional to the frequency of the pulses, which is known as clock frequency or clock speed, and is measured in Hertz. The higher the **clock speed**, the higher the speed of the processor.

Nowadays, we can find multiple processors on a single chip, which is known as a **multi core** processor. It allows us to perform parallel operations on the CPU, which helps to perform logical operations at a higher speed. The more cores you have, the faster you can perform operations.

The following is a schematic diagram of a Central Processing Unit:



As you can see, a CPU is made up of various parts. These parts include an **Arithmetic and Logical Unit (ALU)**, a control unit, the cache memory, the main memory, auxiliary storage, and input and output devices. We will mainly concentrate on the parts that are important for performing machine learning tasks. We will discuss the ALU and the control unit together as the *Processor* and then discuss the main memory or *RAM* and the auxiliary storage, or the **hard disk drive (HDD)**.

We mentioned that a processor performs various operations. But what kind of operation does it perform and how does it know what to do? This process is similar to ordering your favorite meal at a Subway outlet. You need to instruct them about what kind of bread you want, which vegetables you want, and what kind of sauce they should put in there. You need to do a similar kind of work with your processor. This is carried out by a **computer program**, which contains instructions for the various tasks that are to be performed by the computer. When you combine multiple programs in a single program, it is known as **software**. Every computer system has specific software that has the responsibility to manage various other software and hardware resources on the system. This software is known as the **operating system (OS)**.

Another important aspect of a CPU is the number of bits that it can read or write at a time. Intel's 8085 was the first general purpose programmable processor that was able to handle 8-bit data. After 8085's huge success, the company launched its successor, 8086, which was capable of handling 16-bit data. Nowadays, you can easily find 32-bit and **64-bit processor architectures**. A 32-bit system can work with a very limited amount of Random Access Memory (for Windows, around 4 GB of RAM) as it can address up to 32-bit addresses only. A 64-bit system, on the other hand, can address a much higher amount of RAM. My workstation, for example, has 256 GB of RAM.

RAM is another important part of computer systems. A computer's processor takes data from RAM, performs different operations on it, and stores the results on the RAM again. This connection is a duplex, or two way, connection. RAM is directly connected to the CPU with a pin to pin parallel connection, so the input and output operations between the RAM and the CPU are very fast. If you want to process large datasets, you need to have larger RAM size.

The next important item on the list is the **HDD**. This is where you store all of your data. Your computer system puts all the data (even the data to do with the operating system) in storage, and whenever it needs to process the data it first accesses the RAM and then the processor. This operation is a serial operation. Nowadays, Solid State Drives (**SSD**) are frequently used for data storage tasks instead. An SSD is smaller and lighter than an HDD, as it doesn't have any mechanical moving parts. A traditional HDD, on the other hand, uses a disk. SSDs are much faster than HDDs, but at the same time they cost more. They may well be the future of general purpose storage

While discussing the CPU, people often neglect a very special component: the **motherboard**. Nobody places much importance on the motherboard when buying a CPU, but remember that it is the motherboard that will hold all the components of your CPU, including, most importantly, the processor and the RAM. If your motherboard doesn't allow you to insert a higher memory RAM, even if your processor can work with it, that RAM is of no use for the system. The motherboard decides how many graphics cards can be added to your CPU, what kind of RAM you can use, and so on.

There are a few more key elements that are required to build a computer system, but we are not going to discuss these in this book.

As we have already seen, the development of computers all started in the 1940s. Since then, we have progressed a lot, as new inventions in hardware technology allow scientists to develop smaller and more powerful computer systems. We have now reached the era of smart devices that can easily be put into our pockets but are at least 10,000 times more powerful than the original, large and heavy systems. Since we have reached the point where we can perform a lot of computation on an ordinary laptop, it is worth mentioning that we can use these same laptops and PCs for training and deploying artificial intelligence applications too.

CPU for machine learning

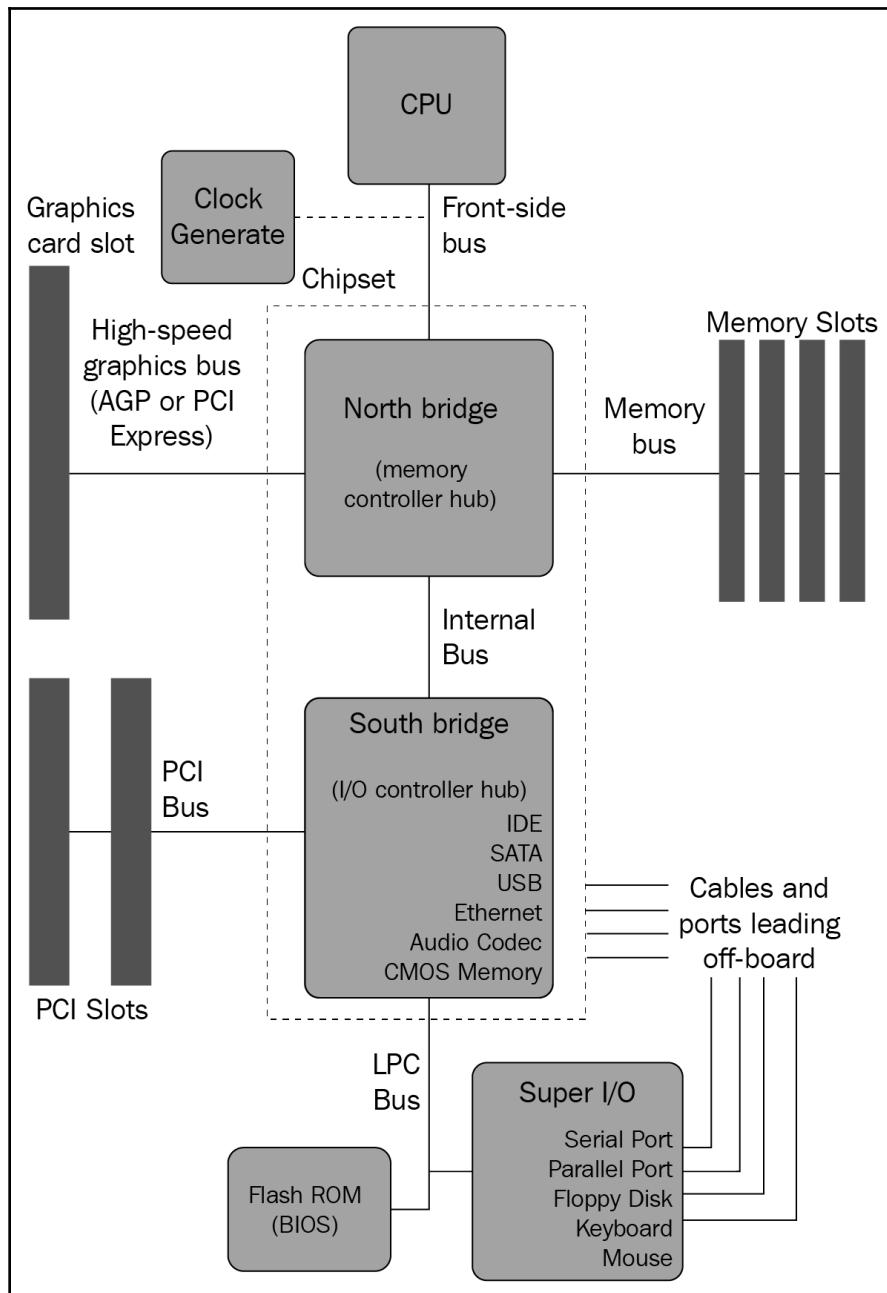
What we want to do is build a system for machine learning tasks. In the previous section, we discussed the four important components of a CPU: the processor, the RAM, the motherboard, and the HDD. To build a powerful machine learning system, we need to work mainly on these four components. Let's take a look at how to configure them properly.

Motherboard

We will start by selecting the motherboard. As the name suggests, the motherboard is the unit on which the whole CPU architecture is built. It consists of the following components:

- A memory controller hub, known as the North Bridge
- An input/output (I/O) operation hub, known as the South Bridge
- Slots for the clock generator and processor
- Slots to insert the RAM
- **Peripheral component interconnect (PCI)**, bus slots for the modem and other connections
- Graphic card slots, where you will put GPUs
- Some slots to connect the HDD, system ROM, keyboard, and mouse

The following diagram shows the schematics of a typical motherboard:



The motherboard may become a very important factor when you need to upgrade your system. If you haven't bought a motherboard that is future-oriented, you won't be able to upgrade two very important components that are required for ML algorithm development: the RAM or the **Graphics Processing Unit (GPU)**.

We will talk about GPUs later in this chapter. For now, let's think about how upgrading the RAM might be affected by the architecture of the motherboard. Suppose your motherboard has a single slot that can support up to 4 GB of RAM. In this case, you can't add more RAM, as there is no slot for it. You can't replace it with an 8 GB RAM, as it doesn't have the pin configuration for that. I am speaking from personal experience here; my laptop only had one RAM slot that could only handle up to 8 GB of RAM. If my laptop had a motherboard with two RAM slots with 8 GB for both the slots, I could simply put an 8 GB RAM into the empty slot, giving my system the power of 12 GB of RAM (4 GB + 8 GB).

If your motherboard doesn't have any slots to add a GPU card, you will never be able to train a deep convolutional neural network on your system. The motherboard is a very important component to select when you build an ML-based system.

I would recommend that you go for a motherboard that can provide at least two RAM slots with at least 16 GB of memory support. Another important factor is that your motherboard must have support for at least one more generation beyond the current one, so that you can upgrade your system after four years. This will save you a lot of money. Your motherboard should also contain at least two GPU slots with at least 8 GB of memory support. This allows you to use both GPUs at the same time for training or testing your model. You can train a very deep convolutional neural network on an 8 GB graphics card.

Processor

Let's talk about some of the specifications that are required to choose a decent processor for ML tasks.

Clock speed

As we have already discussed, a processor with a higher **clock speed** performs calculations at a much higher speed. The recommended clock speed for a ML system is higher than **2 Gigahertz (GHz)**. It is possible to have a clock speed of **1.7 GHz**, but anything lower and your system may struggle during function optimization (classifier training).

Number of cores

We have also already discussed **multi-core** systems. A multi-core system can help if your clock speed is not up to the recommended level. A multi-core processor basically has more than one CPU, and all the CPUs sync together to carry out different operations. The CPUs are connected with a scheduler that divides a task into multiple parts and provides each part with a core of the processor. This process is known as multi-threading.

For example, let's say you need to run a `for...loop` 1,000 times, and your system has four cores. You can load all four cores with 250 iterations, which will be performed simultaneously. At the end, you can add the results that have been gathered from all four cores.

Now, suppose you are training an ML model with a **Stochastic Gradient Descent (SGD)** algorithm. You can use multithreading for the optimization, meaning your classifier can be trained four times faster than a usual system. Your system must have at least **four cores** to utilize the power of multithreading.

Another factor that may affect the capabilities of the processor is its size and the amount of power it consumes. If the size of the processor is large, it will consume more power and you might find that the CPU generates too much heat, which degrades the performance of the system.

Architecture

You can work with either a 32-bit or a 64-bit CPU architecture. However, if you are going to buy a new system, I would recommend that you go for a 64-bit one. It can address a lot more memory than a 32-bit system. Another very important thing is that there is no 32-bit version available for **TensorFlow**. If you are looking for a system that you want to use for the development of ML applications, you must buy a 64-bit system.

RAM

As we have discussed, your system's processor always talks to the RAM for data input and output. This communication is in parallel, which is faster than serial communication. If your system's RAM can load a huge dataset at once, you can train a model directly on the whole dataset. If your dataset can't fit into the RAM, however, you need to use your system's storage and use small parts of the dataset to train your classifier. Data communication between the RAM and the storage is serial, which is slow. This will significantly affect the training time. Another problem with low RAM is that you cannot perform multithreading efficiently, as datasets can't be loaded into the memory at once.

I would recommend that you use at least 8 GB of RAM. If your motherboard supports one more slot, you can put a 2 GB or 4 GB RAM in the other slot, which increases the capability and performance of your ML system.

HDD and SSD

If you work on image datasets frequently, you might have faced a problem with storage. Usually, an image-based dataset occupies multiple gigabytes of space on the system's HDD. If you want to work with various different applications, the amount of storage you require will increase exponentially. Another activity that requires a huge amount of disk space is training the network weights of your deep neural network. When you train a model, you usually store a snapshot of the trained model for a fixed interval. This means that if you are training your model for 1,000 epochs and storing a model snapshot after every 50 epochs, there will be 20 copies of the model on the disk. If a model occupies around 150 MB of space, these 20 models would take up around 3 GB of space.

I would recommend that you have at least 1 TB of space on the HDD so that you can use it for whatever you want. Nowadays, storage devices are much cheaper than before, so you can easily add 1 or 2 TB of HDD on to your system.

If you want to add an SSD to your system, the recommended size is 500 GB, but you must have at least 250 GB. This will help you in faster input and output operations, which eventually help with faster training and testing.

Operating system (OS)

It doesn't really matter which OS you use. Almost every machine learning framework is available for major operating systems such as Windows, Linux (and Ubuntu), and Apple Mac OS X. Which OS you choose is entirely up to you; some people are more comfortable on a Windows OS, while others prefer Ubuntu or Linux.

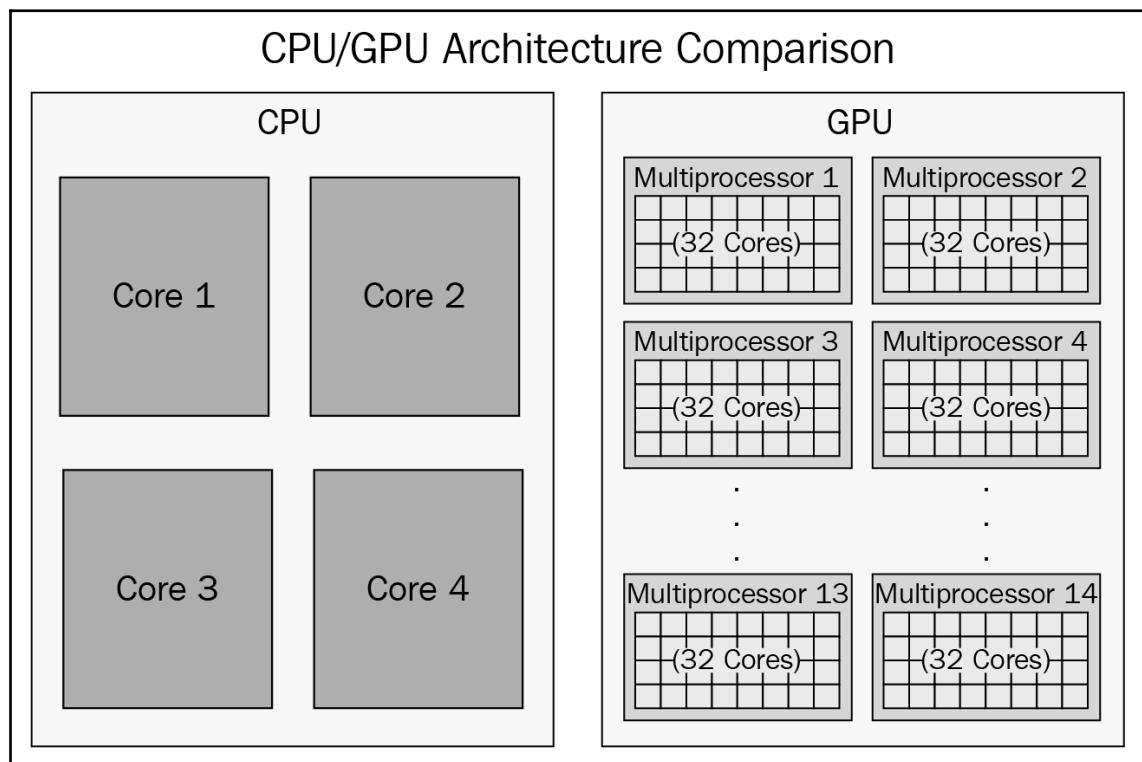
However, I would add that if you go for a MacBook, it does have some hardware upgrade limitations. Ensure that you carry out the appropriate research before investing your money in a Mac.

Some people find that it is easier to work on convolutional neural networks on Linux and Ubuntu because it gives them the flexibility to work with different deep learning frameworks such as Caffe, TensorFlow, PyTorch, and so on. I personally find it a bit difficult to build a Caffe library on Windows. You can solve this, however, by using the Docker toolbox. We will look at how to use Docker in Chapter 24, *TensorFlow Serving*.

These are the basic CPU hardware requirements needed to design a ML development system. Nowadays, machine learning is in quite an advanced stage, so you may frequently work on applications where convolutional neural networks need to be deployed. To train and deploy a CNN, you need to have one or more GPUs. We will discuss this further in the next section.

Graphics Processing Unit (GPU)

When we talked about CPUs, we looked at what a multi-core processor is. You can easily find a workstation that has around 20 cores and that does its work quite efficiently. If you want more cores, however, a normal, dedicated GPU might have at least 100 cores, which can run thousands of threads in parallel. A GPU for dedicated, parallel tasks might have tens of thousands of cores, which have the capability to run a program at least 500 times faster than on a CPU. The following diagram shows the typical difference between a CPU and GPU:



Earlier GPUs were developed to process 3D graphics, especially for enhancing gaming experiences. They were used for graphics rendering and game physics calculation, which required extensive computations. To render data frames at high rates, from 30 to 60 frames per second, there is lot of processing power required, which would be difficult for a CPU to handle. In a GPU, each core works as a worker. We can assign a task to multiple workers, similar to the multithreading method that we saw earlier. Each thread we create executes on a core. This greatly enhances processing power.

Are GPUs expensive? This is the interesting part. Before GPUs started to be used for general purpose work, people used clusters of multiple CPUs to achieve parallelism. To do this, they needed to set up a proper environment, which requires quite a lot of space and power to run the whole cluster. This cost a lot of money. GPUs, however, are a bit different. A GPU with more than 1,000 cores can easily be put into a normal CPU's motherboard, and it can run on the same power supply as that on which your computer's processor is running. This makes it both a space- and power-efficient solution.

Let's get back to the original question regarding its cost. GPUs are much cheaper than a cluster of computers. As a GPU is only designed to perform parallel tasks, it cannot be used as a CPU, so there must be a CPU to instruct and schedule the tasks for the GPU. A GPU is nothing but a slave of the CPU; it follows the CPU's instructions. This is why GPUs are cheaper. You can easily get a GPU with more than 1,200 cores for less than \$300.

GP-GPUs and NVIDIA CUDA

I once went to a computer shop to buy a GPU-enabled laptop for programming purposes. The shopkeeper asked me, *What kind of game do you want to play?* It took me an hour to convince him that GPUs can be programmed. Nowadays, things have changed quite significantly. GPUs are far more often involved in extensive computational tasks than gaming. These GPUs are now known as General Purpose Graphics Processing Units (GP-GPUs). A GP-GPU is simply a plug-and-play device. All you need to do is put it into your computer's motherboard slot, download its driver software, and then you are ready to go.

NVIDIA, a computer hardware company that works primarily with GPU technology, makes GPU programming very easy. They have developed a C-based application interface to program GPUs simply. This API is known as **Compute Unified Device Architecture (CUDA)**. It is a parallel computing platform. The CUDA platform can be seen as a software layer that gives you direct access to your computer's GPU.

As it is a C-based API, its instruction set is very similar to C. If you like to program in C or C++, you will find it very easy and interesting. This API is also available for Java (as a JCUDA wrapper), Python (as a PyCUDA wrapper), and Fortran developers.

The CUDA Software Development Kit (SDK) was launched in early 2007. Since then, it became the first choice for GPU programmers with support for all major operating systems, including Windows, Linux (Ubuntu and Red Hat), and macOS X. However, this API does have one limitation: it can only be used with NVIDIA GPU cards from the G8x series. Despite this, CUDA has many advantages for computation over traditional GPUs APIs, including the following:

- It has a scattered read feature that enables you to read an arbitrary memory address from code.
- It contains various libraries that can be used not only for computations but for various debugging and profiling purposes. It also contains various compiling tools.
- It has a unified memory architecture.
- Multiple threads can share the same memory location.
- It enables faster IO operations on GPUs.
- You can perform various logical (bitwise) and integer arithmetical operations, which also includes integer texture lookups.

There are other APIs also available for GPU programming such as OpenCL and OpenGL. CUDA is much easier to use than the others, as it gives you direct access to your GPU architecture in terms of memory and processing controls.

cuDNN

CUDA Deep Neural Network (cuDNN) is a CUDA-based, GPU-accelerated library. It allows you to use your GPUs for ML purposes. cuDNN is supplied with well-tuned implementations for usual routines, which include various kinds of convolution operations (both forward and backward), pooling, normalization, and different kind of activation layers. cuDNN is part of the NVIDIA Deep Learning SDK (<https://developer.nvidia.com/deep-learning-software>).

Various deep learning researchers and different machine learning framework developers rely on cuDNN to get high-performance GPU acceleration. cuDNN allows them to focus on training neural networks and tuning their architectures for the development of different software applications rather than spending time on GPU performance tuning using programming. cuDNN accelerates widely used deep learning frameworks, such as Caffe2 (<https://www.caffe2.ai/>), MATLAB (<https://www.mathworks.com/solutions/deep-learning.html>), Microsoft Cognitive Toolkit (<https://www cntk ai>), TensorFlow (<https://www.TensorFlow.org/>), Theano (<http://deeplearning.net/software/theano/>), and PyTorch (<http://pytorch.org/>).

To use cuDNN at its full capacity, you must install a compatible version of CUDA. Both the libraries – CUDA and cuDNN – can be downloaded easily from NVIDIA's official website.

The following are the key features of the cuDNN library, which can be found on NVIDIA's official website:

- Forward and backward paths for many common layer types such as pooling, LRN, LCN, batch normalization, dropout, CTC, ReLU, Sigmoid, softmax, and Tanh.
- Forward and backward convolution routines, including cross-correlation, which is designed for convolutional neural nets.
- LSTM and GRU Recurrent Neural Networks (RNNs) and Persistent RNNs.
- Forward and backward passes using FP32 and FP16 (Tensor Cores) data types and forward passes using UINT8 (Volta and later).
- Arbitrary dimension ordering, striding, and sub-regions for 4D tensors. This means that it integrates easily with any neural net implementation.
- Tensor transformation functions.
- A context-based API, which allows for easy multithreading.

What if we want more speed? Google has developed a solution for this: **Tensor Processing Units (TPUs)**. We will look at these closely in the following section.

ASICs, TPUs, and FPGAs

When ML developers found that training highly complex and extremely deep neural networks was a bottleneck for developing ML applications, Google came up with a solution. They developed TPUs to help them achieve high speed optimizations.

Google introduced TPUs for the first time in 2016. A TPU is a proprietary processor that is specifically designed for the development of ML applications using neural networks. It is a type of **Application-Specific Integrated Circuit (ASIC)**. Let's briefly discuss ASICs in the next section.

ASIC

As the name suggests, ASICs are special circuits that are designed to serve a specific type of work, making them difficult to use for the development of general purpose applications. ASIC is not a new term; it was introduced in the 1980s when Sinclair Research, a British Consumer Electronics Company, started using gate array circuitry for commercial purposes. It used to be a simple 8-bit system that performed various logical operations for switching.

As the embedded industry has grown and various design tools have improved over the years, the complexity and functionality that is possible in an ASIC has grown from around 5,000 logic gates to more than 100 million logic gates on a single chip. Modern ASICs may include entire microprocessors, different memory blocks including ROM, RAM, flash memory, and the other building blocks of a computer system. An ASIC is also called a **system-on-a-chip (SoC)**. Designers of digital ASICs use a specific hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

Nowadays, ASICs are being used for making various consumer electronics products such as smartphone chips, smart TVs, Android boxes, audio processing units, and so on. The bitcoin miner system is an example of a complex ASIC system. ASICs used for Bitcoin mining are specifically designed motherboards and power supplies that are constructed into a single unit. It's not just a purpose-built machine, it's a purposefully designed and specifically developed hardware, starting at the chip level.

Generally, mining is the process of running extremely complex calculations in the search of a specific number. It is similar to searching for a random number. A mining hardware, whether it's an ASIC miner or a GPU-based mining rig, needs to run through millions of calculations before getting to that magical number. In systems such as Bitcoin, the first person who finds that number gets a reward in the form of a Bitcoin.

ASIC miners are different from a graphics card—or a CPU-based mining system. GPUs and CPUs are more general kinds of hardware that are designed to do more than one task. When the task is to mine cryptocurrencies, all that really matters is that the currency you are going to mine must be worth more than what you are going to spend on hardware and power consumption. These margins might be thinner than you think, because mining cryptocurrencies can be very expensive. The hardware for this task can be costly to buy upfront and some of the available systems for mining tasks cost thousands of dollars per year for the electricity required to run them.

The choice of mining hardware depends heavily on how efficient the systems are in terms of cost as well as power consumption. Here is where ASIC miners come into the picture. Since they are designed to perform high speed calculations, ASICs are well suited for tasks that require you to solve specific cryptographic hash algorithms that can be used by an individual or a group of people. ASICs are very powerful and offer a high hash rate in an energy efficient way; they use far less power than a general purpose machine.

The combination of high performance and low power consumption makes ASICs a much more economical choice than more general purpose hardware. This is why, in the case of various cryptocurrencies such as Bitcoin and Litecoin, ASIC mining is the most appropriate technique.

Now, let's get back to our topic of interest: TPUs.

TPU

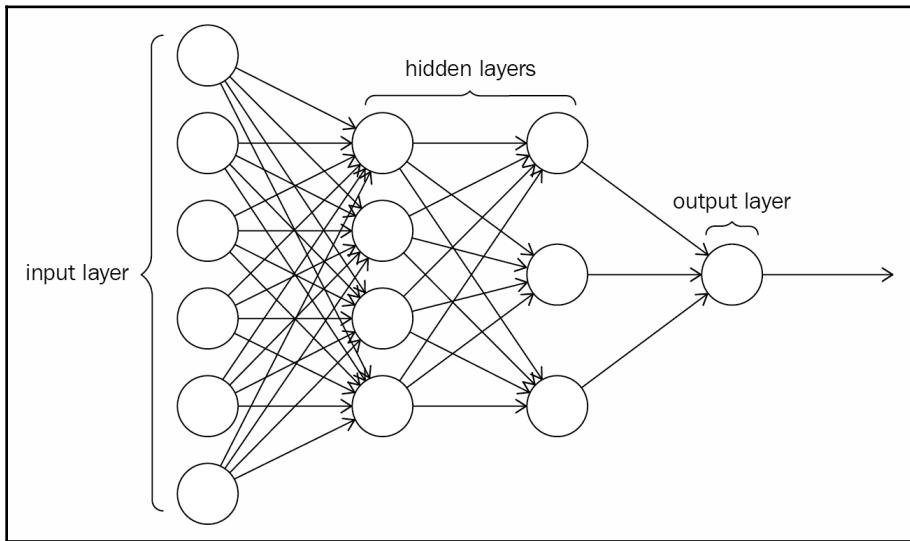
As we have already discussed, TPUs are types of ASICs that are specifically designed to carry out neural network-based machine learning tasks. They are not available to an individual directly, but you can use them as a cloud-based service. Google has used TPUs to play games, specifically GO; their ML system beat the world champion of the GO game Lee Sedol. They have also used TPUs to play rule-based games such as chess and Shogi. Google's ML system uses a reinforcement learning-based system to learn and play the game. It might surprise you that an individual TPU can process more than 100 million photos in Google photos. Google also uses TPUs for their search engine RankBrain to provide search results.

Technically, a TPU is designed to perform low precision computations, mainly of 8-bit precision. When compared to a GPU, TPUs have higher input/output operations per second (IOPS) per watt. They also don't have any kind of hardware component for rasterization or texture mapping, which is used for 3D data rendering, especially for games. This allows TPUs to deliver over 30 times the performance of GPUs, which is remarkable. The speed optimization of TPUs allows them to run state-of-the-art deep neural networks, specifically recurrent neural nets and convolutional neural nets, which require extensive computations.

Norm Jouppi, who is the tech lead of the TPU project, says that Google started working on ASICs for neural networks in 2006, but it didn't become urgent until 2013. After that, the team worked hard on the project. Jouppi also says that an ASIC development usually takes years, but because the team worked so hard in this case, they were able to design and deploy the system within 15 months.

Why did TPUs become such a world-changing phenomena in the development of neural network-based applications? First, let's think about how neural networks predict outputs step by step.

Consider the following feed-forward network, which has two hidden layers: one input, and one output layer:

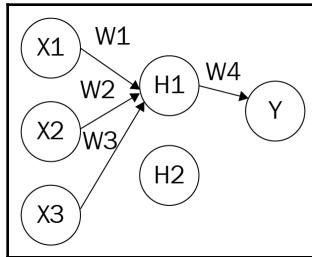


The connections that you can see between one layer of neurons to the next are weighted connections. This means that when one neuron from the first layer passes any value to the next neuron, it will be multiplied with the connection's weight (W), and it will pass it to the next layer. The multiplication of input values and weight values is straightforward:

$$y = W * X + b$$

Here, W is the weight of the connection and b is the bias, while X and y are the input and output values respectively.

When neurons from the hidden layer receive an input from the previous layer, this is the sum of all weighted connections from the previous layer to the hidden layer. This operation is shown in the following diagram:



The previous diagram shows the calculation of inputs received at H1. If we expand this, we get the following equation:

$$H_1 = X_1 * W_1 + X_2 * W_2 + X_3 * W_3$$

We can summarize this, as shown in the following equation:

$$H_{jl} = \sum X_{il-1} * W_{kl}$$

Here, i and j are the i th and j th neurons of the layers $l-1$ and l , respectively. k is the k th weight between the layers $l-1$ and l . This equation allows us to estimate the input of any neuron in the hidden layer.

Let's talk about the output of neuron H1. This should be the sum of the product, as we saw in the previous paragraph. However, there is one problem with this: if we choose the sum of the product as the output of any neuron in the network, it might be a very large number or it might be a negative number. To keep the number within a defined range, we will use an activation function that will transform the input number to within a defined range. There are many types of activation functions available, such as sigmoid (logistic), tanh, softmax, and so on. An activation function must be differentiable. This is because gradient-based methods, such as gradient descent, are involved during the weight-learning process, so we need to calculate the derivative of the activation function.

After applying the activation to the neuron, the output will change as follows:

$$H_{jl} = \text{activation} \left(\sum X_{il-1} * W_{kl} \right)$$

These operations will be applied to each neuron repetitively. We can summarize the whole process in the following three operations:

- **Multiplication** between the weights and the inputs
- **Addition** of all the multiplications of the current layer before aggregating to the next layer
- Applying an **activation** function to the summed value

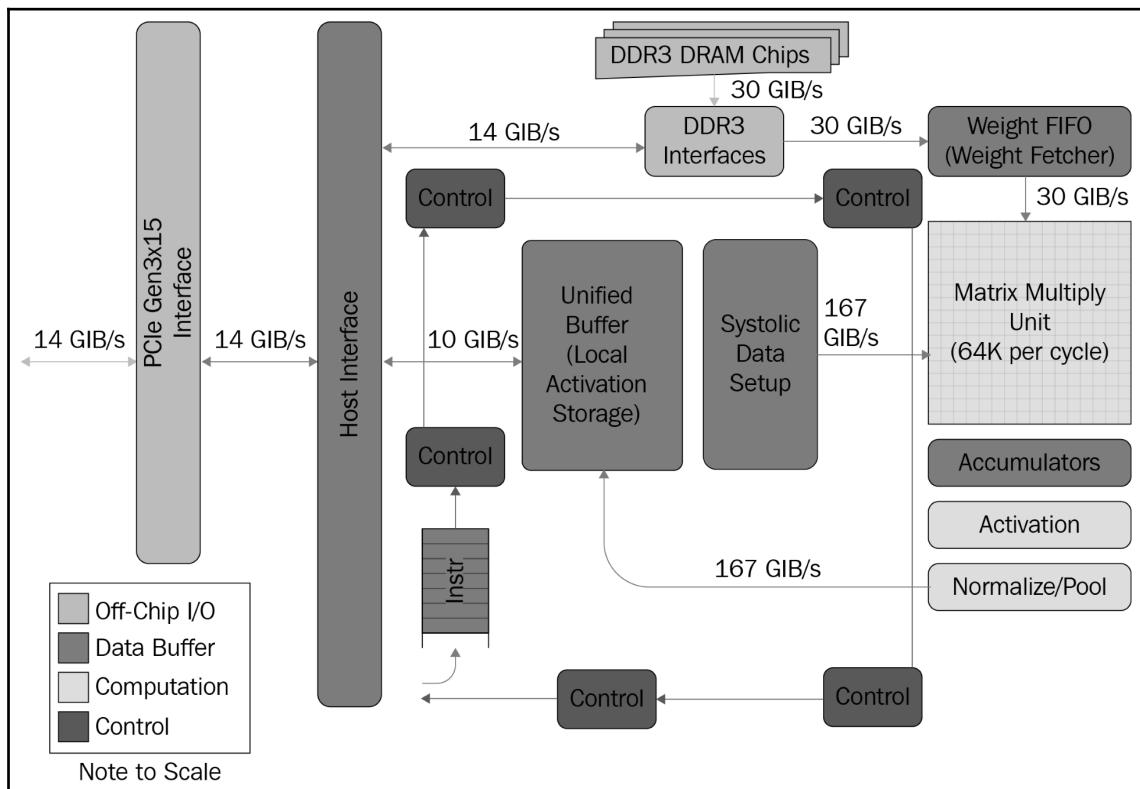
As you can see, there are three multiplications, three additions, and one activation operation performed for each neuron when three input nodes are provided. Google has looked at different LSTM networks, CNNs, and MultiLayer Perceptron (MLP) models to check how many weights are required for processing the output. These results were quite shocking. They are summarized in the following table:

Network type	Number of layers	Number of weights
MLP0	5	20 million
MLP1	4	5 million
LSTM0	58	52 million
LSTM1	56	34 million
CNN0	16	8 million
CNN1	89	100 million

As you can see in the previous table, for a five-layer MLP network, the number of weights is 20 million, while for a CNN with 89 layers, the number of weights is 100 million. Think about how much computation is required to get predictions from these networks.

A normal CPU or GPU performs floating point calculations that incorporate a high level of precision (16- or 32-bit). This requires quite a lot of memory and processing power. The TPU, on the other hand, uses a method named **quantization** to reduce this 16-bit or 32-bit precision value to an 8-bit precision with a negligible degradation in the output's performance. This reduces the total amount of memory and calculation by a huge margin. This makes the TPUs much faster and more power-efficient than CPUs and GPUs.

The following is a block diagram of a TPU:



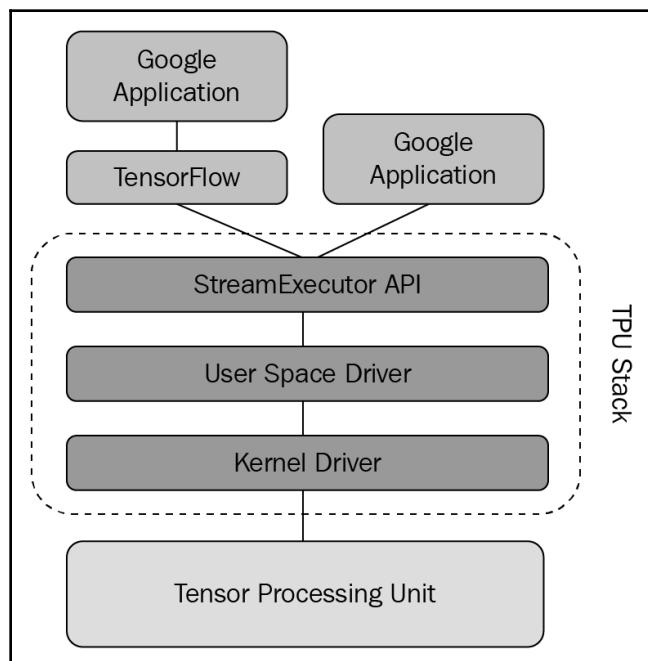
A TPU uses the following resources for computation:

- **Matrix Multiplier Unit (MXU)**: 65,536 8-bit multiply-and-add units for matrix operations
- **Unified Buffer (UB)**: 24 MB of SRAM that work as registers
- **Activation Unit (AU)**: Hardwired activation functions

To control these units, Google has defined an instruction set for various different neural network inferences. The following table shows a few of these instructions:

TPU instruction	Operation
Read_Host_Memory	Reads data from memory
Read_Weights	Reads weights from memory
MatrixMultiply, MatrixConvolve	Multiplication and/or convolution operation
Activate	Applies activation functions
Write_Host_Memory	Writes results to host memory

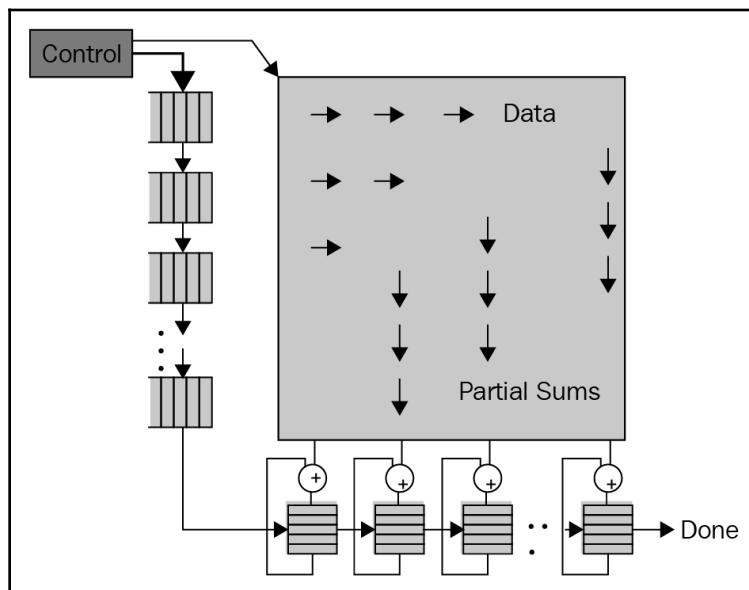
Google provides two different ways that you can access TPUs from your local system. You can either use the TensorFlow library, which passes network graphs and sessions to the TPU, or you can use its API. The following is a software stack from the local machine to the TPU:



Systolic array

The term systolic array comes from the human heart's systolic cycle. The reason why Google uses this name is quite interesting. To understand this, let's first look at how a CPU performs calculations on an array. The CPU stores values in different registers (memory units) and, using its instruction set, the ALU of the CPU accesses different registers to read the values and perform the different arithmetic operations. These instructions also tell it where to put the result after the computation. As the CPU performs these operations serially, it reads, operates, and writes the values sequentially. These operations and components require a lot of power as well as chip area.

An MXU, on the other hand, reads an input value once and reuses it for many different operations without storing that in the registers. As TPU consists of many ALUs connected in a cascade, this requires less chip area and latency in passing the values from one to another, which makes it energy efficient. This design is known as systolic because the data flows in the entire chip in waves, similar to the way in which our heart pumps blood. The specific type of systolic array in the MXU is optimized for energy and area efficiency for performing matrix multiplications and it is not recommended for general-purpose computation. This represents a kind of engineering trade-off: limiting registers, control, and operational flexibility in exchange for efficiency and a much higher operation density. Data flows in an MXU in the following manner:



The TPU Matrix Multiplication Unit has a systolic array mechanism that contains a total of 65,536 ALUs (256x256). This means that a TPU can process 65,536 multiply-and-adds for 8-bit integers every cycle. Because a TPU runs at 700 MHz, it can compute 4.5×10^{13} multiply-and-add operations ($65,536 \times 700,000,000$) or 92 Teraops per second (92×10^{12}) in the matrix unit. This enables a TPU to perform up to **128,000 operations per cycle**; while during the same cycle, a GPU can perform around 10,000 operations, and a CPU can perform barely 10.

This makes TPUs very powerful and efficient. If you do not have a high performance GPU card in your system, go for Google Cloud and rent a TPU to develop your application.

There is one more term that is quite popular in the machine learning field: **FPGA**. Let's take a look at what a FPGAs are and how they are different from ASICs.

Field-programmable gate arrays

Field-programmable gate arrays (FPGAs) are not new in the field of computer hardware. They have been in existence for almost the same amount of time as ASICs. Just like ASICs, FGPAs are also dedicated, application-based, and customizable hardware. They can be thought of as semiconductor devices that contain programmable logic components. These components are known as logic blocks. Logic blocks have programmable interconnections. This means that you can program the connections between different logic blocks using a **hardware description language (HDL)**. Programming the interconnection is a one-time process. Once you have burnt the interconnections between logic blocks, you cannot change them.

FPGAs for machine learning also use low precision computations, similar to TPUs, which enables them to achieve higher speed optimizations over GPUs. Recently, Intel have entered into the field of FPGAs and have built some of their own, including Intel Arria10 and Stratix10. These have outperformed NVIDIA's Titan X Pascal GPUs by using low-precision computations.

Right now, the biggest challenge for FPGAs to compete with GPUs and TPUs is the unavailability of a dedicated machine learning library, as CUDA-based cuDNN can be used for GPU-based ML development, while the TensorFlow API can help us use TPUs.

While ASICs are currently beating FPGAs in terms of speed and optimization at various levels of application development, recent developments from big companies like Intel and Microsoft in the field of FPGAs show that they have a promising future too.

We have talked about computers of the past and computers of the present. It is now time to talk about computers of the future. Let's discuss quantum computers.

Quantum computers

The term quantum comes directly from physics. It mainly deals with multiple states of particles at the same time. If you learned about the property of light in your physics class, you might remember the dual nature of light. It can be seen as particle (photons or packets of light) and a wave at the same time.

Scientists are trying to transfer this theory from physics to the computer world. As you know, computers work using binary numbers, where a bit can either represent a 1 (ON) or a 0 (OFF). It cannot have both these representations at the same time. We have seen that a basic building block in the computer system is a transistor. Researchers are now planning to reduce the size of a transistor so that it is small enough that the rules of atoms can be applied to it. According to Richard P. Feynman, one of the greatest scientists of the 20th century, *Things on a very small scale behave like nothing you have any direct experience about... or like anything that you have ever seen.* (*Six Easy Pieces*, p116.).

As you know, the basic elements of a general computer are registers, bits, different logic gates, and so on. These work on programs or algorithms that are written by humans. A quantum computer is equipped with the same kind of elements as an ordinary computer. Instead of bits, however, a quantum computer has quantum bits, or **qubits**. Qubits work quite differently from bits in a normal computer. A qubit can store either a 0 or a 1, or it can store both at the same time. This is made possible due to the concept of superposition in physics, which states that any wave that is created with the combination of many different waves contains information from all of the original sources. Qubits uses the principle of superposition to represent multiple states in the same way.

If this computer has multiple states at the same time, it can process them simultaneously, which is similar to processing information in parallel. Estimates suggest that due to this parallelism, a quantum computer would be able to process information a million times faster than an ordinary computer.

Can we really build a quantum computer?

In practice, there are a few ways of containing atoms and changing their states using different equipment, such as laser beams, electromagnetic fields, or radio waves. One method is to make qubits using quantum dots, which are nanoscopic tiny particles of semiconductors, where individual charge carriers, such as electrons and holes, can be controlled.

Another method is to make qubits from what are called **ion traps**, where you add or take away electrons from an atom to make an ion. You then hold it steady in a kind of laser spotlight and flip it into different states with laser pulses. Alternatively, we can treat qubits as photons inside **optical cavities**, which are spaces between extremely tiny mirrors. This is also the principle behind lasers.

Since the entire field of quantum computing is still largely abstract and theoretical, however, the only thing we really need to know is that qubits are stored by atoms or other quantum-scale particles that can exist in different states and can be switched between them.

The question remains, then, how far are we from the quantum realm?

How far are we from the quantum era?

Quantum computers are still largely theoretical. Even so, we have made some encouraging progress toward understanding a quantum machine. There were two important breakthroughs in 2000. First, Isaac Chuang (<http://feynman.mit.edu/ike/homepage/index.html>), an MIT faculty member, used five fluorine atoms to make a simple, 5-qubit quantum computer. In the same year, researchers at Los Alamos National Laboratory figured out how to make a 7-qubit machine using a drop of liquid. Five years later, in 2005, researchers at the University of Innsbruck added an extra qubit and produced the first quantum computer that could manipulate a qubyte (eight qubits).

These were quite promising and important first steps. In the next few years, researchers from various research labs announced more experiments that added greater numbers of qubits in the system. By 2011, a Canadian company called **D-Wave Systems** (<https://www.dwavesys.com/>) announced that it had created a 128-qubit quantum computer. Three years later, in 2014, Google announced that it was going to create a team of academics to develop its own quantum computers based on D-Wave's approach. In March 2015, the Google team announced they were *a step closer to quantum computation*. In 2016, MIT's Isaac Chuang and other scientists from the University of Innsbruck revealed a 5-qubit, ion-trap quantum computer that was able to calculate the factors of 15. All of this progress shows that one day, a scaled-up version of these kinds of machines might evolve into a fully-fledged encryption system.

There's absolutely no doubt that these achievements of the field are hugely important, and the signs are growing slowly toward more encouragement that quantum technology will deliver a computing revolution when the time comes. In December 2017, Microsoft introduced a complete quantum development kit (<https://blogs.microsoft.com/ai/future-quantum-microsoft-releases-free-preview-quantum-development-kit/>), which included a new computer language, Q#, that was developed specifically for designing and developing quantum applications. In early 2018, D-wave announced plans to start rolling out quantum power to a cloud computing platform, which is revolutionary. A few weeks later, Google announced Bristlecone (<https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html>), a quantum processor based on a 72-qubit array.

All of this is very exciting. Even so, these are very early days for the whole field, and many researchers have agreed that there is still more to be done in the field and that it may take decades to get a practical quantum computing-based system.

Summary

We have learned quite a lot about modern day hardware that could be revolutionary for the field of AI and machine learning. We started right from the beginning, when the idea for a machine arose in the mind of Napoleon. We saw how Charles Babbage came up with his difference and analytical engines, which cleared the path for researchers to make a fully functional computer system. We learned about various computer hardware details such as Processors, RAM, HDD, and so on. We also saw the recommended configuration to build a ML application development system.

Then, we moved on to look at GPUs and learned why they are very famous in ML-based solutions. We learned about NVIDIA CUDA, a GPU programming API, and cuDNN, which is used by a huge community of researchers for the development of neural network-based solutions. We also seen the recommended hardware requirements to build a GPU-based ML application development system.

Later, we moved on and looked at much faster cloud-based systems: TPUs. We learned about the basic workings of TPUs and also learned about ASICs. We saw how ASICs can be useful for ML applications. We have also discussed a little bit about FPGAs and how they are evolving into the current trend.

Finally, we learned about quantum computers, which might have seemed quite difficult if you are not from a physics background. We looked at the basics of quantum computing and what kind of promises it has to offer.

There is a lot of progress that is happening in terms of hardware and software in the industry, leading us to a better lifestyle. Only time can tell where we will be heading in the future. Let's hope for the best and keep learning.

In the next chapter, we shall focus on how to deploy our trained models on servers so that the majority of people can use our solutions. We will learn about TensorFlow Serving and we will deploy a very simple model on a local server.

24

TensorFlow Serving

In this chapter, we will take a close look at the following topics:

- What is TensorFlow Serving?
- Installing and running TensorFlow
- Operations for a serving model
- What is gRPC?

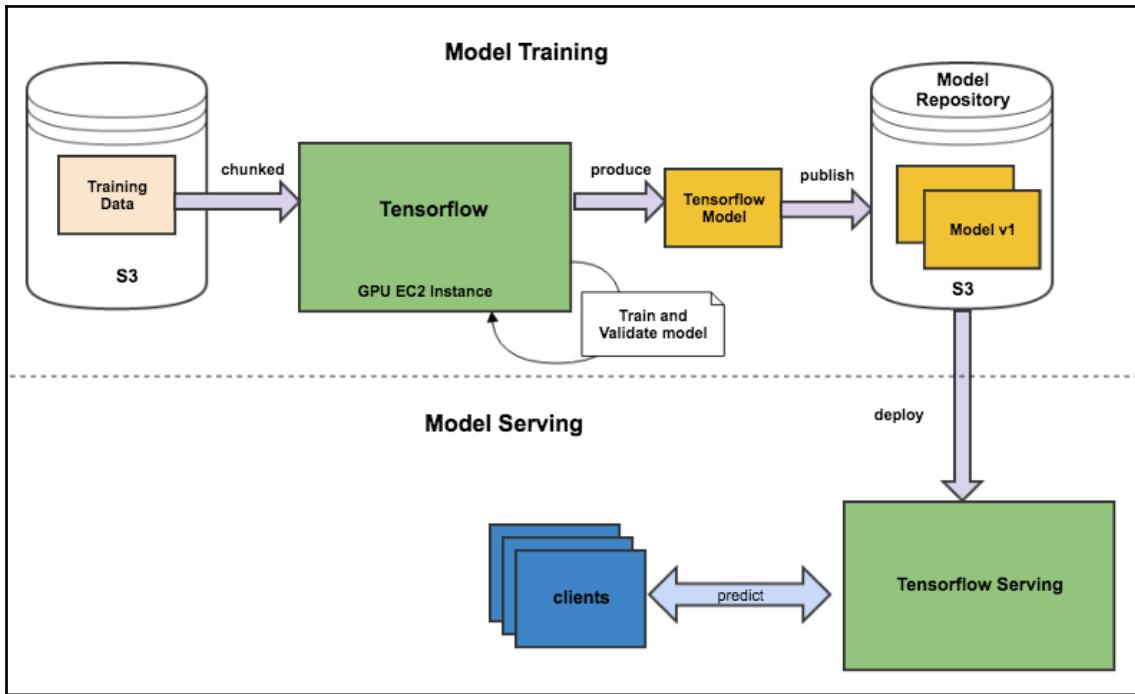
What is TensorFlow Serving?

In Chapter 22, *Gene*

rative Adversarial Networks, we looked at how train deep learning models for various applications. Have you ever wondered how we can deploy these models so that our applications can be useful for others, too?

TensorFlow provides a perfect solution for this kind of issue. We will use TensorFlow Serving (<https://www.tensorflow.org/serving/>), which is a TensorFlow library that makes it quick and easy to deploy our models in a production environment. It is not too different to how we would deploy any other service. It also works with different TensorFlow models out of the box without much modification.

The following diagram shows how this works:



We always start by generating training data and training the model using this data. We will then use the TensorFlow Serving library to deploy the trained model to the clients.

Before we move ahead to the deployment, we need to learn some terms that will be used during the model's deployment.

Understanding the basics of TensorFlow Serving

The following concepts are essential if you want to understand the architecture of TensorFlow Serving. You can find an in-depth explanation of these concepts on the TensorFlow Serving official page (https://www.tensorflow.org/serving/architecture_overview), but I have made sure to simplify them here so that you can get a better understanding of the architecture.

Servables

A servable is a TensorFlow session that is created during the training or validation of a dataset. Alternatively, it may be some kind of lookup table that is used for word embedding or vocabulary.

Servables are the central reverie in TensorFlow Serving. They are the fundamental objects that clients use to perform computation. This could be in the form of a lookup or inference of a model.

The size and coarseness of a servable is flexible in nature. It might include anything from a single fragment of a lookup table to a single model or a tuple of inference models. Servables may be of any type and any kind of interface, which allows you to implement future improvements such as the following:

- Streaming different results
- Different experimental APIs
- They can be used in asynchronous modes of operation

Servables cannot manage their own life cycles.

Typical servables may include the following:

- A TensorFlow `SavedModelBundle`, which is a session instance (`TensorFlow::Session`)
- A lookup table, which may be used for word embedding or vocabulary lookups

Servable versions

TensorFlow Serving has the ability to handle one or multiple **versions** of a servable in the whole lifetime of a single server instance. This allows for weights, fresh algorithm configurations, and other data. This data can be loaded over a period of time. Versions allow more than one version of a servable to be loaded together, which means we can implement successive roll out. During serving time, clients can request either a latest version or a particular version ID, for a particular model.

Versions help us to maintain different versions of our models. Suppose we train a model that can classify images into 10 classes. At some point, we may want to retrain our model to classify an input into two more classes, but the accuracy of this model is slightly degraded. Some clients might want the older version of the model instead. This is where versions come in handy.

Multiple versions that are in sequence using a **servable stream** are also available. This sorts the versions in increasing order, based on the version number.

Models

With TensorFlow Serving, a **model** is represented as one or more servables. A machine learning model may include more than one algorithm (including learned weights) and also lookup or embedding tables.

A mixture model can be in either of the following forms:

- Different, independent servables
- Single, complex servable

A servable may also correspond to part of a model. An enormous lookup table, for example, could be shared across different TensorFlow Serving instances.

Sources

Sources are different plugin modules. Their work is to find and provide servables. Every Source can provide multiple servable streams. A Source is responsible for supplying one Loader instance for every version that it makes available.

TensorFlow Serving's interface for Sources is able to discover servables from arbitrary storage systems. TensorFlow Serving is made up of the usual reference Source implementations. For example, Sources may access mechanisms such as a **Remote Procedure Call (RPC)**. They might also poll a filesystem.

Basically, the serving life cycle starts whenever TensorFlow Serving recognizes a model on the disk. The Source component is responsible for taking care of it and for detecting new models that should be loaded. It always keeps an eye on the filesystem to identify whether a new model version has arrived on the disk. Upon detection of a new version, it proceeds by creating a Loader for that particular version of the model.

Loaders

The life cycle of a servable is managed by a loader. The Loader API allows a common infrastructure that is independent from the particular learning algorithms, data, or the product use cases that are involved. The APIs for loading and unloading a servable are regulated by a Loader.

The Loader contains all the required knowledge about a model. This includes information on how to load the model and estimating the required resources for the model. These resources include the RAM and GPU memory, as requested by the system. The loader also contains a pointer, which points to the model available on the disk, along with all the necessary metadata that helps to load it.

After creation of the Loader, the Source sends it to the Manager, which is known as an aspired version.

Aspired versions

Aspired versions are essentially a set of servable versions that should be loaded and ready. Sources are responsible for communicating this set of servable versions for a single servable stream at a time. A Source gives a new list of different aspired versions to the Manager, which replaces the previous list for that servable stream. The Manager unloads any previously loaded versions that are no longer present in the list.

Managers

The major responsibility of handling the entire life cycle of a Servable lies with the manager. This includes the following tasks:

- Loading servables
- Serving servables
- Unloading servables

Apart from this, the managers also listen to different Sources and track all the versions presents on a particular disk. They try to fulfil the different requests made by these sources. Under special circumstances, the managers may refuse to load a specific aspired version. These circumstances may include a required resource being unavailable. Managers also have the power to delay an unload whenever it deems fit. As an example, a Manager may wait to unload a source until a newer version finishes loading, based on a defined policy that guarantees that at least one version is loaded at all times.

To understand this better, let's take a closer look at the following scenario.

Once the manager receives an aspired version of the source, it proceeds with the serving process. At this juncture, there are two possibilities:

- The first possibility is that the first model version is already deployed. In this case, the Manager will make sure that all the required resources are available at that time. After this, the Manager gives the go-ahead to the loader which will then load the model.
- The second possibility is that we proceed with a new version of an existing model. In this case, the Manager has to make it a point to look up the Version Policy plugin before it goes ahead. This is an essential step as the Version Policy is what tells the manager how the process of loading a new model version takes place.

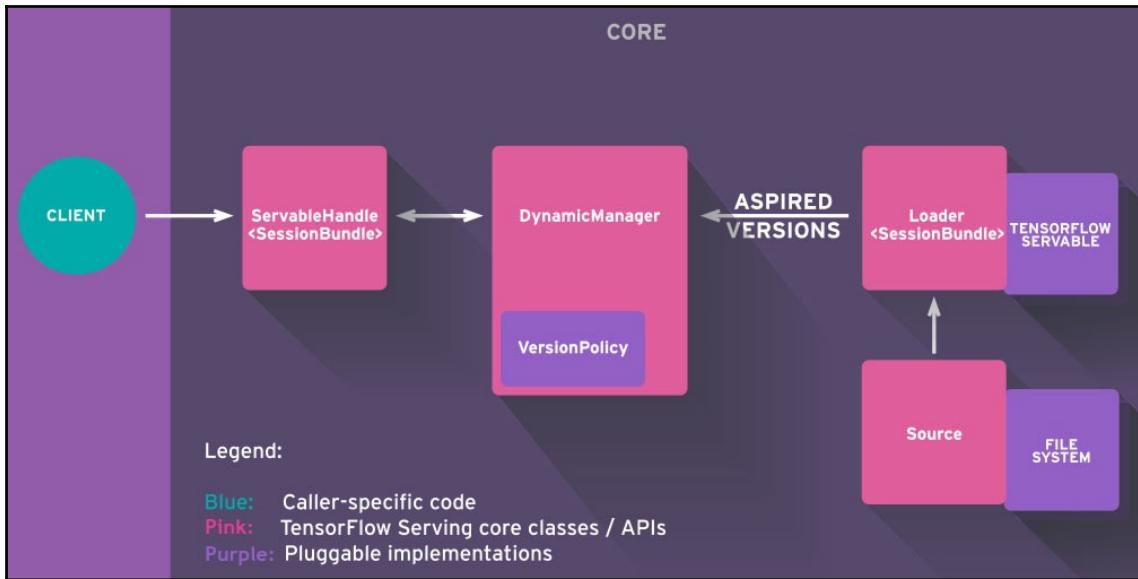
Specifically, when a Manager loads a new version of a model, it is up to us if we want to keep the availability or the resources. In the first case, we are more interested in making sure that our system is available for any incoming client requests. We already know that the Manager allows the Loader to represent the new graph with the new weights.

Now, we have two versions of the same model that are loaded at the same time. The Manager then unloads the older version, but only when the loading is finished. It is necessary to make sure that it is safe to switch between two models.

On the other hand, if we want to save different resources and compromise on having extra space for the new version, we can choose to preserve these resources. This might be quite useful if you have very heavy models in order to save some memory in exchange for getting a little latency in availability.

At the end, when a client requests a handle for the model, the Manager returns a handle to the servable.

The following diagram shows the deployment process:



Let's begin by creating a very simple TensorFlow model and try to serve it using a serving mechanism.

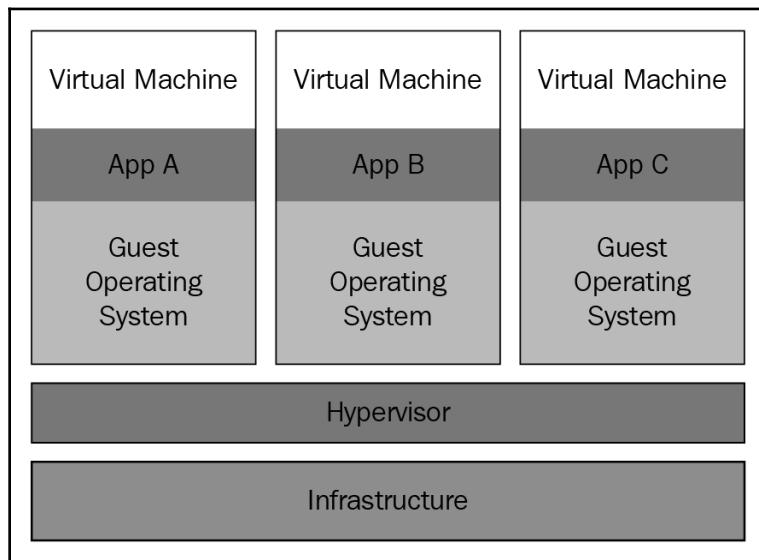
Installing and running TensorFlow Serving

To run TensorFlow Serving you will need a Docker toolbox. This allows you to run different kinds of virtual environments on a single machine using their virtual images. Docker toolbox is an open source software that can be downloaded from its official website (https://docs.docker.com/toolbox/toolbox_install_windows/).

Virtual images work as a mini operating system. They are generally .exe files that contain important dynamic libraries needed to run a virtual OS. You can load these images into Docker and then it can run independently of the actual OS on your machine. For example, you can run a Linux environment on your Windows machine using the Docker toolbox. Before going ahead with this discussion, we need to learn about two terms: virtual machines (VMs) and containers.

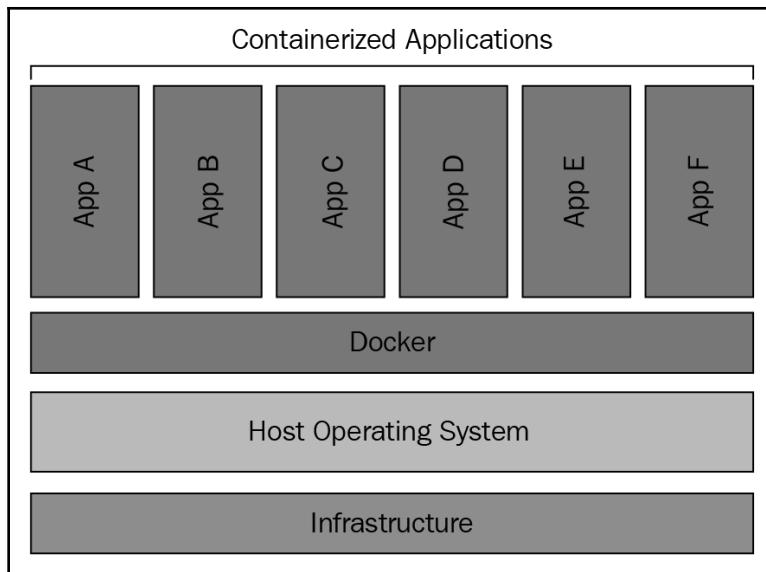
Virtual machines

VMs help to split a piece of hardware so that its power can be shared between different users. This implies that it appears as separate servers or machines. In technical terms, they are a type of abstraction of the physical hardware that helps to convert one server into multiple servers. With the use of a hypervisor, we can create multiple VMs that can run on a single machine. Each VM can have a full copy of an OS, application, and the necessary binaries and libraries that are required to run different programs. It can occupy tens of gigabytes of space on the hard disk and is generally quite slow to boot. A VM architecture can be understood by looking at the following diagram:



Containers

Containers help to virtualize the OS. They allow us to split the OS into different virtualized compartments to run container applications. Taking a more technical approach, we could say that containers are a kind of abstraction at the application layer that packages code and different dependencies together. Multiple containers can be run on the same machine; they share the OS kernels because each container runs as an independent process in the user space. An advantage of using containers in space optimization is that they take up less space than VMs. They can also handle more applications and require fewer VMs and OSes. The architecture of containers is shown in the following diagram:



Installation using Docker Toolbox

A Docker container loads a virtual image and runs it independently on your machine's OS. We will do something similar to run TensorFlow Serving. We will load a Docker image of the TensorFlow Serving application in a Docker container and run it to serve a model.

Start by installing Docker toolbox onto your system and pulling the TensorFlow Serving image from the GitHub repository. Once you have installed the Docker toolbox, run it. Type the following lines of code on the command line to pull the TensorFlow Serving image from the web:

```
docker pull TensorFlow/serving
git clone https://github.com/TensorFlow/serving
```

This will clone the repository to your local disk, and you will be ready to serve your model.

Operations for model serving

We will start by creating a basic TensorFlow model that will be responsible for a simple addition operation. Later, you can follow the same steps to serve any complicated model according to your needs. Right now, we will turn our attention toward the underlying concepts of model serving.

Model creation

The following lines of code will create the model for us:

```
# First we will import the TensorFlow library
import TensorFlow as tf
# Here we will define the flags to store our model. These flags will help
# us # to maintain the repository. It will automatically assign the
# version numbers to our models
tf.app.flags.DEFINE_integer('model_version', 1, 'version number of the
model.')
tf.app.flags.DEFINE_string('work_dir', '/tmp', 'Working directory.')
FLAGS = tf.app.flags.FLAGS
# Name the place holder
placeholder_name = 'a'
# We will do a simple scalar addition operation
operation_name = 'add'
# Let's create a placeholder to store the input number
a = tf.placeholder(tf.int32, name=placeholder_name)
# The input number will be added to a constant 10
b = tf.constant(10)
# Whenever an input comes we will add a constant to that
add = tf.add(a, b, name=operation_name)
# The next step is to create a session which can perform
# the above task.
# Run a few operations to make sure our model works
with tf.Session() as sess:
    #Let's call our model to do an addition
    c = sess.run(add, feed_dict={a: 2})
    print('10 + 2 = {}'.format(c))
    # One more time
    d = sess.run(add, feed_dict={a: 10})
    print('10 + 10 = {}'.format(d))
```

As you can see, our model simply adds a constant to an input number. For now, we will just serve this model.

Saving the model

Serving a model accurately is based on how we save our models. It needs to be in a particular format. To do this, we need to follow the serving guidelines.

In the previous example, we began by creating a placeholder that will store the input value, a constant, and an operation (addition) on the default TensorFlow graph. We then created a session to run the add operation. Here, we could wrap this code in an API endpoint for deployment purposes. It may be written in a Python framework, such as Flask or Falcon, which are web frameworks that are used for web-based application deployment, but we will avoid doing this for the following reasons:

- If our model is complex, it may run slowly on a CPU. We could run it on GPUs for speed optimization. If we use an API microservice, it might run fine on general CPUs, but it might be difficult to run on different hardware.
- If we have not optimized our heavy TensorFlow model well in the Docker image, it may unexpectedly use different system resources such as CPU, memory, or container image size.
- If your service uses different models that have been written in different TensorFlow versions, they will be difficult to manage at the same time in the Python API.

A possible solution to these problems might be to wrap a model into an API. This would give you a service for each model and then you can run different services on different hardware. This is where TensorFlow Serving ModelServer comes in handy.

It's now time to save the model. Follow these steps:

1. First, we need to consider the **input** and **output** tensors.
2. Then, we will create a **signature definition** using the input and output tensors. The model builder uses the signature definition to save a model that the server can load.
3. Now, save the model on a specified path so that server can load it from there.

Before performing these operations, we first need to figure out which nodes are input and which are output. We have a very simple math model that just performs addition in the form **a + b**, where **b** is a constant number. If we replace the constant, we could write this in the form **a + 10**. We can easily take the input tensor as **a:0** and the output tensor as **add:0**.

In the following snippet, we will get our input and output tensors and then create a signature definition that will be used to save the model:

```
# Pick out the model's input and output
a_tensor = sess.graph.get_tensor_by_name(placeholder_name + ':0')
sum_tensor = sess.graph.get_tensor_by_name(operation_name + ':0')
model_input = build_tensor_info(a_tensor)
model_output = build_tensor_info(sum_tensor)
# Create a signature definition for tf-serving
signature_definition = signature_def_utils.build_signature_def(
    inputs={placeholder_name: model_input},
    outputs={operation_name: model_output},
    method_name=signature_constants.PREDICT_METHOD_NAME)
```

We have used the names `placeholder_name` and `operation_name` for the tensor `a` and the add operation. You can use any kind of string that you like to name the input and output of your models. TensorFlow has already defined some constants for us that can be used for naming purposes. You can find these constants in the `signature_constants.py` file (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/signature_constants.py). There, you will find three kinds of constants: one for **predictions**, one for **classification**, and one for **regression**. If you take a look at `signature_constants.py`, you'll find that they have named the input and output constants as inputs and outputs only.

We will be using `a` and `add` so that you can see that there is no need to specify the TensorFlow constants here. If you are writing a model with multiple inputs and multiple outputs, you will need to name the variables yourself.

While creating a signature definition, you need to use a string defined by TensorFlow for `method_name`. This is the third parameter. It is mandatory that this must be one of either a prediction (`predict`) kind of method or a regression (`regress`) kind of method. You can find these method names in the `signature_constants.py` file as

`CLASSIFY_METHOD_NAME`, `PREDICT_METHOD_NAME`, and `REGRESS_METHOD_NAME`.

For now, let's concentrate on saving our model. We will pass the path to store the model to the `builder`. We will append the model name as the version of the model. You should use a number in increasing order each time you retrain your model.

We need to pass some constants defined by TensorFlow to the builder. Then, we need to pass the signature definition that we created in the previous snippet and save the model. We will do this in the following lines:

```
# Let's create a model builder which has a specified path
# to store our model
export_path_base = sys.argv[-1]
```

```
export_path = os.path.join(
    tf.compat.as_bytes(export_path_base),
    tf.compat.as_bytes(str(FLAGS.model_version)))
print('Exporting trained model to', export_path)
builder = tf.saved_model.builder.SavedModelBuilder(export_path)
builder = saved_model_builder.SavedModelBuilder(
    'Path to simple model')
# Build a graph containing TensorFlow session which contains all
# the variables.
builder.add_meta_graph_and_variables(
    sess, [tag_constants.SERVING],
    signature_def_map={
        signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
            signature_definition
    })
# Save the model so we can serve it with a model server
builder.save()
```

The directory that will store the model will be created by `SavedModelBuilder`, but only if it does not exist. `FLAGS.model_version` will specify the model **version**. You need to specify a larger integer value whenever you export a new version of the previous model. Each version will be stored to a different sub-directory in the given path, which will help you in version control. A metagraph and variables can be added to the builder using `add_meta_graph_and_variables()` with the following arguments:

- `sess` is the TensorFlow session. It contains the trained model that we will be exporting.
- `tags` is the set of tags that is used to save the metagraph. In this case, since we intend to use the graph in serving, we use the `serve` tag from the predefined `SavedModel` tag constants. For more details, see `tag_constants.py` (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/tag_constants.py) and the related TensorFlow API documentation (https://www.tensorflow.org/api_docs/python/tf/saved_model/tag_constants).
- `signature_def_map` specifies the map of user-supplied keys to generate a `signature` for a `TensorFlow::SignatureDef`, which can be added to the metagraph. The signature specifies which type of model is being exported and the input and output tensors to bind to when running the inference.

Once we have saved our model successfully into the directory, we are now ready to serve this model.

Serving a model

Before we start serving the model, we need to save it in the desired format. To do this, we need to run the code blocks written earlier. We will carry out the following steps in the Docker toolbox. For simplicity, I have put our `BasicModel.py` file in the same folder as where the TensorFlow serving examples have been kept by the TensorFlow serving developers. We will run the Python script as follows:

```
$ python /TensorFlow_serving/example/BasicModel.py \
TensorFlow_serving/models/basic/
```

In the preceding lines, we are simply running our Python script and storing the model in the location provided by the user. After the successful execution of this line, you will get an output that looks as follows:

```
10 + 2 = 12
10 + 10 = 20
Exporting trained model to b'TensorFlow_serving/models/basic/1
```

Let's see what we have got in the folder:

```
$ ls models/mnist
1
```

`FLAGS.model_version` has a default value of 1; therefore, the corresponding sub-directory, 1, is created:

```
$ ls models/mnist/1
saved_model.pb variables
```

Each version of the sub-directory contains the following files:

- `saved_model.pb` is the serialized `TensorFlow::SavedModel`. This may include one or more graph definitions of the model, as well as the metadata of the model, such as its signatures.
- **Variables** are the files that hold the serialized variables of the graphs.

Our TensorFlow model can now be exported to the directory and is ready to be loaded.

Once we have got our model in the desired format, we are ready to serve it. To do this, we will once again use our Docker toolbox. We will write the following command to start serving the model:

```
docker run -p 8500:8500 \
--mount type=bind,source=$(pwd)/models/basic,target=/models/basic\
-e MODEL_NAME=BasicModel -t TensorFlow/serving &
```

Here, we will use the serving image that we pulled before we started writing our code.

The `-p` option tells Docker to map its internal port 8500 out to port 8500 of the local host. If we omit this option, our model server will serve our model on port 8500, which is inside the container. This would mean that we would be unable to send any requests to the server from our computer as it would not be able to find it on the port address.

After this, Docker will mount the model from the source directory where we have kept the model and variables directory with the defined `MODEL_NAME`.

We are now serving a model using TensorFlow Serving! Unfortunately, the model server that we are running is not an HTTP server, it's a gRPC service (<https://grpc.io/>), which is TensorFlow's default. We'll discuss gRPC briefly in the following section.

What is gRPC?

Suppose you have an app that helps users to see the status of all of their friends when they click on the Show Friends button in the defined UI. Once the click has been identified, the client calls the `getFriends()` function directly, without sending any other requests to the server. As you can see, it becomes quite simple to call any function from the server, as you don't need to send and maintain any kind of request for the connections. Let's take a closer look at what gRPC does.

The official web page of gRPC says:

"In gRPC, a client application can directly call methods on a server application on a different machine as if it was a local object, making it easier for you to create distributed applications and services. As in many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types."

As our model is on the server and we want to access it remotely, we need to write a Python client that uses gRPC to call the predict method, which finally runs over the model on the server.

Let's get started with writing the client application.

Calling the model server

As you know, our model is running on a defined port. It is now time to ping it using a Python class. Open a Python file called `ProdClient`, a name that comes from production client. Start by importing the useful packages:

```
# We will start by importing the libraries
import logging
import time
# grpc will be required for communication
import grpc
from grpc import RpcError
# predict_client will help us to use the client services
# without having TensorFlow on the client system
from predict_client.pbs.prediction_service_pb2 import PredictionServiceStub
from predict_client.pbs.predict_pb2 import PredictRequest
from predict_client.util import predict_response_to_dict, make_tensor_proto
#Next, we will define the class name and the constructor that will take the
required input arguments.
# Let's write a ProdClient class
class ProdClient:
    def __init__(self, host, model_name, model_version):
        # The input to the class will be the host id,
        # the model's name, and its version
        # Load the value in self so that it will be
        # visible to each definition in the package
        self.logger = logging.getLogger(self.__class__.__name__)
        self.host = host
        self.model_name = model_name
        self.model_version = model_version
```

Now, let's add a `predict` method to our class, which will be responsible for executing the model on the server. We will create a logger that will have all the relevant information for our model's execution, such as the ID of the host port, the model's name and its version, and so on:

```
# Here will be the predict method that will load the model and
# make predictions for us whenever it gets called
def predict(self, request_data, request_timeout=10):
```

```
# SELF: is the class variable
# REQUEST_DATA: is the input to the model
# REQUEST_TIMEOUT: is the time in seconds after which the
# request will end
    # Start with logger information
    self.logger.info('Sending request to tf-serving model')
    self.logger.info('Host: {}'.format(self.host))
    self.logger.info('Model name: {}'.format(self.model_name))
    self.logger.info('Model version: {}'.format(self.model_version))
```

It's now time to create a gRPC request to our model by creating a communication channel using the host name. We will log the response in the logger. This can be done as follows:

```
# Create gRPC client and request
t = time.time()
channel = grpc.insecure_channel(self.host)
self.logger.debug('Establishing insecure channel took:
    {}'.format(time.time() - t))
# We will check the time taken for each operation
t = time.time()
stub = PredictionServiceStub(channel)
self.logger.debug('Creating stub took: {}'.format(time.time() - t))
t = time.time()
request = PredictRequest()
self.logger.debug('Creating request object took:
    {}'.format(time.time() - t))
```

Now, we will define the model's name and put the model's version in the request:

```
# Here we will define the model name
request.model_spec.name = self.model_name
# Specific version of the model
if self.model_version > 0:
    request.model_spec.version.value = self.model_version
t = time.time()
# Put inputs in the request
for d in request_data:
    tensor_proto = make_tensor_proto(d['data'],
                                    d['in_tensor_dtype'])
    request.inputs[\n        d['in_tensor_name']].CopyFrom(tensor_proto)
self.logger.debug(\n    'Making tensor protos took: {}'.format(time.time() - t))
```

We are now ready to send the request to our model:

```
# Now we are ready for prediction, the following line will help us:
try:
    t = time.time()
    predict_response = stub.Predict(request,
                                      timeout=request_timeout)
    self.logger.debug(
        'Actual request took: {} seconds'.format(time.time() - t))
    predict_response_dict = \
        predict_response_to_dict(predict_response)
    keys = [k for k in predict_response_dict]
    self.logger.info(
        'Got predict_response with keys: {}'.format(keys))
    return predict_response_dict
except RpcError as e:
    self.logger.error(e)
    self.logger.error('Prediction failed!')
return {}
```

Running the model from the client side

Let's call this class from the command line:

```
from predict_client.prod_client import ProdClient
client = ProdClient('localhost:8500', 'simple', 1)
req_data = [{'in_tensor_name': 'a', 'in_tensor_dtype': 'DT_INT32',
             'data': 2}]
client.predict(req_data)
```

This will execute our model on the server. Let's see what's actually happening when we execute the preceding lines of code:

- We will start by importing the client. `ProdClient` is the client that we use to send requests to our model server.
- We will now pass the **host**, **model name**, and **version** to the client's constructor, so that the client knows which server it needs to communicate with.
- To make a prediction using our model, we need to send some request data. As the client already knows, the server on `localhost:8500` can host any model. Here, we need to inform the client about our input tensors and their data types and then send data that is compatible to our model.

- Next, 'in_tensor_name': 'a' is a signature definition that we defined earlier. We will use the string 'DT_INT32' so that we are independent of TensorFlow, as it will require use of the tf.int32 data type.
- We will now call `client.predict` and get the following response from the server:

```
{'add': 12}
```

It has worked—this is what we want from our model. It has added the number 10 to the input data. This example has shown how easy it is to create a TensorFlow model and serve it using the TensorFlow serving framework.

Summary

In this chapter, we started by looking at the TensorFlow serving framework and its key concepts. We then moved on to creating a very basic TensorFlow model that just involved the addition of a constant to the input number. To understand these concepts, we should always start with the basics. Then, we learned about Docker toolbox and saw how it can help us to run our model on a virtual server using TensorFlow Serving. Later, we built a Python client that helped us successfully run our model on the server.

For more information, you should look at the official documentation of the framework, which will help you to build more complex models for serving purposes.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

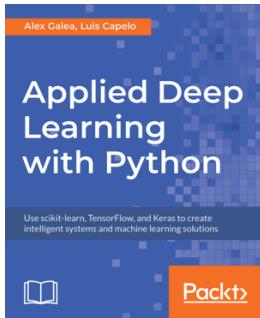


Google Cloud AI Services Quick Start Guide

Arvind Ravulavaru

ISBN: 9781788626613

- Understand Google Cloud Platform and its Cloud AI services
- Explore the Google ML Services
- Work with an Angular 5 MEAN stack application
- Integrate Vision API, Video Intelligence API for computer vision
- Be ready for conversational experiences with the Speech Recognition API, Cloud Language Process and Cloud Translation API services
- Build a smart web application that uses the power of Google Cloud AI services to make apps smarter



Applied Deep Learning with Python

Alex Galea, Luis Capelo

ISBN: 9781789804744

- Discover how you can assemble and clean your very own datasets
- Develop a tailored machine learning classification strategy
- Build, train and enhance your own models to solve unique problems
- Work with production-ready frameworks like Tensorflow and Keras
- Explain how neural networks operate in clear and simple terms
- Understand how to deploy your predictions to the web

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

A-F network
blockchains, using 264
accuracy values
precision, measuring of network 220
activation function
about 217
displaying, through preactivation process 215
affine transformations
about 530
reference 530
AGV virtual clusters
as solution 121
Amazonization project 104
Anaconda installation
DL libraries, installing 327
performing 326
Application-Specific Integrated Circuit (ASIC) 598, 599
approval, of design matrix
about 105
dimension reduction 107
format 105, 106
volume, of training dataset 108
Arithmetic and Logical Unit (ALU) 588
artificial intelligence (AI)
about 144, 145
as frontier 157
avoiding 124
based on mathematical theories 145
inventions, versus innovations 147
revolutionary, versus disruptive solutions 148
automatic guided vehicles (AGVs) 104
AWS
about 130
baseline model, preparing 130

data management 135
full sample training dataset, training 130
random sample, training of training dataset 131

B

backpropagation
vintage XOR solution, implementing in Python 71
batch normalization 567
Bayes rule
reference 547
biases 212
bidirectional generative adversarial networks (BiGAN) 578
binary Automatic Computer (BINAC) 587
bitcoin
minting 261
block
creating 264, 265
exploring 265
blockchain anticipation novelty
about 268
dataset 269
frequency table 270
likelihood table 271
naive Bayes equation 271
blockchain process
naive Bayes, using in 266
Blockchain technology
background 260
blockchains
about 256
using 262
using, in A-F network 264
buckets 135

C

central limit theorem 129
Central Processing Unit (CPU) 587, 588
chatbot 257, 278
classifier 37
clock speed 587
cloud platforms, for deployment
 about 329
 GCP, setting up 329
 prerequisites 329
cloud solutions
 AWS 130
CNN model
 applying 243
 profitable, making 244
 strategy, defining 243
cognitive chatbot service
 about 285
 case study 286
 image + word cognitive chat, activating 289
 implementation 292
 issue, solving 291
cognitive dataset 286
cognitive natural language processing 287, 288
common DL environment
 building 324, 325
 local DL environment, setting up 325
computational neuroscience 208
Compute Unified Device Architecture (CUDA) 596
computers
 about 585
 central processing unit 586
 history 585
conceptual representation learning meta-models (CRLMM)
 about 306
 blessing of dimensionality 256
 curse of dimensionality 255
 motivation 255
 profiling, with images 306
conditional probability 266
conditioning management 110
Conv2D API
 reference 420

convolution autoencoder
 about 454
 defining 456, 457
 dependencies, importing 455
 fitting 459
 loss plot 459
 low-resolution images, generating 456
 Python file 462
 scaling 456
 test results 459
convolution neural networks (ConvNets) 407
convolution
 implementing 419
 in Keras 419, 421
 model, evaluating 422, 423
 model, fitting 422
 Python file 424
 with pooling 432
convolutional neural network (CNN) 146
cost function 72, 73
covariance 319
CPU, for machine learning
 motherboard 590, 592
cross-entropy 223
cryptocurrency
 using 262
CUDA Deep Neural Network (cuDNN)
 about 597
 features 598
CycleGAN 579

D

D-Wave Systems
 reference 609
data augmentation
 about 445
 ImageDataGenerator, fitting 448, 449
 ImageDataGenerator, using 446, 447, 448
 model, compiling 449
 model, evaluating 450, 451
 model, fitting 449, 450
 Python file 452
data exploring, in image segmentation
 annotations 504
 images 502

data flow graph, into source code
about 87
backpropagation 92
cost function 90
gradient descent 90
hidden layer 88
input data layer 87
linear separability, checking 93
loss function 90
output layer 89
session, running 92

data flow graph
architecture, designing 94
displaying, in TensorFlow 96

data management, AWS
access, to output results 136
buckets 135
files, uploading 136

data preparing, image segmentation
about 506
encode 508
model data 509
normalize 507, 508

dataset optimization 104

datasets
designing 29, 104
designing, in natural language meetings 29

DCGAN
about 557
architecture 557
discriminator 558, 560, 561
generator 562

decoder networks 562

deconvolutions 563

deep learning 208

deep learning (DL) technologies
computer vision (CV) 324
natural language processing (NLP) 324

DeepMind 27

DeepSpeech2 (DS2) model
building 388
corpus exploration 389, 390, 391
data preprocessing 388
data transformation 395
defining 396, 397, 398

evaluating 405
feature engineering 391, 393
testing 405
training 400, 401, 402, 404

design matrix
approval 105
dimension reduction 107

DL environment local setup
about 325
Anaconda, downloading 326
Anaconda, installing 326

DL environment
setting up, in cloud 328

dlib
reference 519

Docker
reference 520

domain learning
about 245
programs, using 245
trained models 245
training model program 246

dropout node
features 218

dropout
about 433, 434, 435
convolution, with pooling 438
model, evaluating 436, 437
model, fitting 435, 436

DS2 architecture
implementation 397

E

eigenvalues 319
eigenvector 320
Electronic Numerical Integrator and Computer (ENIAC) 586
embedding 313
energy-based model (EBM) 301
entities 280

F

face detection
classifier, training 534, 535
code, obtaining 520

Docker image, building 520, 521, 523
Docker image, preprocessing 534
environment, setting up 520
faces, aligning 528, 529, 531
feature extraction 532
images, preprocessing 527
model, evaluation 535
pre-trained models, downloading 523, 524, 525
face landmark estimation 528
FaceNet
 reference 519
facial recognition
 about 525
 pipeline, building 525, 526
Fast Image Data Annotation Tool (FIAT) 489
feed keyword 93
feedforward neural network
 building 65
 defining 65, 66
 vintage XOR solution, implementing in Python 70
field-programmable gate arrays (FPGAs) 607
FNN architecture
 building, with TensorFlow 85
FNN XOR solution
 applying, to case study 77, 79, 82
frontier
 exploring 160, 161

G

GAN for 2D data generation
 about 555, 571
 DCGAN 557
 MNIST digit dataset 556
 model, training 573, 578
GAN Zoo
 about 578
 bidirectional generative adversarial networks
 (BiGANs) 578
 CycleGAN 579
 GraspGAN 580
 progressive growth 582
gap concept
 about 236
 jammed lanes 249

loaded 246
open lanes 251
unloaded 248
gap conceptual dataset 251
gap dataset 251
Gaussian naive Bayes
 about 272, 273
 Python program 273
GCP setup
 firewall settings, modifying 330
 project, creating 329
 VM instance, spinning 329
 VM, booting 330
generative adversarial networks (GANs)
 about 539, 541, 549, 552
 discriminator 539, 548
 for 2D data generation 555
 generator 539, 547
 implementing 541
 linear layer 546
 random data generator module 545
 real data generator module 543
 training 552, 554
generative adversarial networks
 about 252
 autoencoders 254
 conceptual representations 253
generator, DCGAN
 about 562
 batch normalization 568, 571
 transposed convolutions (TC) 563
Gibbs random sampling 302
Google Cloud Platform (GCP) 328
Google Cloud
 reference 329
Google Colab
 reference 328
Google Compute Engine (GCE) 328
Google Translate
 about 149
 customized experiment 174
 header 150
 linguist's perspective 152
 linguistic assessment 153
Google

translation service, implementing 151, 152
Google_Translate_Customized.py
KNN compressed function, implementing 166, 170, 171, 173
gradient descent 73
Graphics Processing Unit (GPU)
about 592, 595
CUDA Deep Neural Network (cuDNN) 597
NVIDIA CUDA 596
GraspGAN
used, for deep robotic grasping 580
grouping 77
gRPC
about 626
model server, calling 627
model, executing from client side 629, 630
reference 626

H

handcrafted red object detector 471
handwritten digits classification
code, implementation 408
convolution, defining 418, 419
data augmentation 445
data, exploring 408, 409, 410
deep neural network, building 411, 412, 414
deep neural network, training 411, 412, 414
deeper model, building 439
dependencies, importing 408
dropout, defining 433, 434, 435
hyperparameters, defining 410
MLP 417
model, compiling 440
model, evaluating 415, 442, 443
model, fitting 414, 441, 442
pooling, defining 425, 426, 427, 428
with pooling and Dropout 444
hard disk drive (HDD) 588, 589
hardware description language (HDL) 607
hidden neuron 154
Histogram of Oriented Gradients (HOG) 469, 525
HobChat 278
hub 268
human biomimicking 206
Hyperledger 262

hyperparameters
defining 340

I

IBM Watson
about 277
dialog flow 282
dialog tool 279
entities 280
intent 278
model, building 284
model, scripting 283
services, adding to chatbot 285

image segmentation
about 501
data, exploring 502
data, preparing 506
dependencies, importing 501
hyperparameters, defining 510
SegNet, defining 511
ImageAI
reference 479
implementation
coding 340
inference
building 351, 353
intent 278
ion traps 609

J

Jacobian matrices
using 322
jargon 154
job
creating 138, 139
running 140

K

k-means clustering solution
data 109
implementing 108
vision 109
k-means clustering
about 111, 112
applying 125

goal 114
mathematical definition 113
k-nearest neighbor (KNN) algorithm 161, 162, 163
Keras Dense API
reference 411
Keras Model API
reference 411
Keras model
loading, after training 236
loading, for optimizing training 236, 238, 239
loading, for usage 239, 240, 242
Keras
reference 408, 412, 445, 458
KNN compressed function
implementing, in
Google_Translate_Customized.py 166, 170, 171, 173
knn_polysemy.py program 164, 166

L

law of large numbers (LLN) 128
lexical field 153, 158
linear separability 63, 75, 76
linearly separable models 63
linguistic assessment, Google Translate
jargon 154
lexical field theory 153
Lloyd's algorithm 114
logistic activation functions 37
logistic classifier 38
logistic function 38
logistic sigmoid function 89
LSTM model, for generating lyrics
building 363
data pre-processing 364, 366
deep TensorFlow-based LSTM model, training
369, 371
defining 368
inference 372
output 373
LSTM model, for generating music
building 374, 375
data pre-processing 375, 377, 378
defining 380
music, generating 383, 384

training 380, 382
LSTM model, for text generation
about 357
building 358
data pre-processing 358
defining 360
inference 362
output 362
training 360, 361

M

machine
mind, building 296
Markov Decision Process (MDP) 145
mathematical explanation, PCA
covariance 319
dataset, deriving 321
eigenvalues 319
eigenvectors 320
feature vector creation 321
variance 317, 318
McCulloch-Pitts neuron
about 31, 33, 35
using 30, 31
mind
building, in machine 296
ML/DL model
selecting 104
MNIST dataset
exploring 336, 337, 338
MNIST digit dataset
reference 556
model definition 341, 344
model
creating 621
saving 621, 623
serving 625
serving, operations 620
modelSegNet model
fitting 514
Modified National Institute of Standards and Technology (MNIST)
reference 573
Monte Carlo estimator
using 129

motherboard 589
multi core processor 587
multi-layer perceptron (MLP) 335

N

naive Bayes
example 266
using, in blockchain process 266
natural language meetings
datasets, designing in 29
natural language processing (NLP) 149
Neural Machine Translation (NMT) 167
neural networks 146
NP-hard
hard 127
non-deterministic 127
polynomial 126

O

object detection in real-time, with YOLOv2
about 487
custom dataset, using 489, 490
dataset, preparing 487
dependencies, installing 490
model, evaluating 498
model, training 493, 495
pre-existing COCO dataset, using 488
YOLO model, configuring 491
YOLOv2 model, defining 492
object detection intuition
about 467
object detection models, improving 469
object detection, using OpenCV
about 470
dependencies, installing 471
handcrafted red object detector 471
image data, exploring 472, 473
image, normalizing 474, 475
mask, applying 478
mask, post-processing 476, 477
mask, preparing 475
object detection, with deep learning
about 479
dependencies, installing 479
deployment 484, 486

implementation 481, 482
implementing 479
OpenFace
reference 519, 529
operating system (OS) 588
optical cavities 609
optimal production rate (OPR) 234
optimization 349
optimizers, Keras
reference 410

P

peripheral component interconnect (PCI) 590
Pert
about 104
results, analyzing 119
results, presenting 119
Phrase-Based Machine Translation (PBMT) 166
polysemy 159
pooling
about 425, 426, 427, 428
model, evaluating 430, 431
model, fitting 429, 430
Python file 432
posterior probability 266
preactivation 212
prediction, as data augmentation
about 311
data augmentation, providing 312
input file, providing 311
RNN, running 311
principal component analysis
about 316
intuitive explanation 317
mathematical explanation 317
processor specification, for machine learning
architecture 593
clock speed 592
HDD 594
number of cores 593
Operating system (OS) 594
RAM 593
SSD 594
profit
generating, with transfer learning 233

project structure

defining 339

proof of concept (POC) approach 124

Python TensorFlow

architecture 35

Python

vintage XOR solution, implementing 70

Q

quantization 604

quantum bits (qubits) 608

quantum computers

about 608

building 608

quantum era 609

R

RAM 589

random sampling

about 127

applications 130

central limit theorem 129

LLN 128

rectified linear unit (ReLU) function 397

Recurrent Neural Network (RNN)

about 167

for data augmentation 308

LSTM 309

vanishing gradients 310

working 310

region-based fully convolutional network (R-FCN)

467

regression model

building for predictions, MLP used 335, 336

regression

defining 339

Remote Procedure Call (RPC) 615

Restricted Boltzmann Machines (RBMs)

about 298

energy-based model (EBM) 301

epochs, running 302

Gibbs random sampling 302

hidden layers 299

results, analyzing 303

visible layers 299

return on investment (ROI) 245

reward matrix 42

S

SageMaker notebook 137

scheduling 256

segmentation 562

SegNet model

compiling 514

defining 511, 512

testing 515

self-driving cars 258

sentiment analysis

about 304

datasets, parsing 304

setup process

automating 330, 331, 332

shuffling

alternative, to random sampling 133

skip-gram model 313

sklearn

hyperparameters 116

k-means clustering algorithm 116

result labels, defining 117

results, displaying 117

training dataset 115

softmax function 40, 41

software 588

solid-state drive (SSD) 467

speech recognition

building, with DeepSpeech2 (DS2) 387

stochastic gradient descent (SGD) 92

stock keeping unit (SKU) 77

Stratified Sampling 546

subnets

testing 279

system-on-a-chip (SoC) 599

systolic array 606

T

Tensor Processing Unit (TPU)

used, for optimizing speed 226, 228

Tensor Processing Units (TPUs) 598, 600, 602, 604, 605

TensorBoard Projector 321

- TensorBoard
about 209
architecture, designing of deep learning solutions 94
architecture, designing of machine learning 94
data flow graph, displaying 96
input data 209
inputs, managing of network 211
using, in corporate environment 97, 98, 100
- TensorFlow Serving
about 612
aspired versions 616
basics 613
containers 619
Docker Toolbox, used for installation 620
executing 618
installing 618
loaders 616
managers 616
models 615
reference 612, 618
servable versions 614
servables 614
sources 615
virtual machines 619
- TensorFlow
about 86, 206
code, writing with data flow graph as architectural roadmap 86
FNN architecture, building 85
reference 327
- training loop
building 344, 345, 348, 349
overfitting 349
underfitting 350
- transfer learning
- inductive abstraction 234
inductive thinking 233
motivation 233
profit, generating with 233
using 242
- translation
checking 156, 157
- transposed convolutions (TCs) 563, 567
- U**
- unit testing model 209
- Universal Automatic Computer (UNIVAC) 587
- Usty 153
- V**
- variance 317
- vector space model (VSM) 313
- vintage XOR solution
implementing, in Python 70
- W**
- weights 212
- word embedding 313
- Word2vec model 313, 315
- X**
- XOR function 62
- XOR limit
of linear model 64
of original perceptron 62
- XOR problem
solving, example 66, 67, 69
- Y**
- YOLOv2 487
- you only look once (YOLO) 467