Hands-On
Transfer Learning with Python

Implement advanced deep learning and neural network models using TensorFlow and Keras

By Dipanjan Sarkar, Raghav Bali and Tamoghna Ghosh

Hands-On Transfer Learning with Python

Implement advanced deep learning and neural network models using TensorFlow and Keras

Dipanjan Sarkar
Raghav Bali
Tamoghna Ghosh

Hands-On Transfer Learning with Python

Mapt

Foreword

Chances are you are familiar with the recent and seemingly endless machine learning innovations, but do you know about what goes into training a machine learning model? Generally, a given machine learning model is trained on specific data for a particular task. This training process can be exceptionally resource and time-consuming, and since the resulting models are task-specific, the maximum potential of the resulting model is not realized.

Optimally-performing neural network models, for example, are often the result of many iterations of fine-tuning from researchers or practitioners. Could these trained models not be additionally exploited for a wider assortment of tasks? Transfer learning involves the leveraging of existing machine learning models for use in scenarios in which the models were not originally trained.

Much as humans do not discard everything they have previously learned and start a fresh each time they take up a new task, transfer learning allows a machine learning model to port the knowledge it has acquired during training to new tasks, extending the reach of the combination of computation and expertise having been used as fuel for the original model. Simply put, transfer learning can save training time and extend the usefulness of existing machine learning models. It is also an invaluable technique for tasks where the large amounts of training data typically required for training a model from scratch are not available.

Becoming familiar with complex concepts and implementing these concepts in practice are two very different things, and this is where *Hands-On Transfer Learning with Python* shines. The book starts with a deep dive into both deep learning and transfer learning, conceptually. This is followed by practical implementations of these concepts with real-world examples and research problems, using modern deep learning tools from the Python ecosystem, such as TensorFlow and Keras. Dipanjan, Raghav, and Tamoghna excel at elegantly marrying the theoretical and the practical, a remarkable advantage for the reader of such a well-crafted publication.

Transfer learning has shown much promise of late in many domains, and is a very active area of contemporary machine learning research. If you are looking for a complete guide to both deep learning and transfer learning, starting from zero, *Hands-On Transfer Learning with Python* should be your first stop.

Matthew Mayo

Editor, KDnuggets

@mattmayo13

Contributors

About the authors

Dipanjan (**DJ**) **Sarkar** is a Data Scientist at Intel, leveraging data science, machine learning, and deep learning to build large-scale intelligent systems. He holds a master of technology degree with specializations in Data Science and Software Engineering.

He has been an analytics practitioner for several years now, specializing in machine learning, NLP, statistical methods, and deep learning. He is passionate about education and also acts as a Data Science Mentor at various organizations like Springboard, helping people learn data science. He is also a key contributor and editor for *Towards Data Science*, a leading online journal on AI and Data Science. He has also authored several books on R, Python, machine learning, NLP, and deep learning.

Raghav Bali is a Data Scientist at Optum (United Health Group). His work involves research and development of enterprise-level solutions based on machine learning, deep learning, and NLP for Healthcare and Insurance related use cases. In his previous role at Intel, he was involved in enabling proactive data driven IT initiatives. He has also worked in ERP and finance domains with some of the leading organizations in the world. Raghav has also authored multiple books with leading publishers.

Raghav has a master's degree (gold medalist) in Information Technology from International Institute of Information Technology, Bangalore. He loves reading and is a shutterbug capturing moments when he isn't busy solving problems.

Tamoghna Ghosh is a machine learning engineer at Intel Corporation. He has overall 11 years of work experience including 4 years of core research experience at Microsoft Research (MSR) India. At MSR he worked as a research assistant in cryptanalysis of block ciphers.

His technical expertise's are in big data, machine learning, NLP, information retrieval, data visualization and software development. He received M.Tech (Computer Science) degree from the Indian Statistical Institute, Kolkata and M.Sc. (Mathematics) from University of Calcutta with specialization in functional analysis and mathematical modeling/dynamical systems. He is passionate about teaching and conducts internal training in data science for Intel at various levels.

About the reviewer

Nitin Panwar has a master's degree in Computer Science from the Indian Institute of Information technology, Gwalior. He is a Technical Lead (data science) at Naukri, India's No.1 job site, where he works on data science, machine learning, and text analytics. He has also worked as a data scientist at Intel, the world's largest silicon company. Nitin's interests include learning about new technology, AI-powered start-ups, and data science.

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

With the world moving towards digitization and automation, as a technologist/programmer it is important to keep oneself updated and learn how to leverage these tools and techniques. This book, *Hands-On Transfer Learning with Python*, is an attempt to help practitioners get acquainted with and equipped to use these advancements in their respective domains. This book is structured broadly into three sections:

- Deep learning foundations
- Essentials of transfer learning
- Transfer learning case studies

Transfer learning is a **machine learning** (**ML**) technique where knowledge gained during the training of one set of ML problems can be used to train other similar types of problems.

The purpose of this book is two-fold. We will focus on detailed coverage of deep learning and transfer learning, comparing and contrasting the two with easy-to-follow concepts and examples. The second area of focus will be on real-world examples and research problems using TensorFlow, Keras, and the Python ecosystem with hands-on examples.

The book starts with core essential concepts of ML and deep learning, followed by some depictions and coverage of important deep learning architectures, such as CNNs, DNNs, RNNs, LSTMs, and capsule networks. Our focus then shifts to transfer learning concepts and pretrained *state of the art networks* such as VGG, Inception, and ResNet. We also learn how these systems can be leveraged to improve performance of our deep learning models. Finally, we focus on a multitude of real-world case studies and problems in areas such as **computer vision**, **audio analysis**, and **natural language processing** (**NLP**).

By the end of this book, you will be all ready to implement both deep learning and transfer learning principles in your own systems.

Who this book is for

Hands-On Transfer Learning with Python is for data scientists, ML engineers, analysts, and developers with an interest in data and applying state-of-the-art transfer learning methodologies to solve tough real-world problems.

Basic proficiency in ML and Python is required.

What this book covers

`Chapter 1`, *Machine Learning Fundamentals*, introduces the CRISP-DM model, which presents an industry standard framework/workflow for any data science, ML, or deep learning project. We will also touch upon various important concepts covering the fundamentals in the ML landscape such as exploratory data analysis, feature extraction and engineering, evaluation metrics, and so on.

`Chapter 2`, *Deep Learning Essentials*, provides a whirlwind tour of deep learning essentials, providing an overview of the basic building blocks of neural networks and also how deep neural networks are trained. Starting from the basics of how a single neural unit works, important concepts like activation functions, loss functions, optimizers and neural network hyperparameters are covered. Special focus is also emphasized on setting up on-premise and cloud-based deep learning environments.

`Chapter 3`, *Understanding Deep Learning Architectures*, focuses on understanding the various standard model architectures present today in deep learning. We have come a long way since traditional ANNs in the 1960s, and essential model architectures such as fully connected **deep neural networks** (**DNNs**), **Convolutional Neural Networks** (**CNNs**), **recurrent neural networks** (**RNNs**), **Long-Short Term Memory** (**LSTM**) networks, and the most recent **Capsule Networks** will be covered, to name a few.

`Chapter 4`, *Transfer Learning Fundamentals*, looks at the core concepts, terminology, and model architectures associated with the concept of transfer learning. Concepts and architectures pertaining to pretrained models will be discussed in detail. We will also compare and contrast transfer learning with deep learning and talk about types and strategies of transfer learning.

`Chapter 5`, *Unleashing the Power of Transfer Learning*, takes an actual example with a dataset from Kaggle, leverages deep learning models on it, and gives readers an understanding of the challenges faced when we have a small number of data points, and how transfer learning can unleash its true power and potential to give us superior models in these scenarios. We will tackle the very popular dogs and cats classification task here with the twist of a less data availability constraint.

`Chapter 6`, *Image Recognition and Classification*, is the first in a series of real-world applications/case studies of concepts discussed in detail in the previous two parts of the book. The chapter begins with an introduction to the task of image classification, and goes on to discuss and implement some of the popular, state-of-the-art deep learning models on diverse image classification problems.

`Chapter 7`, *Text Document Categorization*, discusses the application of transfer learning to a very popular natural language processing problem, text document categorization. The chapter begins with a high-level introduction to the multi-class text classification problem, traditional models, benchmark text classification datasets such as 20-newsgroups and performance. Later, it introduces deep learning document models for text classification and their advantages over traditional models. We learn about word feature representation using dense vectors and how to leverage the same for applying transfer learning in our text categorization problem such that our source and target domains might be different. Other unsupervised tasks like document summarization are also depicted.

`Chapter 8`, *Audio Identification and Classification*, solves the tough problem of identifying and classifying very short audio clips. Here we leverage transfer learning using some innovation of applying the power of pre-trained deep learning models from the computer vision domain into a totally different domain of audio identification.

`Chapter 9`, *DeepDream*, focuses on a gentle introduction to the domain of generative deep learning, which is one of the core ideas at the forefront of true artificial intelligence. We will be focusing on how convnets (CNNs) think or dream and visualize patterns in images by leveraging transfer learning. First released by Google in 2015, it became a viral sensation due to the interesting patterns deep networks started to generate from images, as if thinking and dreaming on their own!

`Chapter 10`, *Style Transfer*, leverages concepts from deep learning, transfer learning, and generative learning to showcase artistic image neural style transfer with hands-on examples on different content images and styles.

`Chapter 11`, *Automated Image Caption Generator*, covers one of the most complex problems in computer vision, as well as natural language generation—image captioning. While classifying images into fixed categories is challenging yet not impossible, this is a slightly more complex task involving generating human-like natural language textual captions for any photo or scene. Leveraging the power of transfer learning, natural language processing and generative models, you will learn how to build your own automated image captioning system from scratch.

`Chapter 12`, *Image Colorization*, presents a unique case-study where the task is to colorize black and white or grayscale images. This chapter introduces readers to the basics of various color scales and why image colorization is such a difficult task.

To get the most out of this book

1. It would be great if you have basic proficiency in ML and Python.
2. An avid interest in data analysis, ML, and deep learning would be beneficial.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.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.packtpub.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
- Zipeg/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/Hands-On-Transfer-Learning-with-Python`. If 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!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/HandsOnTransferLearningwithPython_ColorImages.pdf`.

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: "Once complete, remember to add the following lines to the end of `~/.bashrc` using your favorite editor (we used `vim`)."

A block of code is set as follows:

```
import glob
import numpy as np
import os
import shutil
from utils import log_progress

np.random.seed(42)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
preprocessor = Preprocess()
corpus_to_seq = preprocessor.fit(corpus=corpus)

holdout_corpus = test_df['review'].values
holdout_target = test_df['sentiment'].values
holdout_corpus_to_seq = preprocessor.transform(holdout_corpus)
```

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

```
ubuntu@ip:~$ mkdir ssl
ubuntu@ip:~$ cd ssl
ubuntu@ip:~/ssl$
```

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: "Once you launch your instance, you can check out the **Instances** section and try to connect to the instance."

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

1
Machine Learning Fundamentals

One day the AIs are going to look back on us the same way we look at fossil skeletons on the plains of Africa. An upright ape living in dust with crude language and tools, all set for extinction.

– Nathan Bateman, Ex Machina (Movie 2014)

This quote may seem exaggerated to the core and difficult to digest, yet, with the pace at which technology and science are improving, who knows? We as a species have always dreamt of creating intelligent, self-aware machines. With recent advancements in research, technology, and the democratization of computing power, **artificial intelligence** (**AI**), **machine learning** (**ML**), and deep learning have gotten enormous attention and hype amongst technologists and the population in general. Though Hollywood's promised future is debatable, we have started to see and use glimpses of intelligent systems in our daily lives. From intelligent conversational engines, such as Google Now, Siri, Alexa, and Cortana, to self-driving cars, we are gradually accepting such smart technologies in our daily routines.

As we step into the new era of learning machines, it is important to understand that the fundamental ideas and concepts have existed for some time and have constantly been improved upon by intelligent people across the planet. It is well known that 90% of the world's data has been created in just the last couple of years, and we continue to create far more data at ever increasing rates. The realm of ML, deep learning, and AI helps us utilize these massive amounts of data to solve various real-world problems.

This book is divided into three sections. In this first section, we will get started with the basic concepts and terminologies associated with AI, ML, and deep learning, followed by in-depth details on deep learning architectures.

This chapter provides our readers with a quick primer on the basic concepts of ML before we get started with deep learning in subsequent chapters. This chapter covers the following aspects:

- Introduction to ML
- ML methodologies
- CRISP-DM—workflow for ML projects
- ML pipelines
- Exploratory data analysis
- Feature extraction and engineering
- Feature selection

Every chapter of the book builds upon concepts and techniques from the previous chapters. Readers who are well-versed with the basics of ML and deep learning may pick and choose the topics as they deem necessary, yet it is advised to go through the chapters sequentially. The code for this chapter is available for quick reference in the `Chapter 1` folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

Why ML?

We live in a world where our daily routine involves multiple contact points with the digital world. We have computers assisting us with communication, travel, entertainment, and whatnot. The digital online products (apps, websites, software, and so on) that we use seamlessly all the time help us avoid mundane and repetitive tasks. These software have been developed using computer programming languages (like C, C++, Python, Java, and so on) by programmers who have explicitly programmed each instruction to enable these software to perform defined tasks. A typical interaction between a compute device (computer, phone, and so on) and an explicitly programmed software application with inputs and defined outputs is depicted in the following diagram:
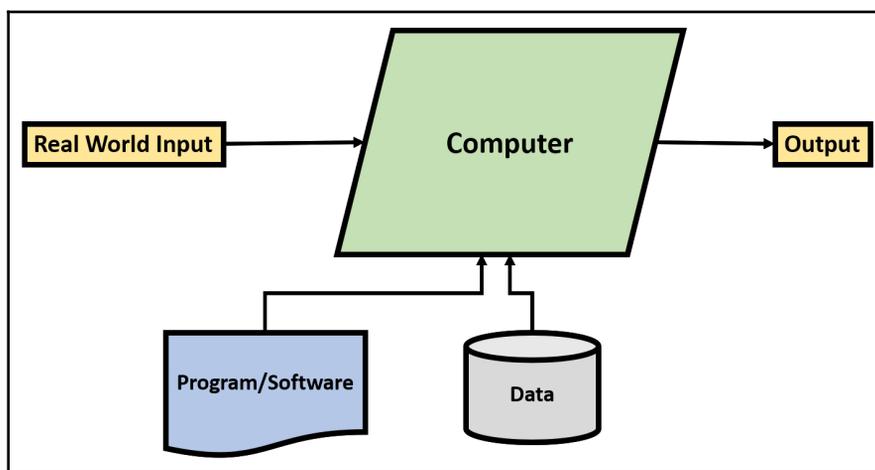
Tradition programming paradigm

Though the current paradigm has been helping us develop amazingly complex software/systems to address tasks from different domains and aspects in a pretty efficient way, they require somebody to define and code explicit rules for such programs to work. These are the tasks that are easy for a computer to solve but difficult or time consuming for humans. For instance, performing complex calculations, storing massive amounts of data, searching through huge databases, and so on are tasks that can be performed efficiently by a computer once the rules are defined.

Yet, there is another class of problems that can be solved intuitively by humans but are difficult to program. Problems like object identification, playing games, and so on are natural to us yet difficult to define with a set of rules. Alan Turing, in his landmark paper *Computing Machinery and Intelligence* (`https://www.csee.umbc.edu/courses/471/papers/turing.pdf`), which introduced the **Turing test**, discussed general purpose computers and whether they could be capable of such tasks.

This new paradigm, which embodies the thoughts about general purpose computing, is what gave rise to AI in a broader sense. This new paradigm, better termed as an ML paradigm, is one where computers or machines learn from experience (analogous to human learning) to solve tasks rather than being explicitly programmed to do so.

AI is thus an encompassing field of research, with ML and deep learning being specific subfields of study within it. AI is a general field that includes other subfields as well, which may or may not involve learning (for instance, see symbolic AI). In this book we will concentrate our time and efforts upon ML and deep learning only. The scope of artificial intelligence, machine learning, and deep learning can be visualized as follows:
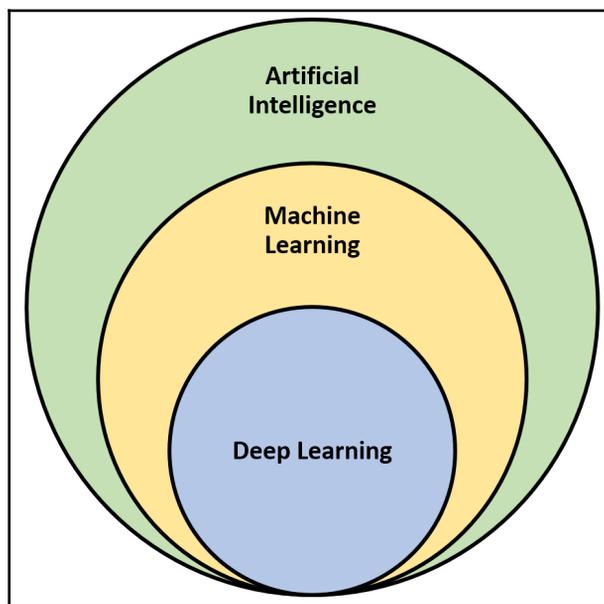
Scope of artificial learning, with machine learning, and deep learning as its subfields

Formal definition

A formal definition of ML, as stated by Tom Mitchell, is explained as follows.

A computer program is said to learn from experience *E* with respect to some task *T* and some performance measure *P*, if its performance on *T*, as measured by *P*, improves with experience *E*.

This definition beautifully captures the essence of what ML is in a very concise manner. Let's take an example from the real world to understand it better. Let's consider a task (*T*) is to identify spam emails. We may now present many examples (or experiences *E*) to a system about spam and non-spam emails, from which it learns rather than being explicitly programmed. The program or system may then be measured for its performance (*P*) on the learned task of identifying spam emails. Interesting, isn't it?

Shallow and deep learning

ML is thus the task of identifying patterns from training examples and applying these learned patterns (or representations) to new unseen data. ML is also sometimes termed as **shallow learning** because of its nature of learning *single layered representations* (in most cases). This brings us to the questions of *what layers of representation are?* and *what deep learning is?* We will answer these questions in the subsequent chapters. Let's have a quick overview of deep learning.

Deep learning is a subfield of ML that is concerned with learning successive meaningful representations from training examples to solve a given task. Deep learning is closely associated with artificial neural networks that consist of multiple layers stacked one after the other, which capture successive representations.

Do not worry if it was difficult to digest and understand, as mentioned, we will cover more in considerable depth in subsequent chapters.

ML has become a buzzword thanks to the amount of data we are generating and collecting along with faster compute. Let's look at ML in more depth in the following sections.

ML techniques

ML is a popular subfield of AI, one which covers a very wide scope. One of the reasons for this popularity is the comprehensive toolbox of sophisticated algorithms, techniques, and methodologies under its gambit. This toolbox has been developed and improved over the years, and new ones are being researched on an ongoing basis. To understand and use the ML toolbox wisely, consider the following few ways of categorizing it.

Categorization based on amount of human supervision:

- **Supervised learning**: This class of learning involves high-human supervision. The algorithms under supervised learning utilize the training data and associated outputs to learn a mapping between the two and apply the same on unseen data. Classification and regression are two major types of supervised learning algorithms.
- **Unsupervised learning**: This class of algorithms attempts to learn inherent latent structures, patterns, and relationships from the input data without any associated outputs/labels (human supervision). Clustering, dimensionality reduction, association rule mining, and so on are a few major types of unsupervised learning algorithms.
- **Semi-supervised learning**: This class of algorithms is a hybrid of supervised and unsupervised learning. In this case, the algorithms work with small amounts of labeled training data and more of unlabeled data. Thus making a creative use of both supervised and unsupervised methods to solve a given task.
- **Reinforcement learning**: This class of algorithms is a bit different from supervised and unsupervised learning methods. The central entity here is an agent, which trains over a period while interacting with its environment to maximize some reward/award. The agent iteratively learns and changes strategies/policies based on rewards/penalties from interacting with the environment.

Categorization based on data availability:

- **Batch learning**: This is also termed as **offline learning**. This type of learning is utilized when the required training data is available, and a model can be trained and fine-tuned before deploying into production/real world.
- **Online learning**: As the name suggests, in this case the learning is not stopped once the data is available. Rather, in this case, data is fed into the system in mini-batches and the training process continues with new batches of data.

The previously discussed categorizations give us an abstract view of how ML algorithms can be organized, understood, and utilized. The most common way to categorize them is into supervised and unsupervised learning algorithms. Let's go into a bit more detail about these two categories as this should help us get started for further advanced topics to be introduced later.

Supervised learning

Supervised learning algorithms are a class of algorithms that utilize data samples (also called **training samples**) and corresponding outputs (or **labels**) to infer a mapping function between the two. The inferred mapping function or the learned function is the output of this training process. The learned function is then utilized to correctly map new and unseen data points (input elements) to test the performance of the learned function.

Some key concepts for supervised learning algorithms are as follows:

- **Training dataset**: The training samples and corresponding outputs utilized during the training process are termed as **training data**. Formally, a training dataset is a two-element tuple consisting of an input element (usually a vector) and a corresponding output element or signal.
- **Test dataset**: The unseen dataset that is utilized to test the performance of the learned function. This dataset is also a two-element tuple containing input data points and corresponding output signals. Data points in this set are not used for the training phase (this dataset is further divided into the validation set as well; we will discuss this in more detail in subsequent chapters).
- **Learned function**: This is the output of the training phase. Also termed as *inferred function* or the *model*. This function is inferred based on the training examples (input data points and their corresponding outputs) from the training dataset. An ideal model/learned function would learn the mapping in such a way that the results can be generalized for unseen data as well.

There are various supervised learning algorithms available. Based on the use case requirements, they can be majorly categorize into classification and regression models.

Classification

In the simplest terms, these algorithms help us answer *objective* questions or a *yes-no* prediction. For instance, these algorithms are useful in scenarios like *is it going to rain today?*, or *can this tumour be cancerous?*, and so on.

Formally, the key objective of classification algorithms is to predict output labels that are categorical in nature depending upon the input data points. The output labels are categorical in nature; namely, they each belong to a discrete class or category.

Logistic regression, **Support Vector Machines** (**SVMs**), **Neural Networks**, **Random Forests**, **k-Nearest Neighbours** (**KNN**), **Decision Trees**, and so on are some of the popular classification algorithms.

Suppose we have a real-world use case to evaluate different car models. To keep things simple, let's assume that the model is expected to predict an output for every car model as either acceptable or unacceptable based on multiple input training samples. The input training samples have attributes such as buying price, number of doors, capacity (in number of persons), and safety.

The level apart from the class label denotes each data point as either acceptable or unacceptable. The following diagram depicts the binary classification problem at hand. The classification algorithm takes the training samples as input to prepare a supervised model. This model is then utilized to predict the evaluation label for a new data point:
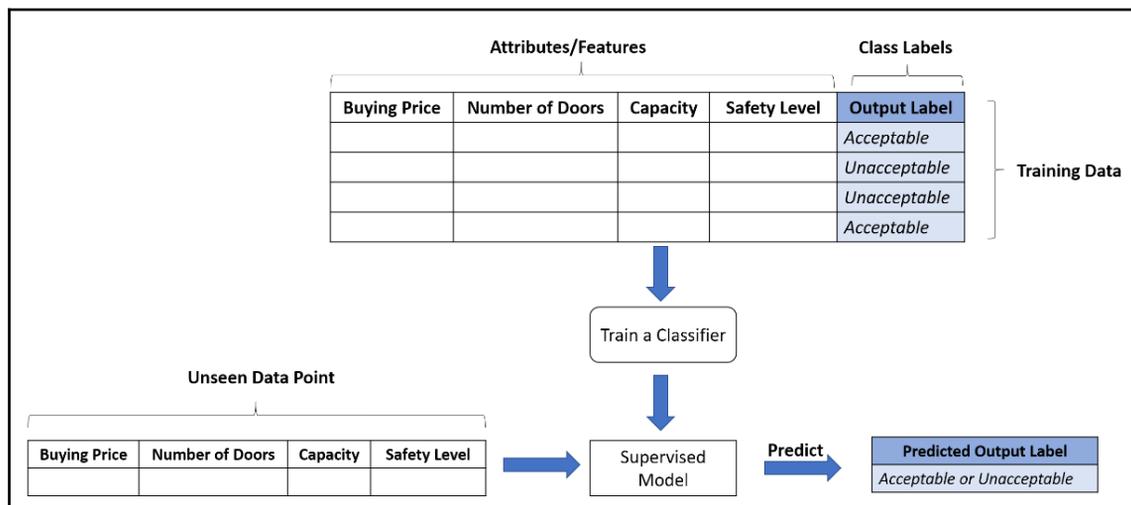
Supervised learning: Binary classification for car model evaluation

Since output labels are discrete classes in case of classification problems, if there are only two possible output classes the task is termed as a **binary classification problem**, and a multi-class classification otherwise. Predicting whether it will rain tomorrow or not would be a binary classification problem (with output being a yes or a no) while predicting a numeric digit from scanned handwritten images would be multi-class classification with 10 labels (zero to nine possible output labels).

Regression

This class of supervised learning algorithms helps us answer *quantitative* questions of the type *how many* or *how much?*. Formally, the key objective for regression models is value estimation. In this case, the output labels are continuous in nature (as opposed to being discrete in classification).

In the case of regression problems, the input data points are termed as *independent* or *explanatory variables*, while the output is termed as a *dependent variable*. Regression models are also trained using training data samples consisting of input (or independent) data points along with output (or dependent) signals. Linear regression, multivariate regression, regression trees, and so on are a few supervised regression algorithms.

> Regression models can be further categorized based on how they model the relationship between dependent and independent variables.

Simple linear regression models work with single independent and single dependent variables. **Ordinary Least Squares** (**OLS**) regression is a popular linear regression model. Multiple regression or multivariate regression is where there is a single dependent variable, while each observation is a vector composed of multiple explanatory variables.

Polynomial regression models are a special case of multivariate regression. Here the dependent variable is modeled to the nth degree of the independent variable. Since polynomial regression models fit or map nonlinear relationships between dependent and independent variables, these are also termed as **nonlinear regression** models.

The following is an example of linear regression:
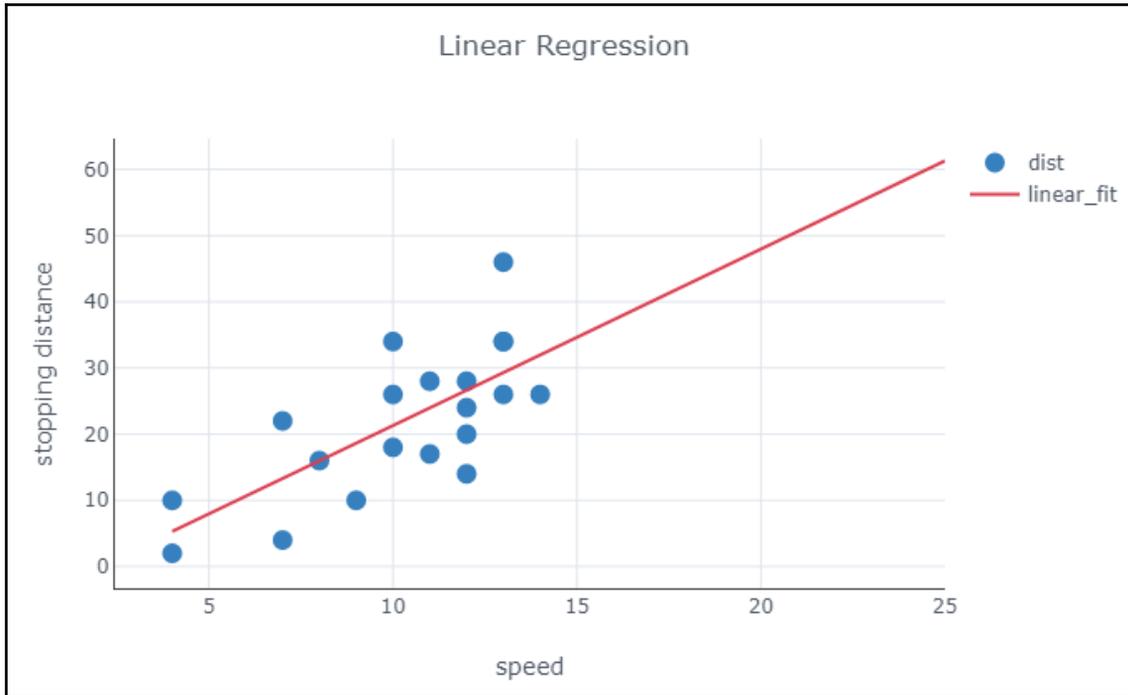
Supervised learning: Linear regression

To understand different regression types, let's consider a real-world use case of estimating the stopping distance of a car, based on its speed. Here, based on the training data we have, we can model the stopping distance as a linear function of speed or as a polynomial function of the speed of the car. Remember that the main objective is to minimize the error without overfitting the training data itself.

The preceding graph depicts a linear fit while the following one depicts a polynomial fit for the the same dataset:
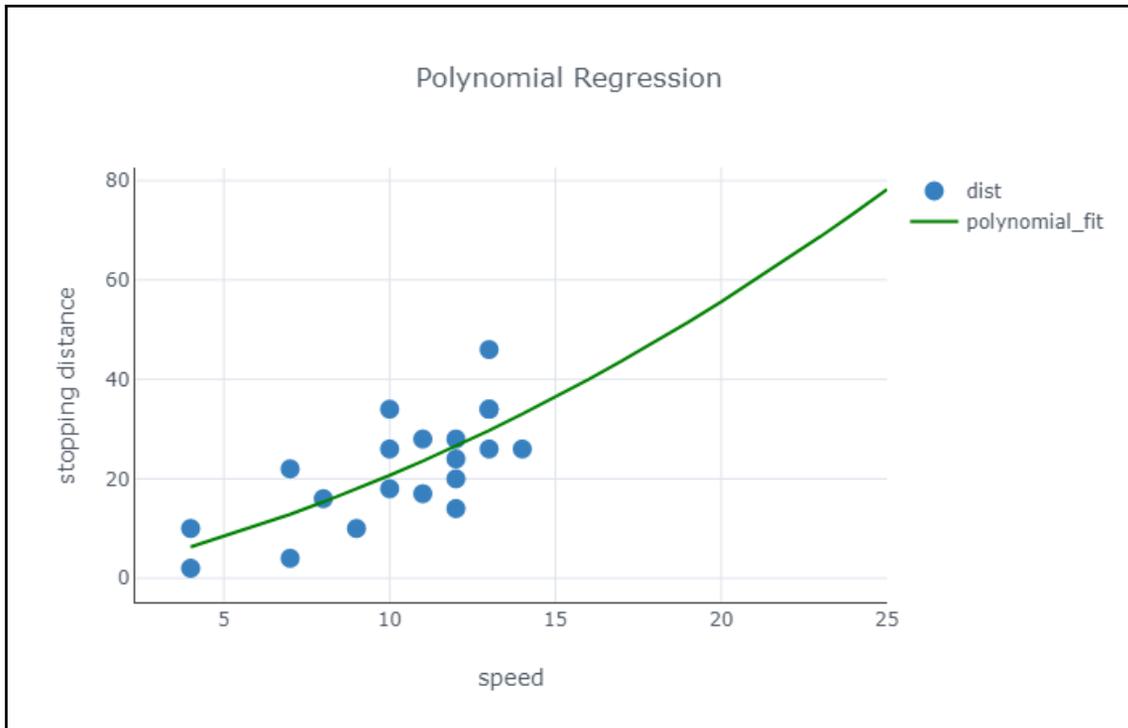
Supervised learning: Polynomial regression

Unsupervised learning

As the name suggests, this class of algorithms learns/infers concepts without supervision. Unlike supervised learning algorithms, which infer a mapping function based on training dataset consisting of input data points and output signals, unsupervised algorithms are tasked with finding patterns and relationships in the training data without any output signals available in the training dataset. This class of algorithms utilizes the input dataset to detect patterns, and mine for rules or group/cluster data points so as to extract meaningful insights from the raw input dataset.

Unsupervised algorithms come in handy when we do not have the liberty of a training set that contains corresponding output signals or labels. In many real-world scenarios, datasets are available without output signals and it is difficult to manually label them. Thus, unsupervised algorithms are helpful in plugging such gaps.

Similar to supervised learning algorithms, unsupervised algorithms can also be categorized for ease of understanding and learning. The following are different categories of unsupervised learning algorithms.

Clustering

The unsupervised equivalent of classification is termed as **clustering**. These algorithms help us cluster or group data points into different groups or categories, without the availability of any output label in the input/training dataset. These algorithms try to find patterns and relationships from the input dataset, utilizing inherent features to group them into various groups based on some similarity measure, as shown in the following diagram:
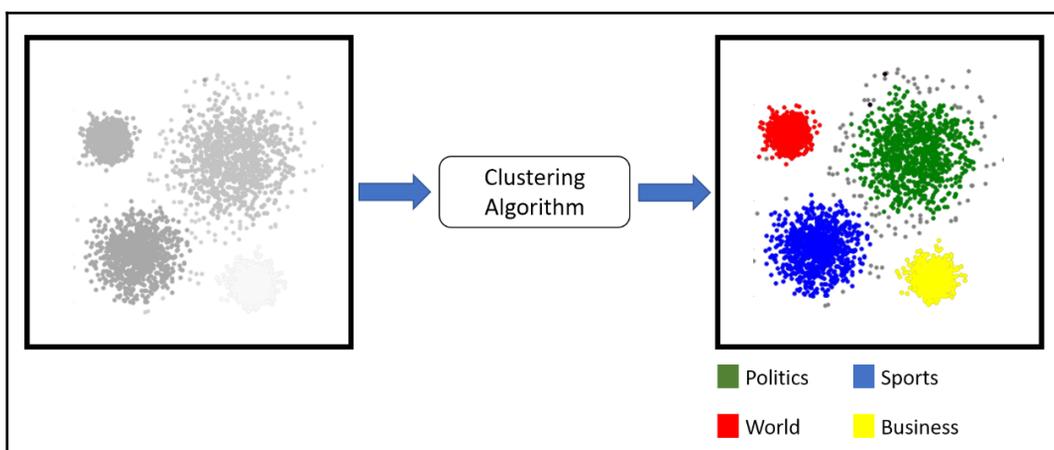
Unsupervised learning: Clustering news articles

A real-world example to help understand clustering could be news articles. There are hundreds of news articles written daily, each catering to different topics ranging from politics and sports to entertainment, and so on. An unsupervised approach to group these articles together can be achieved using clustering, as shown in the preceding figure.

There are different approaches to perform the process of clustering. The most popular ones are:

- Centroid based methods. Popular ones are K-means and K-medoids.
- Agglomerative and divisive hierarchical clustering methods. Popular ones are Ward's and affinity propagation.
- Data distribution based methods, for instance, Gaussian mixture models.
- Density based methods such as DBSCAN and so on.

Dimensionality reduction

Data and ML are the best of friends, yet a lot of issues come with more and bigger data. A large number of attributes or a bloated-up feature space is one common problem. A large feature space poses problems in analyzing and visualizing the data along with issues related to training, memory, and space constraints. This is also known as **the curse of dimensionality**. Since unsupervised methods help us extract insights and patterns from unlabeled training datasets, they are also useful in helping us reduce dimensionality.

In other words, unsupervised methods help us reduce feature space by helping us select a representative set of features from the complete available list:
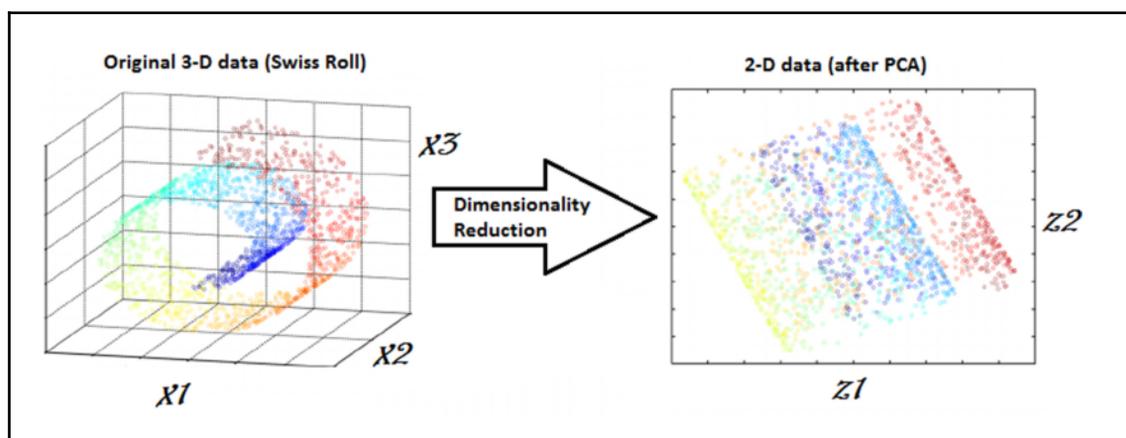
Unsupervised learning: Dimensionality reduction using PCA

Principal Component Analysis (**PCA**), nearest neighbors, and discriminant analysis are some of the popular dimensionality reduction techniques.

The preceding diagram is a famous depiction of the workings of the PCA based dimensionality reduction technique. It shows a swiss roll shape with data represented in three-dimensional space. Application of PCA results in transformation of the data into two-dimensional space, as shown on the right-hand side of the diagram.

Association rule mining

This class of unsupervised ML algorithms helps us understand and extract patterns from transactional datasets. Also termed as **Market Basket Analysis** (**MBA**), these algorithms help us identify interesting relationships and associations between items across transactions.

Using association rule mining, we can answer questions like *what items are bought together by people at a given store?*, or *do people who buy wine also tend to buy cheese?*, and many more. FP-growth, ECLAT, and Apriori are some of the most widely used algorithms for association rule mining tasks.

Anomaly detection

Anomaly detection is the task of identifying rare events/observations based on historical data. Anomaly detection is also termed as **outlier detection**. Anomalies or outliers usually have characteristics such as being infrequent or occurring in short sudden bursts over time.

For such tasks, we provide a historical dataset for the algorithm so it can identify and learn the normal behavior of data in an unsupervised manner. Once learned, the algorithm helps us identify patterns that differ from this learned behavior.

CRISP-DM

Cross Industry Standard Process for Data Mining (**CRISP-DM**) is one of the most popular and widely used processes for data mining and analytics projects. CRISP-DM provides the required framework, which clearly outlines the necessary steps and workflows for executing a data mining and analytics project, from business requirements to the final deployment stages and everything in between.

More popularly known by the acronym itself, CRISP-DM is a tried, tested, and robust industry standard process model followed for data mining and analytics projects. CRISP-DM clearly depicts the necessary steps, processes, and workflows for executing any project, right from formalizing business requirements to testing and deploying a solution to transform data into insights. Data science, data mining, and ML are all about trying to run multiple iterative processes to extract insights and information from data. Hence, we can say that analyzing data is truly both an art as well as a science, because it is not always about running algorithms without reason; a lot of the major effort involves understanding the business, the actual value of the efforts being invested, and proper methods for articulating end results and insights.

Data science and data mining projects are iterative in nature to extract meaningful insights and information from data. Data science is as much art as science and thus a lot of time is spent understanding the business value and the data at hand before applying the actual algorithms (these again go through multiple iterations) and finally evaluations and deployment.

Similar to software engineering projects, which have different life cycle models, CRISP-DM helps us track a data mining and analytics project from start to end. This model is divided into six major steps that cover from aspects of business and data understanding to evaluation and finally deployment, all of which are iterative in nature. See the following diagram:
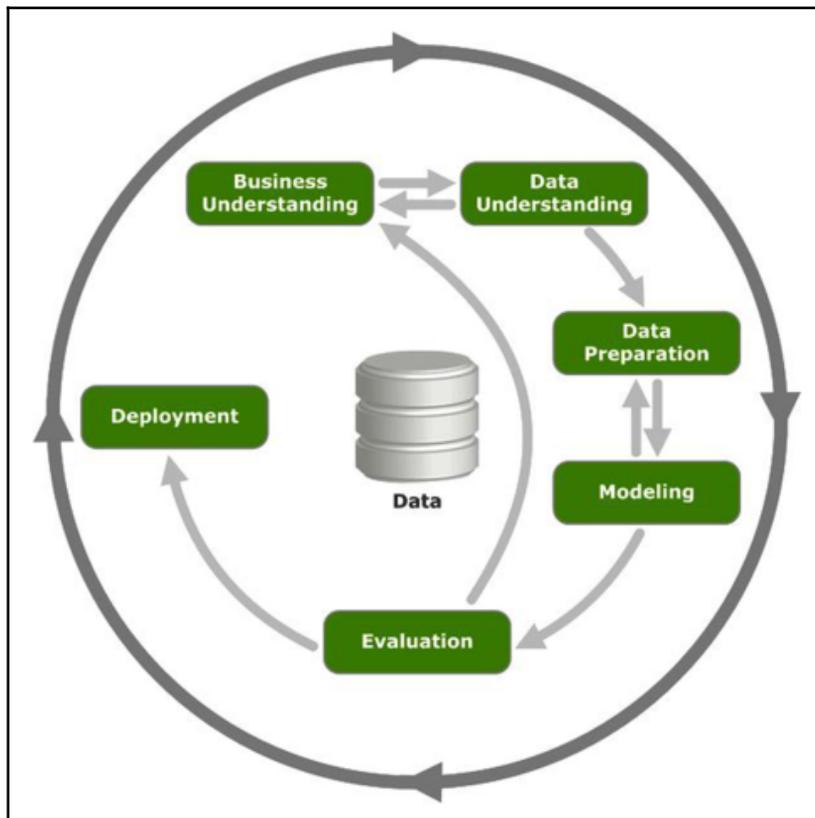
CRISP-DM model depicting workflow for ML projects

Let's now have a deeper look into each of the six stages to better understand the CRISP-DM model.

Business understanding

The first and the foremost step is understanding the business. This crucial step begins with setting the business context and requirements for the problem. Defining the business requirements formally is important to transform them into a data science and analytics problem statement. This step also used to set the expectations and success criteria for both business and data science teams to be on the same page and track the progress of the project.

The main deliverable of this step is a detailed plan consisting of major milestones, timelines, assumptions, constraints, caveats, issues expected, and success criteria.

Data understanding

Data collection and understanding is the second step in the CRISP-DM framework. In this step we take a deeper dive to understand and analyze the data for the problem statement formalized in the previous step. This step begins with investigating the various sources of data outlined in the detailed project plan previously. These sources of data are then used to collect data, analyze different attributes, and make a note of data quality. This step also involves what is generally termed as exploratory data analysis.

Exploratory data analysis (**EDA**) is a very important sub-step. It is during EDA we analyze different attributes of data, their properties and characteristics. We also visualize data during EDA for a better understanding and uncovering patterns that might be previously unseen or ignored. This step lays down the foundation for the coming step and hence this step cannot be neglected at all.

Data preparation

This is the third and the most time-consuming step in any data science project. Data preparation takes place once we have understood the business problem and explored the data available. This step involves data integration, cleaning, wrangling, feature selection, and feature engineering. First and the foremost is data integration. There are times when data is available from various sources and hence needs to be combined based on certain keys or attributes for better usage.

Data cleaning and wrangling are very important steps. This involves handling missing values, data inconsistencies, fixing incorrect values, and converting data to ingestible formats such that they can be used by ML algorithms.

Data preparation is the most time-consuming step, taking over 60-70% of the overall time taken for any data science project. Apart from data integration and wrangling, this step involves selecting key features based on relevance, quality, assumptions, and constraints. This is also termed as **feature selection**. There are also times when we have to derive or generate features from existing ones. For example, deriving age from date of birth and so on, depending upon the use case requirements. This step is termed as **feature engineering** and is again required based on use case.

Modeling

The fourth step or the modeling step is where the actual analysis and ML takes place. This step utilizes the clean and formatted data prepared in the previous step for modeling purposes. This is an iterative process and works in sync with the data preparation step as models/algorithms require data in different settings/formats with varying set of attributes.

This step involves selecting relevant tools and frameworks along with the selection of a modeling technique or algorithms. This step includes model building, evaluation, and fine-tuning of models, based on the expectations and criteria laid down during the business understanding phase.

Evaluation

Once the modeling step results in a model(s) that satisfies the success criteria, performance benchmarks, and model evaluation metrics, a thorough evaluation step comes into picture. In this step, we consider the following activities before moving ahead with the deployment stage:

- Model result assessment based on quality and alignment with business objectives
- Identifying any additional assumptions made or constraints relaxed
- Data quality, missing information, and other feedback from data science team and/or **subject matter experts** (**SMEs**)
- Cost of deployment of the end-to-end ML solution

Deployment

The final step of the CRISP-DM model is deployment to production. The models that have been developed, fined-tuned, validated, and tested during multiple iterations are saved and prepared for production environment. A proper deployment plan is built, which includes details on hardware and software requirements. The deployment stage also includes putting in place checks and monitoring aspects to evaluate the model in production for results, performance, and other metrics.

Standard ML workflow

The CRISP-DM model provides a high-level workflow for management of ML and related projects. In this section, we will discuss the technical aspects and implementation of standard workflows for handling ML projects. Simply put, an ML pipeline is an end-to-end workflow consisting of various aspects of a data intensive project. Once the initial phases such as business understanding, risk assessments, and ML or data mining techniques selection have been covered, we proceed towards the solution space of driving the project. A typical ML pipeline or workflow with different sub-components is shown in the following diagram:
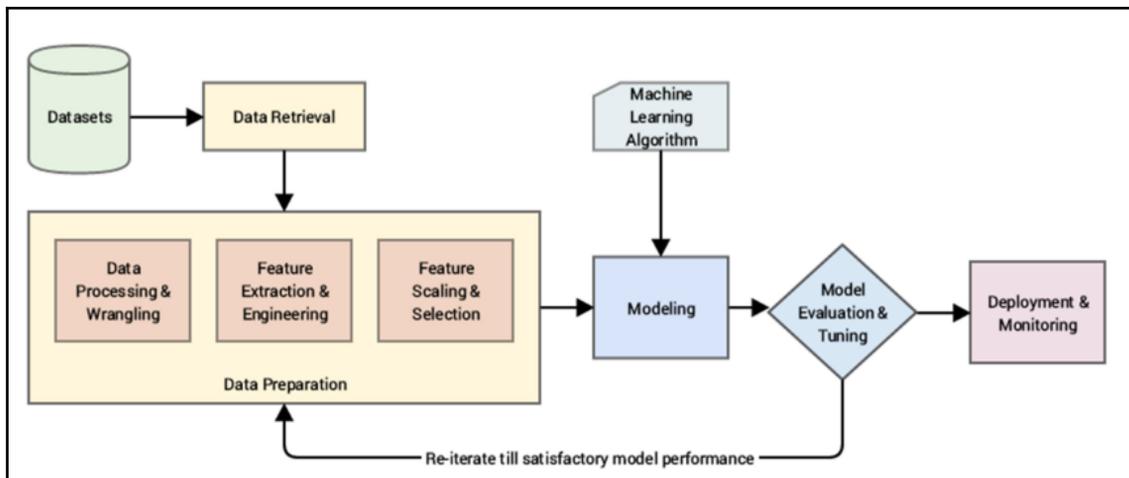
Typical ML pipeline

A standard ML pipeline broadly consists of the following stages.

Data retrieval

Data collection and extraction is where the story usually begins. Datasets come in all forms including structured and unstructured data that often includes missing or noisy data. Each data type and format needs special mechanisms for data handling as well as management. For instance, if a project concerns analysis of tweets, we need to work with Twitter APIs and develop mechanisms to extract the required tweets, which are usually in JSON format.

Other scenarios may involve already existing structured or unstructured public datasets or private ones, both may require additional permissions apart from just developing extraction mechanisms. A fairly detailed account pertaining to working with diverse data formats is discussed in Chapter 3 of the book *Practical Machine Learning with Python*, Sarkar and their co-authors, Springer, 2017 in case you are interested in diving deeper into further details.

Data preparation

It is worth reiterating that this is where the maximum time is spent in the whole pipeline. This is a fairly detailed step that involves fundamental and important sub-steps, which include:

- Exploratory data analysis
- Data processing and wrangling
- Feature engineering and extraction
- Feature scaling and selection

Exploratory data analysis

So far, all the initial steps in the project have revolved around business context, requirements, risks, and so on. This is the first touch point where we actually explore in depth the data that is collected/available. EDA helps us understand various facets of our data. In this step, we analyze different attributes of data, uncover interesting insights, and even visualize data on different dimensions to get a better understanding.

This step helps us gather important characteristics of the dataset at hand, which not only is useful in later stages of the project but also helps us identify and/or mitigate potential issues early in the pipeline. We cover an interesting example later on in this chapter for readers to understand the process and importance of EDA.

Data processing and wrangling

This step is concerned with the transformation of data into a usable form. The raw data retrieved in the first step is in most cases unusable by ML algorithms. Formally, data wrangling is the process of cleaning, transforming, and mapping data from one form to another for consumption in later stages of the project life cycle. This step includes missing data imputation, typecasting, handling duplicates and outliers, and so on. We will cover these steps in the context of use-case driven chapters for a better understanding.

Feature engineering and extraction

Preprocessed and wrangled data reaches the state where it can be utilized by the feature engineering and extraction step. In this step, we utilize existing attributes to derive and extract context/use-case specific attributes or features that can be utilized by ML algorithms in the coming stages. We employ different techniques based on data types.

Feature engineering and extraction is a fairly involved step and hence is discussed in more detail in the later part of this chapter.

Feature scaling and selection

There are cases when the number of features available is so large that it adversely affects the overall solution. Not only is the processing and handling of a dataset with a huge number of attributes an issue but it also leads to difficulty in interpretation, visualization, and many more. These issues are formally termed as the **curse of dimensionality**.

Feature selection thus helps us identify representative sets of features that can be utilized in the modeling step without much loss of information. There are different techniques to perform feature selection; some of them are discussed in the later sections of the chapter.

Modeling

In the process of modeling, we usually feed the data features to a ML method or algorithm and train the model, typically to optimize a specific cost function, in most cases with the objective of reducing errors and generalizing the representations learned from the data.

Depending upon the dataset and project requirements, we apply one or a combination of different ML techniques. These can include supervised techniques such as classification or regression, unsupervised techniques such as clustering, or even a hybrid approach combining different techniques (as discussed earlier in the *ML techniques* sections).

Modeling is usually an iterative process, and we often leverage multiple algorithms or methods and choose the best model, based on model evaluation performance metrics. Since this is a book about transfer learning, we will mostly be building deep learning based models in subsequent chapters, but the basic principles of modeling are quite similar to ML models.

Model evaluation and tuning

Developing a model is just one portion of learning from data. Modeling, evaluation, and tuning are iterative steps that help us fine-tune and select the best performing models.

Model evaluation

A model is basically a generalized representation of data and the underlying algorithm used for learning this representation. Thus, model evaluation is the process of evaluating the built model against certain criteria to assess its performance. Model performance is usually a function defined to provide a numerical value to help us decide the effectiveness of any model. Often, cost or loss functions are optimized to build an accurate model based on these evaluation metrics.

Depending upon the modeling technique used, we leverage relevant evaluation metrics. For supervised methods, we usually leverage the following techniques:

- Creating a confusion matrix based on model predictions versus actual values. This covers metrics such as **True Positive** (**TP**), **False Positive** (**FP**), **True Negative** (**TN**), and **False Negative** (**FN**) considering one of the classes as the positive class (which is usually a class of interest).
- Metrics derived from the confusion matrix, which include accuracy (overall performance), precision (predictive power of the model), recall (hit rate), and the F1-score (harmonic mean of precision and recall).

- The **receiver operator characteristic** (**ROC**) curve and the **area under curve** (**AUC**) metric, which represents the AUC.
- R-square (coefficient of determination), **root mean square error** (**RMSE**), F-statistic, **Akaike information criterion** (**AIC**), and p-values specifically for regression models.

Popular metrics for evaluating unsupervised methods such as clustering include the following:

- Silhouette coefficients
- Sum of squared errors
- Homogeneity, completeness, and the V-measure
- Calinski-Harabaz index

> Do note that this list depicts the most popular metrics, which are extensively used, but is by no means an exhaustive list of model evaluation metrics.

Cross-validation is also an important aspect of the model evaluation process where we leverage validation sets based on cross-validation strategies to evaluate model performance by tuning various hyperparameters of the model. You can think of hyperparameters as knobs that can be used to tune the model to build efficient and better performing models. The usage and details of these evaluation techniques will be more clear when we use them for evaluating our models in subsequent chapters with extensive hands-on examples.

Bias variance trade-off

Supervised learning algorithms help us infer or learn a mapping from input data points to output signals. This learning results in a target or a learned function. Now, in an ideal scenario, the target function would learn the exact mapping between input and output variables. Unfortunately, there are no ideals.

As discussed while introducing supervised learning algorithms, we utilized a subset of data called the **training dataset** to learn the target function and then test the performance on another subset called the **test dataset**. Since the algorithm only sees a subset of all possible combinations of data, there arises an error between the predicted outputs and the observed outputs. This is called the **total error** or the **prediction error**:

Total Error = Bias Error + Variance Error + Irreducible Error

The irreducible error is the inherent error introduced due to noise, the way we have framed the problem, collected the data, and so on. As the name suggests, this error is irreducible, and we can do little from an algorithmic point of view to handle this.

Bias

The term **bias** refers to the underlying assumptions made by a learning algorithm to infer the target function. High bias suggests that the algorithm makes more assumptions about the target function while low bias suggests lesser assumptions.

The error due to bias is simply the difference between the expected (or average) prediction values and the actual observed values. To get an average of predictions, we repeat the learning step multiple times and then average the results. Bias error helps us understand how well the model generalizes. Low bias algorithms are usually non-parametric algorithms such as decision trees, SVMs, and so on, while parametric functions such as linear and logistic regression are high on bias.

Variance

Variance marks the sensitivity of a model towards the training dataset. As we know, the learning phase relies on a small subset of all possible data combinations called the training set. Thus, variance error captures the changes in the model's estimates as the training dataset changes.

Low variance suggests significantly fewer changes to prediction values, as the underlying training dataset changes while high variance points in the other direction. Non-parametric algorithms such as decision trees have high variance, while parametric algorithms such as linear regression are less flexible and hence low on variance.

Trade-off

The bias-variance trade-off is the problem of simultaneously reducing the bias and variance errors of a supervised learning algorithm, which prevents the target function from generalizing well beyond the training data points. Let's have a look at the following illustrations:
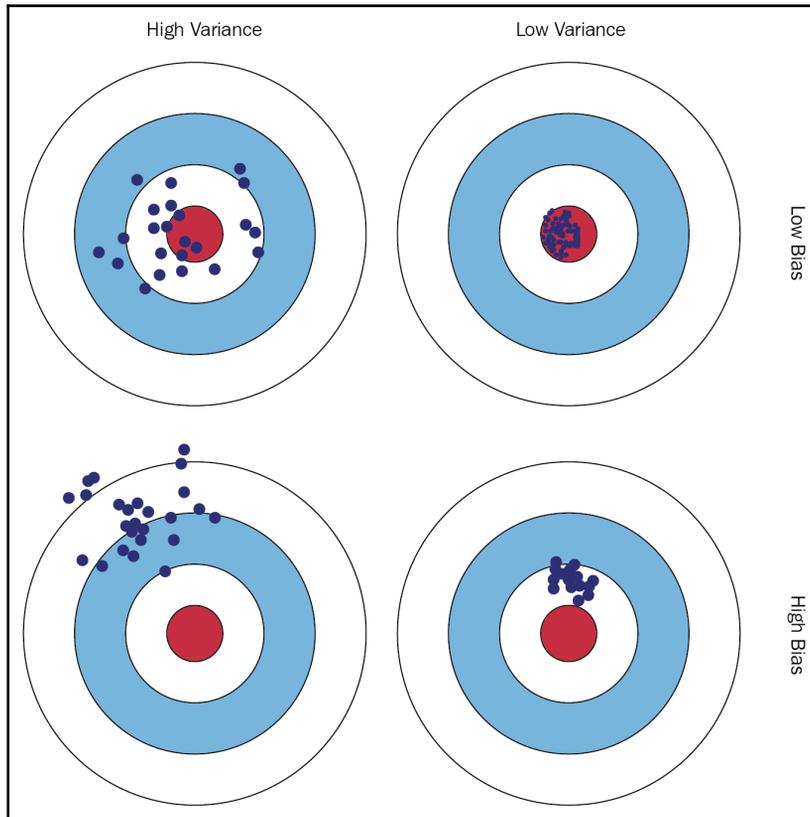
Bias variance trade-off

Readers are encouraged to visit the following links for a better and in-depth understanding of bias-variance trade-off: `http://scott.fortmann-roe.com/docs/BiasVariance.html` and `https://elitedatascience.com/bias-variance-tradeoff`.

Consider that we are given a problem statement as: g*iven a person's height, determine his/her weight*. We are also given a training dataset with corresponding values for height and weight. The data is shown in the following diagram:
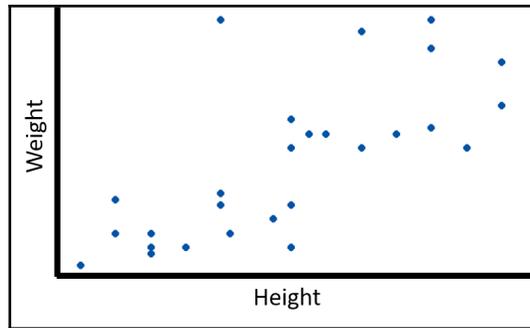
Plot depicting height-weight dataset

Please note that this is a toy example to explain important concepts, we will use real-world cases in subsequent chapters while solving actual problems.

This is an instance of supervised learning problem, more so of a regression problem (see why?). Utilizing this training dataset, our algorithm would have to learn the target function to find a mapping between heights and weights of different individuals.

Underfitting

Based on our algorithm, there could be different outputs of the training phase. Let's assume that the learned target function is as shown in the following diagram:
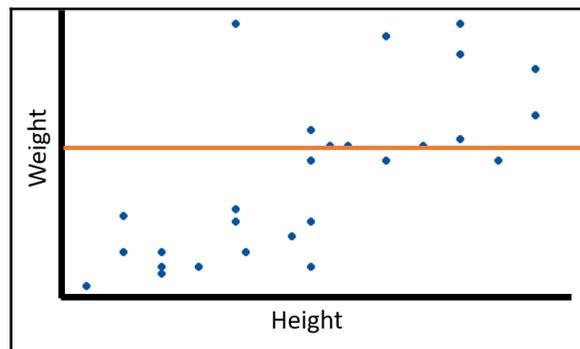
Underfit model

This lazy function always predicts a constant output value. Since the target function is not able to learn the underlying structure of the data, it results in what is termed as **underfitting**. An underfit model has a poor predictive performance.

Overfitting

The other extreme of the training phase is termed as overfitting. The overfitting graph can be represented as follows:
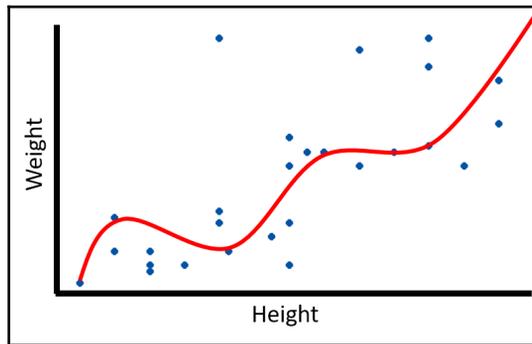
Overfit model

This shows a target function that perfectly maps each data point in our training dataset. This is better known as **model overfitting**. In such cases, the algorithm tries to learn the exact data characteristics, including the noise, and thus fails to predict reliably on new unseen data points.

Generalization

The sweet spot between underfitting and overfitting is what we term as a **good fit**. The graph for a model which may generalize well for the given problem is as follows:
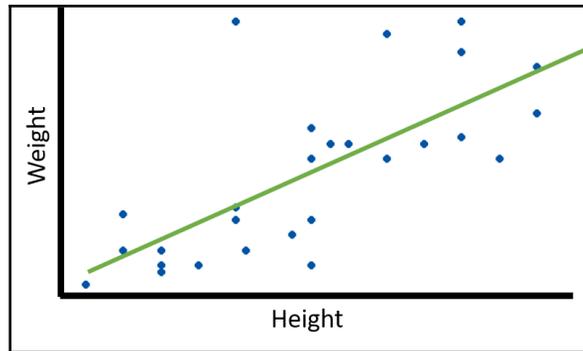
Well generalizing fit

A learned function that can perform well enough on unseen data points as well as on the training data is termed a **generalizable function**. Thus, generalization refers to how well a target function can perform on unseen data, based on the concepts learned during the training phase. The preceding diagram depicts a well generalizing fit.

Model tuning

Preparing and evaluating a model is as essential as tuning one. Working with different ML frameworks/libraries that provide us with the standard set of algorithms, we hardly ever use them straight out of the box.

ML algorithms have different parameters or knobs, which can be tuned based on the project requirements and different evaluation results. Model tuning works by iterating over different settings of hyperparameters or metaparameters to achieve better results. Hyperparameters are knobs at a high-level abstraction, which are set before the learning process begins.

> This is different from model level parameters, which are learned *during* the training phase. Hence, model tuning is also termed **hyperparameter optimization**.

Grid search, randomized hyperparameter search, Bayesian optimization, and so on are some of the popular ways of performing model tuning. Though model tuning is very important, overdoing it might impact the learning process adversely. Some of the issues related to overdoing the tuning process were discussed in the section *bias-variance trade-off*.

Deployment and monitoring

Once model development, evaluation, and tuning is complete, along with multiple iterations of improving the results, the final stage of model deployment comes into the picture. Model deployment takes care of aspects such as model persistence, exposing models to other applications through different mechanisms such as API endpoints, and so on, along with developing monitoring strategies.

We live in a dynamic world where everything changes every so often, and the same is true about data and other factors related to our use cases. It is imperative that we put in place monitoring strategies such as regular reports, logs, and tests to keep a check on the performance of our solutions and make changes as and when required.

ML pipelines are as much about software engineering as they are about data science and ML. We outlined and discussed the different components of a typical pipeline in brief. Depending upon specific use cases, we modify the standard pipeline to suit the needs yet make sure we do not overlook known pitfalls. In the coming sections, let's understand a couple of the components of a typical ML pipeline in a bit more detail, with actual examples and code snippets.

Exploratory data analysis

EDA is among the first few tasks we perform when we get started on any ML project. As discussed in the section on *CRISP-DM*, data understanding is an important step to uncover various insights about the data and better understand the business requirements and context.

In this section, we will take up an actual dataset and perform EDA using `pandas` as our data manipulation library, coupled with `seaborn` for visualization. Complete code snippets and details for this analysis are available in the Python Notebook `game_of_thrones_eda.ipynb`.

We first begin by importing the required libraries and setting up the configurations as shown in the following snippet:

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: from collections import Counter
   ...:
   ...: # plotting
   ...: import seaborn as sns
   ...: import matplotlib.pyplot as plt
   ...:
   ...: # setting params
   ...: params = {'legend.fontsize': 'x-large',
   ...:           'figure.figsize': (30, 10),
   ...:           'axes.labelsize': 'x-large',
   ...:           'axes.titlesize':'x-large',
   ...:           'xtick.labelsize':'x-large',
   ...:           'ytick.labelsize':'x-large'}
   ...:
   ...: sns.set_style('whitegrid')
   ...: sns.set_context('talk')
   ...:
   ...: plt.rcParams.update(params)
```

Once the settings and requirements are in place, we can begin concentrating on the data. The dataset in consideration for exploratory analysis is the `battles.csv` file, which contains all major battles from the world of *Game of Thrones* (up to season 5).

One of the most popular television series of all time, *Game of Thrones* is a fantasy drama set in the fictional continents of *Westeros* and *Essos*, filled with multiple plots and a huge number of characters all battling for the Iron Throne! It is an adaptation of the *A Song of Ice and Fire* novel series by George R. R. Martin. Being a popular series, it has caught the attention of many, and data scientists aren't to be excluded. This notebook presents EDA on the Kaggle dataset enhanced by Myles O'Neill (more details: `https://www.kaggle.com/mylesoneill/game-of-thrones`). This dataset is based on a combination of multiple datasets collected and contributed to by multiple people. We utilize the `battles.csv` in this analysis. The original battles data was presented by Chris Albon; more details are available at `https://github.com/chrisalbon/war_of_the_five_kings_dataset`.

The following snippet loads the `battles.csv` file using `pandas`:

```
In [2]: battles_df = pd.read_csv('battles.csv')
```

The dataset is as shown in the following screenshot:

	name	year	battle_number	attacker_king	defender_king	attacker_1	attacker_2	attacker_3	attacker_4	defender_1	...	major_death
0	Battle of the Golden Tooth	298	1	Joffrey/Tommen Baratheon	Robb Stark	Lannister	NaN	NaN	NaN	Tully	...	1.0
1	Battle at the Mummer's Ford	298	2	Joffrey/Tommen Baratheon	Robb Stark	Lannister	NaN	NaN	NaN	Baratheon	...	1.0
2	Battle of Riverrun	298	3	Joffrey/Tommen Baratheon	Robb Stark	Lannister	NaN	NaN	NaN	Tully	...	0.0
3	Battle of the Green Fork	298	4	Robb Stark	Joffrey/Tommen Baratheon	Stark	NaN	NaN	NaN	Lannister	...	1.0
4	Battle of the Whispering Wood	298	5	Robb Stark	Joffrey/Tommen Baratheon	Stark	Tully	NaN	NaN	Lannister	...	1.0

Sample rows from battles.csv of Game of Thrones

We can view the total number of rows, data types of each of the attributes, and general statistics of numerical attributes using the `pandas` utilities `shape`, `dtypes`, and `describe()` respectively. We have data about 38 battles, with 25 attributes describing each one of them.

Let's understand the distribution of battles across years from the fantasy land. The following snippet plots a bar graph of this distribution:

```
In [3]: sns.countplot(y='year',data=battles_df)
   ...: plt.title('Battle Distribution over Years')
   ...: plt.show()
```

The following plot shows that the highest number of battles were fought in the year **299**, followed by **300** and **298** respectively:
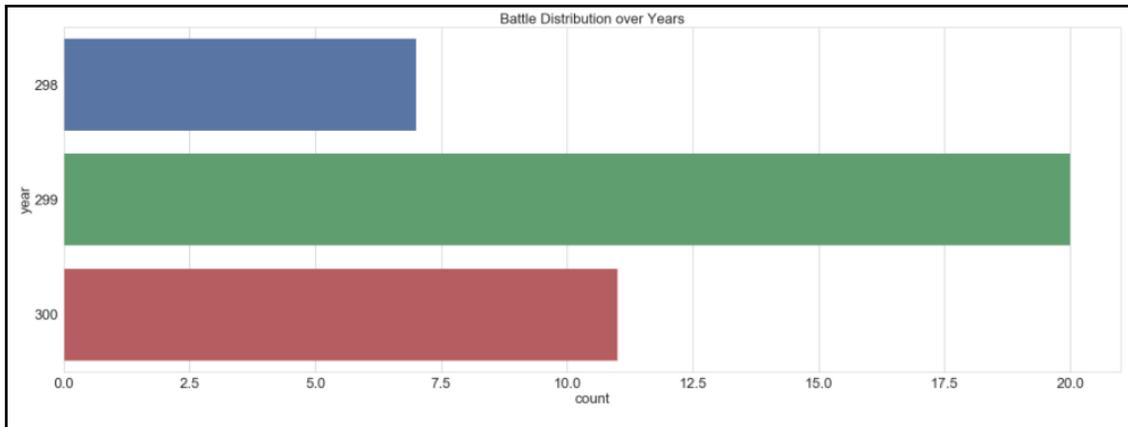
Battle distribution over years

There are different regions in this fantasy land, with battles taking place at every place imaginable. Yet, it would be interesting to see if there were any preferred regions. The following snippet helps us answer this question precisely:

```
In [4]: sns.countplot(x='region',data=battles_df)
...: plt.title('Battles by Regions')
...: plt.show()
```

The following plot helps us identify that **The Riverlands** have seen the most battles, followed by **The North** and **The Westerlands**:
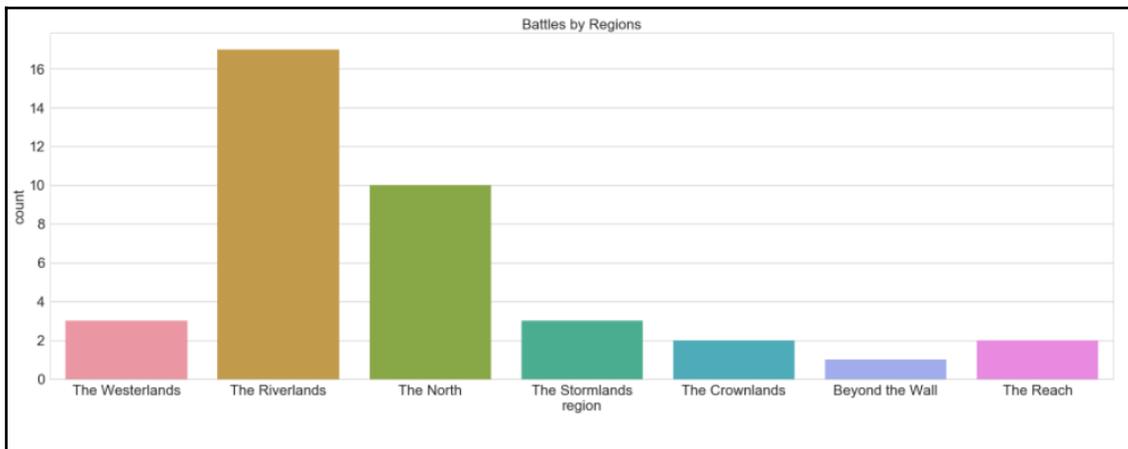
Battles by regions

Another interesting thing to notice is that there has been only one battle **Beyond the Wall** (spoiler alert: stay tuned for later seasons).

We can perform similar analysis using different group-by variations to understand, for instance, the number of major deaths, or captures per region, and so on.

We move on to see which king attacked the most. We visualize this using a pie chart to understand the percentage share of battles fought by each of the kings involved. Please note that we perform this analysis based on attacking kings. Similar analysis can be performed using defending kings as well. The following snippet prepares a pie chart to display each attacking king's share of battles:

```
In [5]: attacker_king = battles_df.attacker_king.value_counts()
   ...: attacker_king.name='' # turn off annoying y-axis-label
   ...: attacker_king.plot.pie(figsize=(6, 6),autopct='%.2f')
```

Each attacking king's share of battles is displayed in the following pie chart:
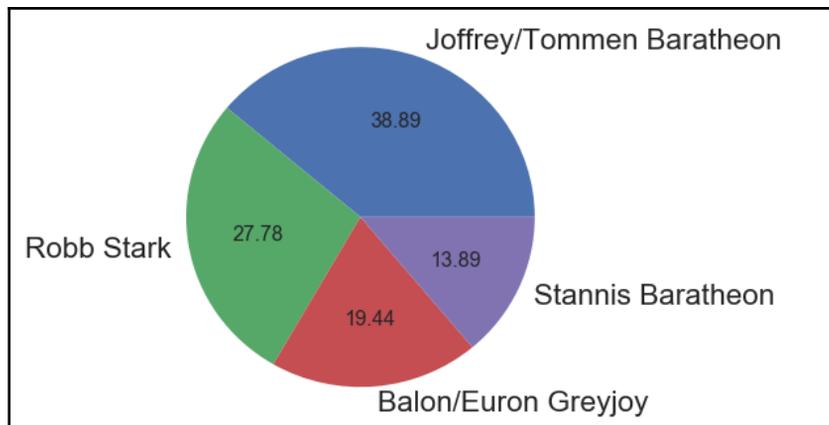
Battle share per attacking king

The lands of Westeros and Essos are dangerous with enemies and threats all across. Let's analyze the data a bit to understand on how many occasions each of the kings was a winner. Since a king can be either defending his land or attacking for power, it would be interesting to see the defending and attacking wins as well. The following snippet helps us prepare a stacked bar chart to analyze each king's attacking and defending wins:

```
In [6] : attack_winners = battles_df[battles_df.
    ...:                             attacker_outcome=='win']
    ...:                                 ['attacker_king'].
    ...:                                     value_counts().
    ...:                                     reset_index()
    ...:
    ...: attack_winners.rename(
    ...:         columns={'index':'king',
    ...:                     'attacker_king':'wins'},
    ...:                     inplace=True)
    ...:
    ...: attack_winners.loc[:,'win_type'] = 'attack'
    ...:
    ...: defend_winners = battles_df[battles_df.
    ...:                             attacker_outcome=='loss']
    ...:                             ['defender_king'].
    ...:                             value_counts().
    ...:                             reset_index()
    ...: defend_winners.rename(
    ...:         columns={'index':'king',
    ...:                     'defender_king':'wins'},
    ...:                     inplace=True)
    ...:
    ...: defend_winners.loc[:,'win_type'] = 'defend'
    ...:
    ...:
    ...: sns.barplot(x="king",
    ...:             y="wins",
    ...:             hue="win_type",
    ...:             data=pd.concat([attack_winners,
    ...:                             defend_winners]))
    ...: plt.title('Kings and Their Wins')
    ...: plt.ylabel('wins')
    ...: plt.xlabel('king')
    ...: plt.show()
```

The preceding snippet calculates the number of wins per king while attacking and then calculates the number of wins per king while defending. We then merge the two results and plot the same using a stacked barplot. The results are shown in the following graph:
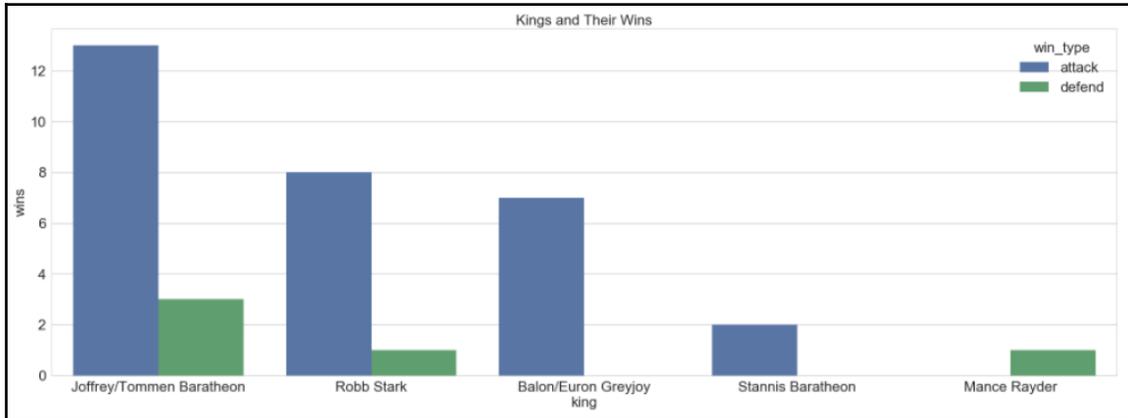
Number of wins per king

The preceding graph clearly shows that the **Baratheon** boys have the most number of wins, both while attacking as well as while defending. Seems like they have luck on their side so far. **Robb Stark** was the second most successful king, until of course the Red Wedding happened.

The dataset also contains attributes describing the number of houses involved, battle commanders, and army sizes. We can perform similar and more in-depth analysis to better understand the battles. We encourage the readers to try out a few of these as exercises and check the Python Notebook for more pointers.

Before we close the section, let's try to identify archenemies in the fight for the Iron Throne. Though the fans will already have a gut feeling about this, let's see what the data has to say about it. The following snippet helps us answer this question:

```
In [7]: temp_df = battles_df.dropna(
   ...:                 subset = ["attacker_king",
   ...:                           "defender_king"])[
   ...:                     ["attacker_king",
   ...:                      "defender_king"]
   ...:                     ]
   ...:
   ...: archenemy_df = pd.DataFrame(
   ...:                 list(Counter(
   ...:                     [tuple(set(king_pair))
```

```
...:                              for king_pair in temp_df.values
...:                              if len(set(king_pair))>1]).
...:                                items()),
...:                    columns=['king_pair',
...:                             'battle_count'])
...:
...: archenemy_df['versus_text'] = archenemy_df.
...:                                 apply(
...:                                     lambda row:
...:                                 '{} Vs {}'.format(
...:                                         row[
...:                                             'king_pair'
...:                                         ][0],
...:                                         row[
...:                                             'king_pair'
...:                                         ][1]),
...:                                     axis=1)
...: archenemy_df.sort_values('battle_count',
...:                          inplace=True,
...:                          ascending=False)
...:
...:
...: archenemy_df[['versus_text',
...:               'battle_count']].set_index('versus_text',
...:                                     inplace=True)
...: sns.barplot(data=archenemy_df,
...:             x='versus_text',
...:             y='battle_count')
...: plt.xticks(rotation=45)
...: plt.xlabel('Archenemies')
...: plt.ylabel('Number of Battles')
...: plt.title('Archenemies')
...: plt.show()
```

We first prepare a temporary dataframe and remove any battles that do not have either the attacking or defending king's name listed. Once we have a clean dataframe, we iterate over each of the rows and count the number of battles every pair has fought. We ignore cases where the battle was among the king's own army (`if len(set(king_pair))>1`). We then simply plot the results in a bar graph, shown as follows:
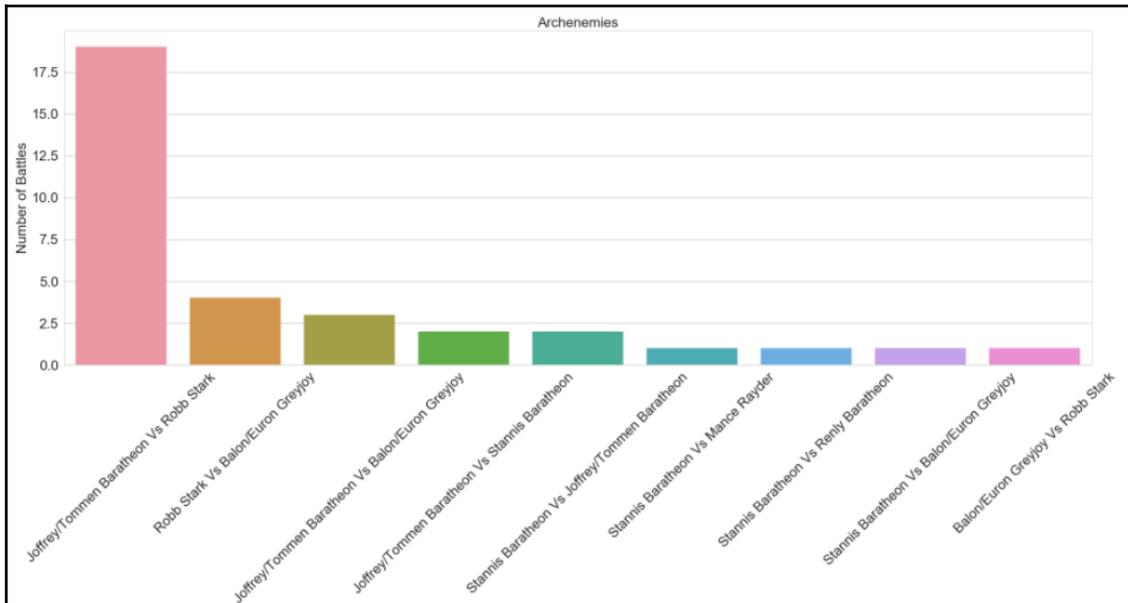
Archenemies from Game of Thrones

We see that the dataset confirms the gut feelings. **Robb Stark** and **Joffrey Baratheon** have fought a total of 19 battles already, with other pairs having fought five or fewer battles.

The analysis and visualizations shared in this section were a glimpse of what can be done on a dataset. There could be many more patterns and insights that could be extracted from this dataset alone.

EDA is a very powerful mechanism for understanding the dataset in detail before jumping into other stages of ML. In the coming chapters, we will regularly perform EDA to assist us in understanding the business problem along with the dataset before we go into modeling, tuning, evaluation, and deployment stages.

Feature extraction and engineering

Data preparation is the longest and the most complex phase of any ML project. The same was emphasized while discussing the CRISP-DM model, where we mentioned how the data preparation phase takes up about 60-70% of the overall time spent in a ML project.

Once we have our raw dataset preprocessed and wrangled, the next step is to make it usable for ML algorithms. Feature extraction is the process of deriving features from raw attributes. For instance, feature extraction while working with image data refers to the extraction of red, blue, and green channel information as features from raw pixel-level data.

On the same lines, feature engineering refers to the process of deriving additional features from existing ones using mathematical transformations. For instance, feature engineering would help us in deriving a feature such as annual income from a person's monthly income (based on use case requirements). Since both feature extraction and engineering help us transform raw datasets into usable forms the terms are used interchangeably by ML practioners.

Feature engineering strategies

The process of transforming raw datasets (post clean up and wrangling) into features that can be utilized by ML algorithms is a combination of domain knowledge, use case requirements, and specific techniques. Features thus depict various representations of the underlying data and are the outcome of the feature engineering process.

Since feature engineering transforms raw data into a useful representation of itself, there are various standard techniques and strategies that can be utilized, based on the type of data at hand. In this section we will discuss a few of those strategies, briefly covering both structured and unstructured data.

Working with numerical data

Numerical data, commonly available in datasets in the form of integers or floating-point numbers and popularly known as **continuous numerical data**, is usually a ML friendly data type. By friendly, we refer to the fact that numeric data can be ingested in most ML algorithms directly. This however, does not mean that numeric data does not require additional processing and feature engineering steps.

There are various techniques for extracting and engineering features from numerical data. Let's look at some of those techniques in this section:

- **Raw measures**: These data attributes or features can be used directly in their raw or native format as they occur in the dataset without any additional processing. Examples can be age, height, or weight (as long as data distributions are not too skewed!).
- **Counts**: Numeric features such as counts and frequencies are also useful in certain scenarios to depict important details. Examples can be the number of credit card fraud occurences, song listen counts, device event occurences, and so on.
- **Binarization**: Often we might want to binarize occurrences or features, especially to just indicate if a specific item or attribute was present (usually denoted with a 1) or absent (denoted with a 0). This is useful in scenarios like building recommendation systems.
- **Binning**: This technique typically bins or groups continuous numeric values from any feature or attribute under analysis to discrete bins, such that each bin covers a specific numeric range of values. Once we get these discrete bins, we can choose to further apply categorical data-based feature engineering on the same. Various binning strategies exist, such as fixed-width binning and adaptive binning.

Code snippets to better understand feature engineering for numeric data are available in the notebook `feature_engineering_numerical_and_categorical_data.ipynb`.

Working with categorical data

Another important class of data commonly encountered is categorical data. Categorical features have discrete values that belong to a finite set of classes. These classes may be represented as text or numbers. Depending upon whether there is any order to the classes or not, categorical features are termed as **ordinal** and **nominal** respectively.

Nominal features are those categorical features that have a finite set of values but do not have any natural ordering to them. For instance, weather seasons, movie genres, and so on are all nominal features. Categorical features that have a finite set of classes with a natural ordering to them are termed as **ordinal features**. For instance, days of the week, dress sizes, and so on are ordinals.

Typically, any standard workflow in feature engineering involves some form of transformation of these categorical values into numeric labels and then the application of some encoding scheme on these values. Popular encoding schemes are briefly mentioned as follows:

- **One-hot encoding**: This strategy creates n binary valued columns for a categorical attribute assuming there are n number of distinct categories
- **Dummy coding**: This strategy creates $n-1$ binary valued columns for a categorical attribute assuming there are n number of distinct categories
- **Feature hashing**: This strategy is leveraged where we use a hash function to add several features into a single bin or bucket (new feature), which is popularly used when we have a large number of features

Code snippets to better understand feature engineering for categorical data are available in the notebook `feature_engineering_numerical_and_categorical_data.ipynb`.

Working with image data

Image or visual data is a rich source of data, with several use cases that can be solved using ML algorithms and deep learning. Image data poses a lot of challenges and requires careful preprocessing and transformation before it can be utilized by any of the algorithms. Some of the most common ways of performing feature engineering on image data are as follows:

- **Utilize metadata information or EXIF data**: Attributes such as image creation date, modification date, dimensions, compression format, device used to capture the image, resolution, focal length, and so on.
- **Pixel and channel information**: Every image can be considered as a matrix of pixel values or a (m, n, c) matrix where m represents the number of rows, n represents the number of columns, and c points to color channels (for instance R, G, and B). Such a matrix can be then transformed into different shapes as per requirements of the algorithm and use case.
- **Pixel intensity**: Sometimes it is difficult to work with colored images that have multiple channels across colors. Pixel intensity-based feature extraction relies on binning pixels based on intensities rather than utilizing raw pixel-level values.
- **Edge detection**: Sharp changes in contrast and brightness between neighboring pixels can be utilized to identify object edges. There are different algorithms available for edge detection.

- **Object detection**: We take the concept of edge detection and extend it to object detection and then utilize identified object boundaries as useful features. Again, different algorithms may be utilized based on the type of image data available.

Deep learning based automated feature extraction

The feature extraction methods for image data and other types discussed so far require a lot of time, effort, and domain understanding. This kind of feature extraction has its merits along with its limitations.

Lately, deep learning, specifically **Convolutional Neural Networks** (**CNNs**), have been studied and utilized as automated feature extractors. A CNN is a special case of deep neural networks optimized for image data. At the core of any CNN are convolutional layers, which basically apply sliding filters across the height and width of the image. The dot product of pixel values and these filters results in activation maps that are learned across multiple epochs. At every level, these convolutional layers help in extracting specific features such as edges, textures, corners, and so on.

There is more to deep learning and CNNs, but, to keep things simple, let's assume that at every layer, CNNs help us extract different low and high-level features automatically. This in turn saves us from manually performing feature extraction. We will study CNNs in more detail in coming chapters and see how they help us extract features automatically.

Working with text data

Numerical and categorical features are what we call structured data types. They are easier to process and utilize in ML workflow. Textual data is one major source of unstructured information that is equally important. Textual data presents multiple challenges related to syntactical understanding, semantics, format, and content. Textual data also presents issues of transformation into numeric form before it can be utilized by ML algorithms. Thus, feature engineering for textual data is preceded by rigorous preprocessing and clean up steps.

Text preprocessing

Textual data requires careful and diligent preprocessing before any feature extraction/engineering can be performed. There are various steps involved in preprocessing textual data. The following is a list of some of the most widely used preprocessing steps for textual data:

- Tokenization
- Lowercasing
- Removal of special characters
- Contraction expansions
- Stopword removal
- Spell corrections
- Stemming and lemmatization

We will be covering most techniques in detail in the chapters related to use cases. For a better understanding, readers may refer to *Chapter 4* and *Chapter 7* of *Practical Machine Learning with Python*, Sarkar and their co-authors, Springer, 2017.

Feature engineering

Once we have our textual data properly processed via the methods mentioned in the previous section, we can utilize some of the following techniques for feature extraction and transformation into numerical form. Code snippets to better understand feature engineering for textual data are available in the Jupyter Notebook `feature_engineering_text_data.ipynb`:

- **Bag-of-words model**: This is by far the simplest vectorization technique for textual data. In this technique, each document is represented as a vector on N dimensions, where N indicates all possible words across the preprocessed corpus, and each component of the vector either denotes the presence of the word or its frequency.

- **TF-IDF model**: The bag-of-words model works under very simplistic assumptions and at certain times leads to various issues. One of the most common issues is related to some words overshadowing the rest of the words due to very high frequency, as the bag-of-words model utilizes absolute frequencies to vectorize. The **Term Frequency-Inverse Document Frequency (TF-IDF)** model mitigates this issue by scaling/normalizing the absolute frequencies. Mathematically, the model is defined as follows:

$$tfidf\,(w,\,D) = tf\,(W,\,D) * idf\,(w,\,D)$$

  Here, *tfidf (w, D)* denotes the TF-IDF score of each word *w* in document *D*, *tf (w, D)* is the frequency of word *w* in document *D* and *idf (w, D)* denotes the inverse document frequency, calculated as the log transformation of total documents in corpus *C* divided by frequency of documents where *w* occurs.

Apart from bag of words and TF-IDF, there are other transformations, such as bag of N-grams, and word embeddings such as Word2vec, GloVe, and many more. We will cover several of them in detail in subsequent chapters.

Feature selection

The process of feature extraction and engineering helps us extract as well as generate features from underlying datasets. There are cases where this leads to large inputs to an algorithm for processing. It such cases, it is suspected that many of the features in the input might be redundant and may lead to complex models and even overfitting. Feature selection is the process of identifying representative features from the complete feature set that is available/generated. The selected set of features are expected to contain the required information such that the algorithm is able to solve the given task without running into processing, complexity, and overfitting issues. Feature selection also helps in better understanding the data that is being used for the modeling process along with making processing quicker.

Feature selection methods can be broadly classified into the following three categories:

- **Filter methods**: As the name suggests, these methods help us rank features based on a statistical score. We then select a subset of these features. These methods are usually not concerned with model outputs, rather evaluating features independently. Threshold based techniques and statistical tests such as correlation coefficients and chi-squared tests are some popular choices.
- **Wrapper methods**: These methods perform a comparative search on the performance of different combinations of subsets of features, and then help us select the best performing subset. Backward selection and forward elimination are two popular wrapper methods for feature selection.
- **Embedded methods**: These methods provide the best of the preceding two methods by learning which subset of features would be the best. Regularization and tree based methods are popular choices.

Feature selection is an important aspect in the process of building a ML system. It is also one of the major sources of biases that can get into the system if not handled with care. Readers should note that feature selection should be done using a dataset separate from your training dataset. Utilizing the training dataset for feature selection would invariably lead to overfitting, while utilizing the test set for feature selection would overestimate the model's performance.

Most popular libraries provide a wide array of feature selection techniques. Libraries such as `scikit-learn` provide these methods out of the box. We will see and utilize many of them in subsequent sections/chapters.

Summary

A firm foundation and common ground for the understanding of concepts and techniques is very important for any journey. Through this chapter on ML fundamentals, we have tried to achieve precisely this. Before getting started with the basics of deep learning, transfer learning, and more advanced concepts, it is imperative that we form a solid foundation of ML concepts. In this chapter, we have covered quite a bit of ground and provided important pointers to study concepts in more details.

We began the chapter by understanding why machine learning is important and how it is a completely different paradigm. We briefly discussed the relationship between AI, ML, and deep learning. The chapter then moved on to present different ML techniques such as supervised, unsupervised, and reinforcement learning. We discussed in detail which different supervised and unsupervised methods are commonly used.

The chapter also included a quick introduction to the CRISP-DM model for ML project workflows along with ML pipelines. We also discussed EDA of the battles dataset from the fantasyland of Game of Thrones to apply different concepts and learn about the importance of EDA. Toward the end of the chapter, feature extraction and engineering and feature selection were introduced.

In the coming chapters, we will build upon these concepts and eventually apply the learning in chapters concerning different real-world use cases. Welcome onboard!

Deep Learning Essentials 2

This chapter provides a whirlwind tour of deep learning essentials, starting from the very basics of what deep learning really means, and then moving on to other essential concepts and terminology around neural networks. The reader will be given an overview of the basic building blocks of neural networks, and how deep neural networks are trained. Concepts surrounding model training, including activation functions, loss functions, backpropagation, and hyperparameter-tuning strategies will be covered. These foundational concepts will be of great help for both beginners and experienced data scientists who are venturing into deep neural network models. Special focus has been given to how to set up a robust cloud-based deep learning environment with GPU support, along with tips for setting up an in-house deep learning environment. This should be very useful for readers looking to build large-scale deep learning models on their own. The following topics will be covered in the chapter:

- What is deep learning?
- Deep learning fundamentals
- Setting up a robust, cloud-based deep learning environment with GPU support
- Setting up a robust, on-premise deep learning environment with GPU support
- Neural network basics

What is deep learning?

In **machine learning** (**ML**), we try to automatically discover rules for mapping input data to a desired output. In this process, it's very important to create appropriate representations of data. For example, if we want to create an algorithm to classify an email as spam/ham, we need to represent the email data numerically. One simple representation could be a binary vector where each component depicts the presence or absence of a word from a predefined vocabulary of words. Also, these representations are task-dependent, that is, representations may vary according to the final task that we desire our ML algorithm to perform.

In the preceding email example, instead of identifying spam/ham if we want to detect sentiment in the email, a more useful representation of the data could be binary vectors where the predefined vocabulary consists of words with positive or negative polarity. The successful application of most of the ML algorithms, such as random forests and logistic regression, depends on how good the data representation is. How do we get these representations? Typically, these representations are human-crafted features that are designed iteratively by making some intelligent guesses. This step is called **feature-engineering**, and is one of the crucial steps in most ML algorithms. **Support Vector Machines** (**SVMs**), or kernel methods in general, try to create more relevant representations of the data by transforming the hand-crafted representation of data into a higher-dimensional-space representation where solving the ML task using either classification or regression becomes easy. However, SVMs are hard to scale to very large datasets and are not that successful for problems such as image-classification and speech-recognition. Ensemble models, such as random forests and **Gradient Boosting Machines** (**GBMs**), create a collection of weak models that are specialized to do a small task well and then combine these weak models in some way to arrive at the final output. They work quite well when we have very large input dimensions, and creating handcrafted features is a very time-consuming step. In summary, all the previously mentioned ML methods work with a shallow representation of data involving the representation of data by a set of handcrafted features followed by some non-linear transformations.

Deep learning is a subfield of ML, where a hierarchical representation of the data is created. Higher levels of the hierarchy are formed by the composition of lower-level representations. More importantly, this hierarchy of representation is learned automatically from data by completely automating the most crucial step in ML, called **feature-engineering**. Automatically learning features at multiple levels of abstraction allows a system to learn complex representations of the input to the output directly from data, without depending completely on human-crafted features.

A deep learning model is actually a neural network with multiple hidden layers, which can help create layered hierarchical representations of the input data. It is called *deep* because we end up using multiple hidden layers to get the representations. In the simplest of terms, deep learning can also be called **hierarchical feature-engineering** (of course, we can do much more, but this is the core principle). One simple example of a deep neural network can be a **multilayered perceptron** (**MLP**) with more than one hidden layer. Let's consider the MLP-based face-recognition system in the following figure. The lowest-level features that it learns are some edges and patterns of contrasts. The next layer is then able to use those patterns of local contrast to resemble eyes, noses, and lips. Finally, the top layer uses those facial features to create face templates. The deep network is composing simple features to create features of increasing complexity, as depicted in the following diagram:
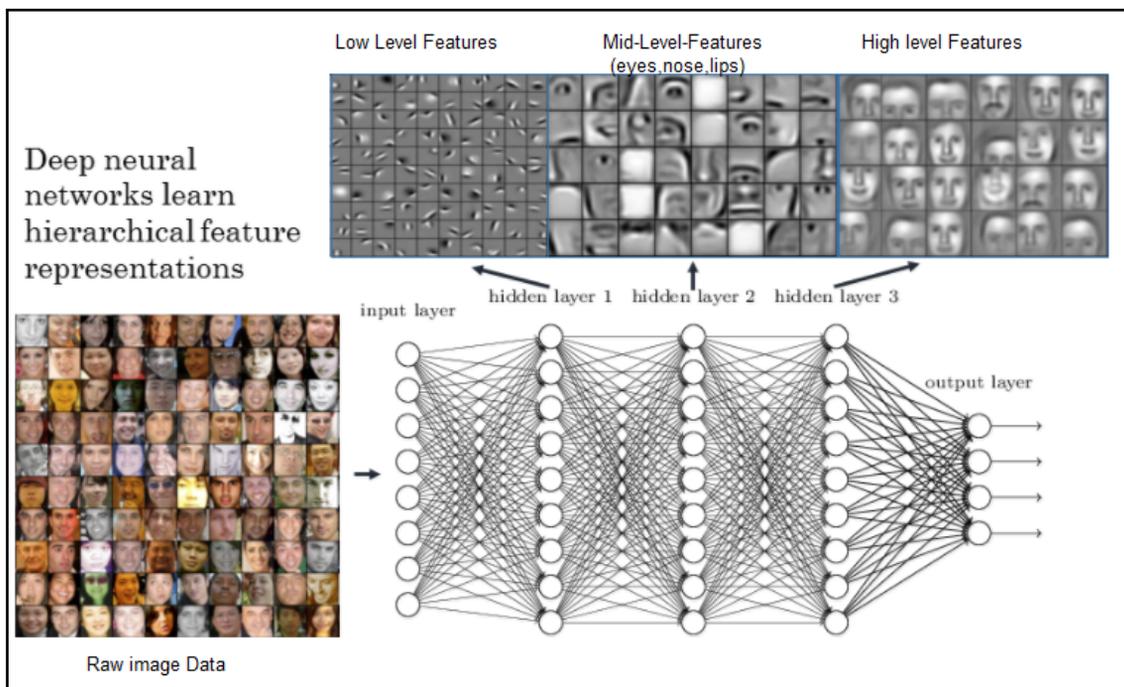
Hierarchical feature representation with deep neural nets (source: https://www.rsipvision.com/exploring-deep-learning/)

To understand deep learning, we need to have a clear understanding of the building blocks of neural networks, how these networks are trained, and how we are able to scale such training algorithms to very large deep networks. Before we dive into more details about neural networks, let's try to answer one question: Why deep learning now? The theory of neural networks, even the **Convolution Neural Networks** (**CNNs**) were built back in the 1990s. The reason they've became more popular now is due to the following three reasons:

- **Availability of efficient hardware**: Moore's law has enabled CPUs with better and faster processing capability and computing power. Besides this, GPUs have also been really useful in computing millions of matrix operations at scale, which is the most common operation in any deep learning model. The availability of SDKs, such as CUDA, have helped research communities rewrite some highly parallelizable jobs to run on a few GPUs, replacing huge CPU clusters. Model training involves many small linear algebra operations, such as matrix multiplications and dot products, which were very efficiently implemented in CUDA to run in GPUs.
- **Availability of large data sources and cheaper storage**: We now have free access to huge collections of labeled training sets for text, image, and speech.

- **Advances in optimization algorithms that are used to train neural networks**: Traditionally, there was only one algorithm used to learn the weights in a neural network, gradient descent or **Stochastic Gradient Descent** (**SGD**). SGD has a few limitations, such as getting stuck in a local minima and slow convergence, that are overcome by the newer algorithms. We will discuss these algorithms in detail in the later sections on *Neural network basics*.

Deep learning frameworks

One of the primary reasons for the widespread popularity and adoption of deep learning is the Python deep learning ecosystem, which consists of easy-to-use open source deep learning frameworks. However, the deep learning landscape is rapidly changing, considering how new frameworks keep getting launched and older ones reach the end of their life. Deep learning enthusiasts might know that Theano was the first and most popular deep learning framework, created by MILA (`https://mila.quebec/`), which was headed by Yoshua Bengio. Unfortunately, it was recently announced that further development and support for Theano is ending after the launch of its latest version (1.0) in 2017. Hence, it is of paramount importance to understand what frameworks are out there that can be leveraged to implement and solve deep learning. Another point to remember here is that several organizations themselves are building, acquiring, and launching these frameworks (often trying to compete with each other in terms of better features, faster execution, and so on) for everyone's benefit. The following figure showcases some of the most popular deep learning frameworks, as of 2018:
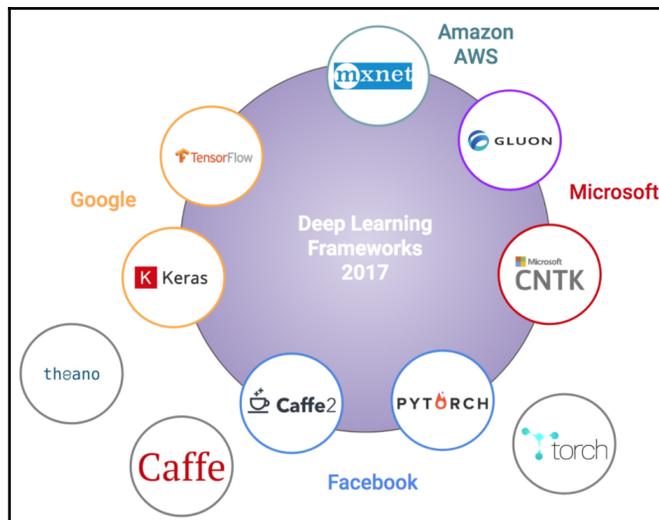
You can also check out the source article of this excellent depiction by Indra den Bakkar at **Towards Data Science** (`https://towardsdatascience.com/battle-of-the-deep-learning-frameworks-part-i-cff0e3841750`) for more details. Let's briefly look at some of the most popular deep learning frameworks:

- **Theano**: By default, Theano is a low-level framework enabling efficient numeric computations on multi-dimensional arrays, popularly known now as **tensors**. Theano is quite stable and the syntax is very similar to TensorFlow. It does have GPU support, but is quite limited, especially if we want to use multiple GPUs. Due to its development and support ceasing after 1.0, care should be taken if you intend to use theano for your deep learning implementations.

- **TensorFlow**: This is perhaps the most (or at least one of the most) popular deep learning framework. Created by Google Brain and open sourced in 2015, it has quickly gathered the attention of ML, deep learning researchers, engineers, and data scientists. While initial releases had problems with performance, it is under active development and keeps getting better with each release. TensorFlow has support for multi-CPU and GPU-based executions, as well as supporting multiple languages, including C++, Java, R, and Python. It only used to support the symbolic programming style to build deep learning models, which is slightly more complex, but given the widespread adoption since v1.5, it has started supporting the more popular and easy-to-use imperative programming style (also called **eager execution**). TensorFlow in general is a low-level library, similar to Theano, but also has capabilities for leveraging high-level APIs for rapid prototyping and development. An important part of TensorFlow also includes the `tf.contrib` module, which contains various experimental features, including the Keras API itself!

- **Keras**: If you find yourself getting perplexed with leveraging low-level deep learning frameworks to solve problems, you can always rely on Keras! This framework is used widely by people with varied skill sets, including scientists who may not be hardcore developers. This is because Keras provides a simple, clean, and easy-to-use high-level API for building efficient deep learning models with minimal code. The beauty of this is that it can be configured to work on top of multiple low-level deep learning frameworks (called **backends**), including theano and TensorFlow. The Keras documentation is accessible at `https://keras.io/` and is quite detailed.

- **Caffe**: This is also one of the first and relatively older deep learning frameworks developed in C++ (including Python bindings) by the Berkeley Vision and Learning Center. The best part about Caffe is that it offers many pretrained deep learning models as a part of the Caffe Model Zoo. Facebook recently open sourced Caffe2, which is built upon Caffe with more enhancements and features and is easier to use than its predecessor.

- **PyTorch:** The Torch framework is written in Lua and is extremely flexible and fast, often giving huge performance gains. PyTorch is a Python-based framework for building deep learning models, that draws inspiration from Torch. It is not just an extension or a Python wrapper over Torch but a complete framework in itself, improving various aspects of the Torch framework architecture. This includes getting rid of containers, leveraging modules, and also performance improvements such as memory optimizations.

- **CNTK**: The Cognitive Toolkit framework has been open sourced by Microsoft and supports both Python and C++. The syntax is very similar to Keras, and it supports a wide variety of model architectures. Though not immensely popular, this is the framework that Microsoft uses internally for several of their cognitive-intelligence capabilities.

- **MXNet**: This has been developed by the **Distributed Machine Learning Community** (**DMLC**), the creators of the very popular XGBoost package. This is now an official Apache Incubator project. MXNet was one of the first deep learning frameworks to support diverse languages, including C++, Python, R, and Julia, as well as multiple operating systems, including Windows, which is often neglected by other deep learning frameworks. The framework is extremely efficient and scalable, with support for multi-GPUs. Hence, it has become Amazon's deep learning framework of choice for which it has also developed a high-level interface, called **Gluon**.

- **Gluon**: This is a high-level deep learning framework, or rather interface, that can be leveraged on top of MXNet as well as CNTK. Developed jointly by Amazon AWS and Microsoft, Gluon is very similar to Keras and can be thought of as a direct competitor. It claims, however, that it will support more low-level deep learning frameworks with time, with a vision of democratizing **artificial intelligence** (**AI**). Gluon provides a very simple, clean, and concise API that anyone can use to easily build deep learning architectures with minimal code.

- **BigDL**: Think of BigDL as deep learning on big data at scale! Developed by Intel, this framework can be leveraged on top of Apache Spark to build and run deep learning models as Spark programs in a distributed fashion over Hadoop clusters. It also leverages the very popular Intel **Math Kernel Library** (**MKL**) for efficient and improved performance.

The preceding list of frameworks is definitely not an exhaustive list of deep learning frameworks, but should give you a good idea of what is out there in the deep learning landscape. Feel free to explore these frameworks and choose any of them based on what suits you best.

Always remember that some of the frameworks have steep learning curves, so do not get disheartened if it takes time to learn and leverage them. While each of these frameworks has their own advantages and disadvantages, you should always focus more on the problem to be solved and then leverage the framework most suitable for solving the problem.

Setting up a cloud-based deep learning environment with GPU support

Deep learning works quite well on a standard single PC setup with a CPU. However, once your datasets start increasing in size and your model architectures start getting more complex, you need to start thinking about investing in a robust deep learning environment. The major expectations being the system can build and train models efficiently, take less time to train models, and is fault tolerant. Most deep learning computations are essentially millions of matrix operations (data is represented as matrices) and enable fast computation in parallel; GPUs have been proven to work really well in this aspect. You can consider setting up a robust cloud-based deep learning environment or even an in-house environment. Let's look at how we can set up a robust cloud-based deep learning environment in this section.

The major components involved are as follows:

- Choosing a cloud provider
- Setting up your virtual server
- Configuring your virtual server
- Installing and updating deep learning dependencies
- Accessing your deep learning cloud environment
- Validating GPU-enablement on your deep learning environment

Let's look at each of these components in further detail and take a step-by-step process that can help you to set up your own deep learning environment.

Choosing a cloud provider

There are multiple cloud providers with affordable and competitive rates these days. We are looking to leverage **Platform as a Service** (**PaaS**) capabilities where we just manage our data, applications, and basic configurations.

The following figure shows some popular cloud providers:
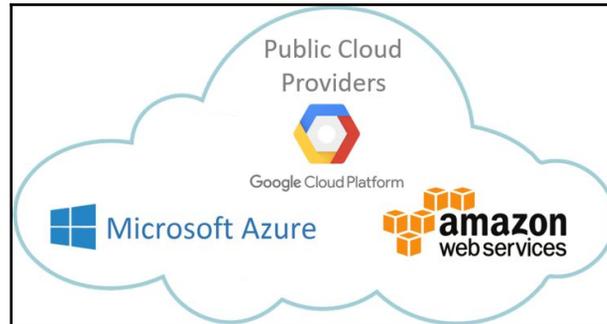
Popular providers include Amazon's AWS, Microsoft's Azure, and Google's **Google Cloud Platform** (**GCP**). For the purpose of this tutorial and for our book, we will be leveraging AWS.

Setting up your virtual server

You need to get an AWS account to follow the remaining steps in this section. Head over to `https://aws.amazon.com/` to create an account in case you do not have one already. Once ready, you can sign in to your account and navigate to the AWS EC2 control panel by going to `https://console.aws.amazon.com/ec2/v2/`, which leverages the **Elastic Compute Cloud** (**EC2**) service, the foundation of Amazon's cloud computing services. Once there, remember to select a region of your choice (I usually go with **US East**) and then click on **Launch Instance** to kickstart the process of creating a new virtual server on the cloud:
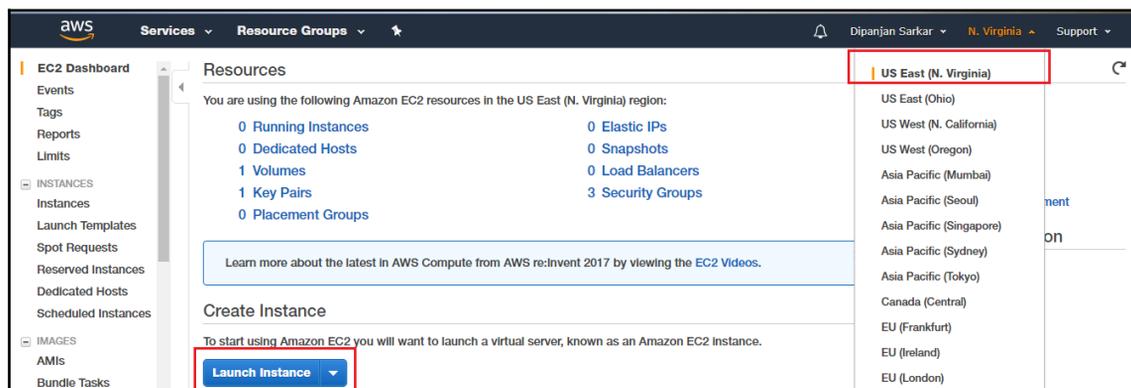
Clicking the **Launch Instance** button should take you to the page to select your own **Amazon Machine Image** (**AMI**). Typically, an AMI, consists of various software configurations necessary to build your virtual server. It includes the following:

- A template for the root volume for the instance, which includes your server's operating system, applications, and other configuration settings.
- Launch-permission settings that control which AWS accounts can use the AMI to launch instances.
- A block-device mapping that specifies the storage volumes to attach to the instance when it's launched. All your data goes here!

We will leverage a prebuilt AMI specialized for deep learning so we do not have to spend time with extra configuration and administration. Head over to the **AWS Marketplace** and select **Deep Learning AMI (Ubuntu)**:
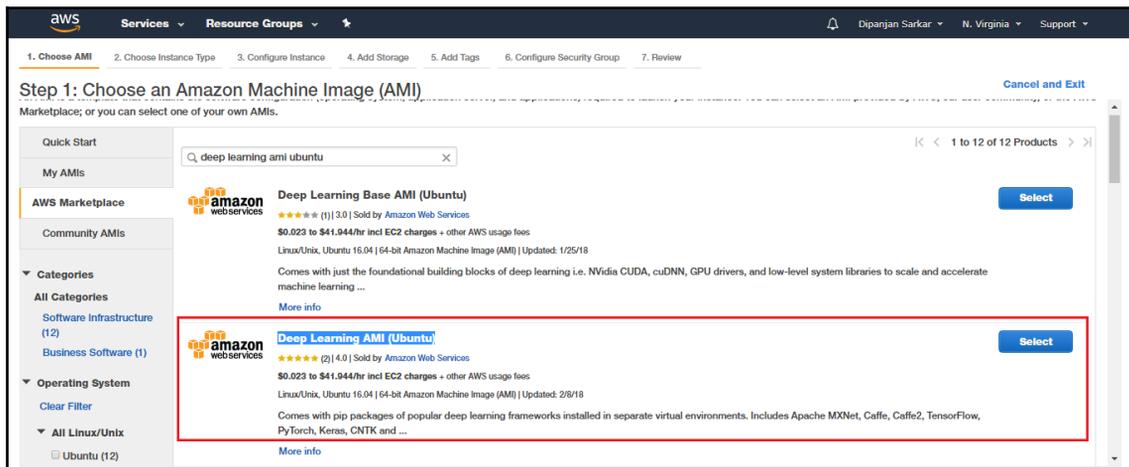
After choosing your AMI, you need to choose the instance type. For GPU-enabled deep learning, we recommend the **p2.xlarge** instance, which is powerful yet economical and costs around $0.90 per hour of use (as of 2018).

P2 instances provide up to 16 NVIDIA K80 GPUs, 64 vCPUs, and 732 GiB of host memory, with a combined 192 GB of GPU memory, as shown in the following screenshot:

Next up is to configure instance details. You can keep the default settings unless you wish to launch more than one instance, specify subnet preferences, and specify shutdown behavior.

The next step involves adding storage details. Typically, you have a root volume where you can increase its size if needed and add additional **Elastic Block Store** (**EBS**) volumes for additional disk space.

Then, we look at adding tags (case-sensitive and key-value pairs) if needed. We don't need this at the moment, so we skip it.

We focus more on the next step of configuring security groups, especially if you want to access your deep learning setup from outside by leveraging the ever-powerful Jupyter Notebooks. To enable this, we create a new security group and create a **Custom TCP** rule to open and enable access of port `8888`, as follows:

> A note of caution here, this rule typically allows any IP to listen to that port (`8888`) on your instance (which is where we will be running our Jupyter Notebooks). If needed, you can change this to just add the IP address of your specific PC or laptop for additional security. Besides this, we will add an additional password-protection feature to the Jupyter Notebooks later on for more security.

Finally, you will need to launch your instance by creating a key-pair (public and private keys) to connect to your instance securely. In case you don't have an existing key-pair, you can create a new one, safely store the private key file to your disk, and launch your instance, as depicted in the following screenshot:

Do note that it might take a few minutes before your virtual server boots up and launches, so you might need to wait a bit. Often, you might see that your instance has failed to launch owing to insufficient limits or capacity for your account.

In case you face this issue, you can request a limit increase for the specific instance type you are using (in our case, `p2.xlarge`):

Usually, AWS responds and approves your request in less than 24 hours, so you might need to wait a bit before you get the approval and then you should be able to launch your instance. Once you launch your instance, you can check out the **Instances** section and try to connect to the instance:
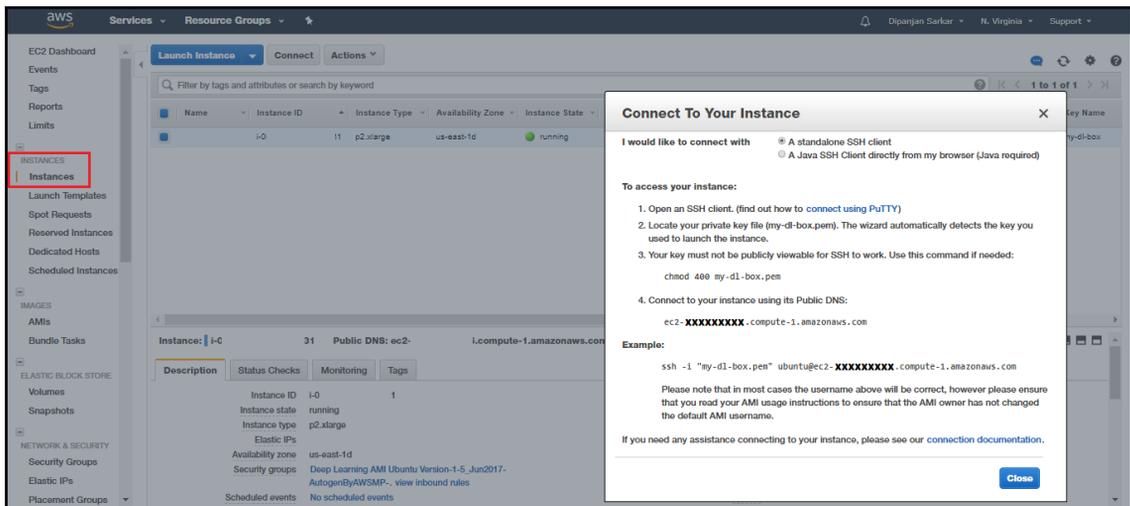
You can make use of a Command Prompt or Terminal from your local system, where you have stored the private AWS key from earlier, to connect to your instance now:

```
[DIP.DipsLaptop]> ssh -i "my-dl-box.pem" ubuntu@ec2-
xxxxx.compute-1.amazonaws.com
Warning: Permanently added 'ec2-xxxxx.compute-1.amazonaws.com' (RSA) to the
list of known hosts.


=========================================================================
Deep Learning AMI for Ubuntu
=========================================================================
```

```
The README file for the AMI : /home/ubuntu/src/AMI.README.md
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-121-generic x86_64)

Last login: Sun Nov 26 09:46:05 2017 from 10x.xx.xx.xxx
ubuntu@ip-xxx-xx-xx-xxx:~$
```

Thus this allows you to successfully log in to your own cloud-based deep learning server!

Configuring your virtual server

Let's set up some basic configurations to leverage the power of Jupyter Notebooks to do our analyses and deep learning modeling on our virtual server without having to code on the Terminal all the time. First, we need to set up SSL certificates. Let's create a new directory:

```
ubuntu@ip:~$ mkdir ssl
ubuntu@ip:~$ cd ssl
ubuntu@ip:~/ssl$
```

Once we are in the directory, we leverage OpenSSL to create a new SSL certificate:

```
ubuntu@ip:~/ssl$ sudo openssl req -x509 -nodes -days 365 -newkey rsa:1024 -
keyout "cert.key" -out "cert.pem" -batch
Generating a 1024 bit RSA private key
......+++++
...+++++
writing new private key to 'cert.key'
-----
ubuntu@ip:~/ssl2$ ls
cert.key cert.pem
```

Now, we need to add that additional layer of password-based security to our Jupyter Notebooks, which we talked about earlier. For this, we need to modify Jupyter's default configuration settings. In case you do not have a `config` file for Jupyter, you can use the following command to generate it:

```
$ jupyter notebook --generate-config
```

To enable password-based security for the notebooks, we need to first generate a password and its hash. We can leverage the `passwd()` function in `Ipython.lib` as follows:

```
ubuntu@ip:~$ ipython
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from IPython.lib import passwd
```

```
In [2]: passwd()
Enter password:
Verify password:
Out[2]: 'sha1:e9ed12b73a30:142dff0cdcaf375e4380999a6ca17b47ce187eb6'
In [3]: exit
ubuntu@:~$
```

Once you enter your password and verify it, the function returns a hash to you, which is the hash of your password (in this case, the password key I typed was literally the word password, which is something you should definitely not be using!). Copy and save that hash value since we will need it soon.

Next, fire up your favorite text editor to edit the Jupyter config file, as follows:

```
ubuntu@ip:~$ vim ~/.jupyter/jupyter_notebook_config.py


# Configuration file for jupyter-notebook.

c = get_config()  # this is the config object
c.NotebookApp.certfile = u'/home/ubuntu/ssl/cert.pem'
c.NotebookApp.keyfile = u'/home/ubuntu/ssl/cert.key'
c.IPKernelApp.pylab = 'inline'
c.NotebookApp.ip = '*'
c.NotebookApp.open_browser = False
c.NotebookApp.password =
'sha1:e9ed12b73a30:142dff0cdcaf375e4380999a6ca17b47ce187eb6' # replace this

# press i to insert new text and then press 'esc' and :wq to save and exit

ubuntu@ip:~$
```

We will now look at some essential dependencies for enabling deep learning before we can start building our models.

Installing and updating deep learning dependencies

There are several major aspects of deep learning and leveraging GPU-enabled deep learning for Python. We will try to cover the essentials, but feel free to refer to other online documentation and resources as needed. You can also skip these steps and head over to the next section to test whether GPU-enabled deep learning is already active on your server. The newer AWS deep learning AMIs have GPU-enabled deep learning set up.

However, often the setup is not the best or some of the configurations might be wrong, so if you see that deep learning is not making use of your GPU, (from the tests in the next section), you might need to go over these steps. You can head over to the *Accessing your deep learning cloud environment* and *Validating GPU-enablement* sections on your deep learning environment to check whether the default setup provided by Amazon works. Then you don't need to go to the trouble of following the remaining steps!

First, you need to check whether the Nvidia GPU is enabled and the drivers for the GPU are properly installed. You can leverage the following commands to check this. Remember, p2.x typically comes armed with a Tesla GPU:

```
ubuntu@ip:~$ sudo lshw –businfo | grep –i display
pci@0000:00:02.0 display GD 5446
pci@0000:00:1e.0 display GK210GL [Tesla K80]

ubuntu@ip–172–31–90–228:~$ nvidia-smi
```

If the drivers are properly installed, you should see an output similar to the following snapshot:

```
Sun Feb 25 15:18:59 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.66                 Driver Version: 384.66                     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           On   | 00000000:00:1E.0 Off |                    0 |
| N/A   40C    P8    31W / 149W |      0MiB / 11439MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

If you get an error, follow these steps to install the Nvidia GPU drivers. Remember to use a different driver link, based on the OS you are using. I have an older Ubuntu 14.04 AMI, for which I used the following:

```
# check your OS release using the following command
ubuntu@ip:~$ lsb_release –a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 14.04.5 LTS
Release: 14.04
```

```
Codename: trusty

# download and install drivers based on your OS
ubuntu@ip:~$
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1404/
x86_64/cuda-repo-ubuntu1404_8.0.61-1_amd64.deb
ubuntu@ip:~$ sudo dpkg -i ./cuda-repo-ubuntu1404_8.0.61-1_amd64.deb
ubuntu@ip:~$ sudo apt-get update
ubuntu@ip:~$ sudo apt-get install cuda -y
# Might need to restart your server once
# Then check if GPU drivers are working using the following command
ubuntu@ip:~$ nvidia-smi
```

If you are able to see the driver and GPU hardware details based on the previous commands, your drivers have been installed successfully! Now you can focus on installing the Nvidia CUDA toolkit. The CUDA toolkit in general provides us with a development environment for creating high-performance GPU-accelerated applications. This is what is used to optimize and leverage the full power of our GPU hardware. You can find out more about CUDA and download toolkits at `https://developer.nvidia.com/cuda-toolkit`.

> Remember that CUDA is very version-specific, and different versions of our Python deep learning frameworks are compatible with only specific CUDA versions. I will use CUDA 8 in this chapter. Please skip this step if CUDA is already installed for you and is working properly with your deep learning ecosystem on the server.

To install CUDA, run the following commands:

```
ubuntu@ip:~$ wget
https://s3.amazonaws.com/personal-waf/cuda_8.0.61_375.26_linux.run

ubuntu@ip:~$ sudo rm -rf /usr/local/cuda*

ubuntu@ip:~$ sudo sh cuda_8.0.61_375.26_linux.run
# press and hold s to skip agreement and also make sure to select N when
asked if you want to install Nvidia drivers

# Do you accept the previously read EULA?
# accept

# Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 361.62?
# *********************** VERY KEY ***************************
# ******************* DON"T SAY Y ***************************
# n

# Install the CUDA 8.0 Toolkit?
# y
```

```
# Enter Toolkit Location
# press enter

# Do you want to install a symbolic link at /usr/local/cuda?
# y

# Install the CUDA 8.0 Samples?
# y

# Enter CUDA Samples Location
# press enter

# Installing the CUDA Toolkit in /usr/local/cuda-8.0 ...
# Installing the CUDA Samples in /home/liping ...
# Copying samples to /home/liping/NVIDIA_CUDA-8.0_Samples now...
# Finished copying samples.
```

Once CUDA is installed, we also need to install cuDNN. This framework has also been developed by Nvidia and stands for the **CUDA Deep Neural Network** (**cuDNN**) library. Essentially, this library is a GPU-accelerated library that consists of several optimized primitives for deep learning and building deep neural networks. The cuDNN framework provides highly optimized and tuned implementations for standard deep learning operations and layers, including regular activation layers, convolution and pooling layers, normalization, and backpropagation! The purpose of this framework is to accelerate the training and performance of deep learning models, specifically for Nvidia GPUs. You can find out more about cuDNN at `https://developer.nvidia.com/cudnn`. Let's install cuDNN using the following commands:

```
ubuntu@ip:~$ wget https://s3.amazonaws.com/personal-waf/cudnn-8.0-
             linux-x64-v5.1.tgz

ubuntu@ip:~$ sudo tar -xzvf cudnn-8.0-linux-x64-v5.1.tgz
ubuntu@ip:~$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
ubuntu@ip:~$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
ubuntu@ip:~$ sudo chmod a+r /usr/local/cuda/include/cudnn.h
             /usr/local/cuda/lib64/libcudnn*
```

Once complete, remember to add the following lines to the end of `~/.bashrc` using your favorite editor (we used `vim`):

```
ubuntu@ip:~$ vim ~/.bashrc

# add these lines right at the end and press esc and :wq to save and
# quit

export
LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/usr/local/cuda
```

```
                    /extras/CUPTI/lib64"
export CUDA_HOME=/usr/local/cuda
export DYLD_LIBRARY_PATH="$DYLD_LIBRARY_PATH:$CUDA_HOME/lib"
export PATH="$CUDA_HOME/bin:$PATH"

ubuntu@ip:~$ source ~/.bashrc
```

Typically, this takes care of most of the necessary dependencies for our GPU. Now, we need to install and set up our Python deep learning dependencies. Usually AWS AMIs come installed with the Anaconda distribution; however, in case it is not there, you can always refer to https://www.anaconda.com/download to download the distribution of your choice, based on the Python and OS version. Typically, we use Linux/Windows and Python 3 and leverage the TensorFlow and Keras deep learning frameworks in this book. In the AWS AMIs, incompatible framework versions might be installed, which don't work well with CUDA or might be CPU-only versions. The following commands install the GPU version of TensorFlow that works best on CUDA 8:

```
# uninstall previously installed versions if any
ubuntu@ip:~$ sudo pip3 uninstall tensorflow
ubuntu@ip:~$ sudo pip3 uninstall tensorflow-gpu

# install tensorflow GPU version
ubuntu@ip:~$ sudo pip3 install --ignore-installed --upgrade
https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-1.2.0-cp
34-cp34m-linux_x86_64.whl
```

Next up, we need to upgrade Keras to the latest version and also delete any leftover `config` files:

```
ubuntu@ip:~$ sudo pip install keras --upgrade
ubuntu@ip:~$ sudo pip3 install keras --upgrade
ubuntu@ip:~$ rm ~/.keras/keras.json
```

We are now almost ready to start leveraging our deep learning setup on the cloud. Hold on tight!

Accessing your deep learning cloud environment

We don't really want to sit and code in the Terminal on the server all the time. Since we want to leverage Jupyter Notebooks for interactive development, we will access the Notebooks on our cloud server from our local system. For this, we first need to kickstart our Jupyter Notebook server on our remote instance.

Log in to your virtual server and start the Jupyter Notebook server:

```
[DIP.DipsLaptop]> ssh -i my-dl-box.pem ubuntu@ec2-
xxxxx.compute-1.amazonaws.com
==================================
Deep Learning AMI for Ubuntu
==================================
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-121-generic x86_64)
Last login: Sun Feb 25 18:23:47 2018 from 10x.xx.xx.xxx

# navigate to a directory where you want to store your jupyter notebooks
ubuntu@ip:~$ cd notebooks/
ubuntu@ip:~/notebooks$ jupyter notebook
[I 19:50:13.372 NotebookApp] Writing notebook server cookie secret to
/run/user/1000/jupyter/notebook_cookie_secret
[I 19:50:13.757 NotebookApp] Serving notebooks from local directory:
/home/ubuntu/notebooks
[I 19:50:13.757 NotebookApp] 0 active kernels
[I 19:50:13.757 NotebookApp] The Jupyter Notebook is running at:
https://[all ip addresses on your system]:8888/
[I 19:50:13.757 NotebookApp] Use Control-C to stop this server and shut
down all kernels (twice to skip confirmation).
```

We now need to enable port-forwarding on our local instance for accessing our server Notebooks from the browser of our local machine. Leverage the following syntax:

```
sudo ssh -i my-dl-box.pem -N -f -L
local_machine:local_port:remote_machine:remote_port ubuntu@ec2-
xxxxx.compute-1.amazonaws.com
```

This will start forwarding your local machine's port (8890, in my case) to port 8888 of the remote virtual server. The following is what I used for my setup:

```
[DIP.DipsLaptop]> ssh -i "my-dl-box.pem" -N -f -L
localhost:8890:localhost:8888
ubuntu@ec2-52-90-91-166.compute-1.amazonaws.com
```

This is also known as **SSH tunneling**. Thus, once you start forwarding, head over to your local browser and navigate to the localhost address, https://localhost:8890, which we are forwarding to the remote Notebook server in our virtual server. Make sure that you use https in the address, otherwise you'll get an SSL error.

If you have done everything correctly so far, you should see a warning screen in your browser, and if you follow the steps depicted in the following screenshot, you should see the familiar Jupyter user interface that comes up whenever you work on any Notebooks:
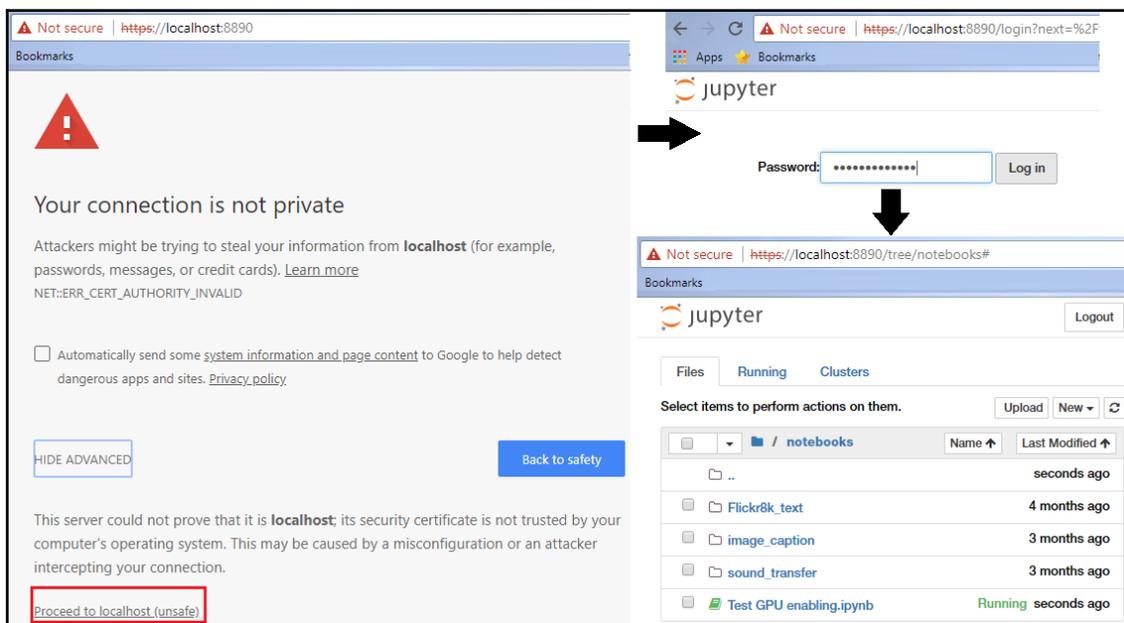
You can safely ignore the **Your connection is not private** warning; it shows up because we generated the SSL certificate ourselves and it hasn't been verified by any trusted authority.

Validating GPU-enablement on your deep learning environment

The final step is to make sure everything is working and that our deep learning frameworks are leveraging our GPU (for which we are paying by the hour!). You can refer to the `Test GPU enabling.ipynb` Jupyter Notebook for all the code used to test this out. We will cover it in detail here. The first thing we will validate is whether `keras` and `tensorflow` are loaded properly in our server. This can be validated by importing them as follows:

```
import keras
import tensorflow

Using TensorFlow backend.
```

If you see the preceding code load up without errors, that is excellent! Otherwise, you might want to retrace the steps you followed earlier and search online for the specific error you are getting; check out the GitHub repositories for each of the frameworks.

The final step is to check whether `tensorflow` is enabled to use our server's Nvidia GPU. You can validate this using the following test:

```
In [1]: from tensorflow.python.client import device_lib
   ...: device_lib.list_local_devices()

Out [1]:
[name: "/cpu:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality {
 }
 incarnation: 9997170954542835749,

 name: "/gpu:0"
 device_type: "GPU"
 memory_limit: 11324823962
 locality {
   bus_id: 1
 }
 incarnation: 10223482989865452371
 physical_device_desc: "device: 0, name: Tesla K80, pci bus id:
0000:00:1e.0"]
```

If you observe the preceding output, you can see our GPU is listed in the list of devices and hence it will leverage the same during training our deep learning models. You have successfully set up a robust deep learning environment on the cloud, which you can now use to train deep learning models faster, using GPUs!

> **TIP**
>
> Always remember that AWS charges instances by the hour and you do not want to keep your instances running once you have finished your analyses and building your models. You can always restart your instance from the EC2 console, as needed.

Setting up a robust, on-premise deep learning environment with GPU support

Often users or organizations may not want to leverage cloud services, especially if their data is sensitive, and so focus on building an on-premise deep learning environment. The major focus here should be to invest in the right type of hardware to enable maximum performance and leverage the right GPU for building deep learning models. With regards to hardware, special emphasis goes to the following:

- **Processor**: You can invest in an i5 or an i7 Intel CPU, or maybe an Intel Xeon if you are looking to spoil yourself!
- **RAM**: Invest in at least 32 GB of DDR4 or better RAM for your memory.
- **Disk**: A 1 TB hard disk is excellent, and also you can invest in a minimum of 128 GB or 256 GB of SSD for fast data access!
- **GPU**: Perhaps the most important component for deep learning. Invest in a NVIDIA GPU, anything above a GTX 1070 with 8 GB.

Other things you shouldn't neglect include a motherboard, power supply, robust case, and cooler.

Once you get your rig set up, for the software configuration, you can repeat all the steps from the previous section, excluding the cloud setup, and you should be good to go!

Neural network basics

Let's try to familiarize ourselves with some of the basic concepts behind neural networks that make all deep learning models tick!

A simple linear neuron

A linear neuron is the most basic building block of a deep neural network. It can be schematically represented as shown in the following figure. Here, $\vec{X} = (x_1, x_2, \ldots, x_n)$ represents the input vector and $w_i s$ are the weights of the neuron. Given a training set consisting a of set of input, target value pairs, a linear neuron tries to learn a linear transformation that can map the input vectors to the corresponding target value. Basically, a linear neuron approximates the input output relationship by a linear function $\mathrm{W}^T \vec{X} = \vec{y}$:
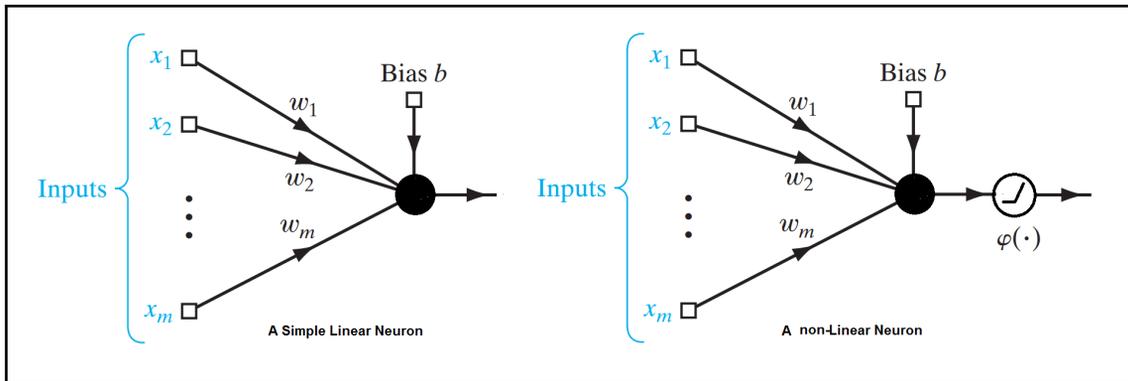
Schematic representation of a simple linear neuron and a simple non-linear neuron

Lets try to model a toy problem with this simple neuron. Employee *A* buys lunch from the cafeteria. Their diet consists of fish, chips, and ketchup. They get several portions of each. The cashier only tells them the total price of the meal. After several days, can they figure out price of each portion?

Well, this sounds like a simple linear programming problem, which can be easily solved analytically. Let's represent this problem using the preceding linear neural unit. Here, $\vec{X} = (x_{fish}, x_{ketchup}, x_{chips})$ and we have the corresponding weights $(w_{fish}, w_{ketchup}, w_{chips})$.

Each meal price gives a linear constraint on the prices of the portions:

$$Price = w_{fish} * x_{fish} + w_{ketchup} * x_{ketchup} + w_{chips} * x_{chips}$$

Let t_n be the true price and y_n the estimated price by our model, which is given by the preceding linear equation. The residual price difference between the target and our estimates is $t_n - y_n$. Now, these residuals for different meals can be positive or negative, and may cancel out, giving zero overall errors. One way to handle this is to use sum squared residuals, $E = \frac{1}{2}\sum_n (t_n - y_n)^2$. If we are able to minimize this error, we can possibly get a good estimate of a set of weights/price per item. So, we have arrived at an optimization problem. Let's first discuss some of the methods to solve an optimization problem.

Gradient-based optimization

Optimization basically involves either minimizing or maximizing some function, *f(x)*, where *x* is a numerical vector or a scalar. Here, *f(x)* is called the **objective function** or **criterion**. In the context of neural networks, we call it the cost function, loss function, or error function. In the previous example, the loss function we want to minimize is *E*.

Suppose we have a function, *y = f(x)*, where *x* and *y* are real numbers. The derivative of this function tells us how this function changes with small changes to *x*. So, the derivative can be used to reduce the value of the function by infinitesimally changing *x*. Suppose $f'(x) > 0$ at *x*. This means *f(x)* will increase if we increase *x* along positive *x* and hence *f(x - ε) < f(x)* for small enough ε. Note *f(x)* can be reduced by moving *x* in small steps within the *opposite* direction of the derivative:
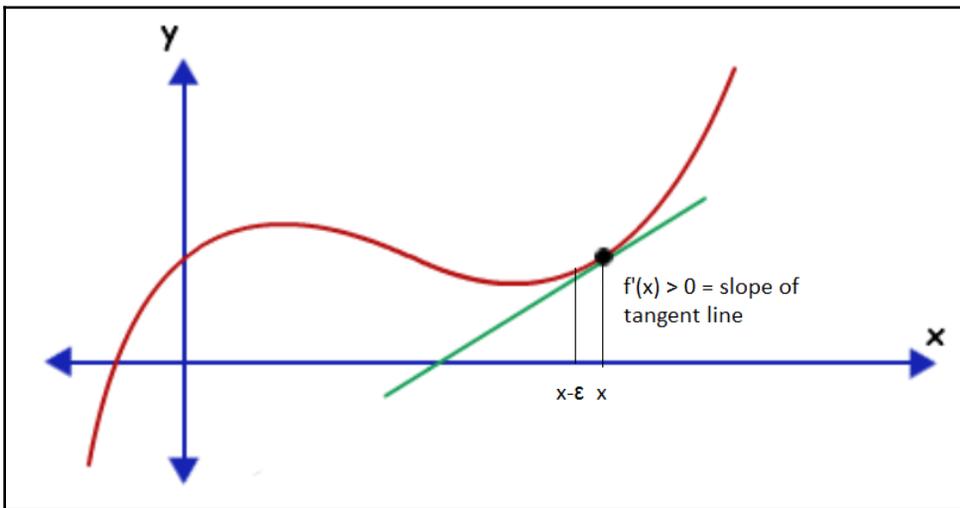
How the function value changes by moving along or in opposite direction of derivative

If the derivative $f'(x) = 0$, then the derivative provides no information in which direction we need to move to reach the function minimum. The derivative can be zero at a local optimum (minimum/maximum). A point, *x**, is called **local minimum** if the value of the function, *f(x)*, at *x** is less than all neighboring points of *x**. Similarly, we can define a local maximum. Some points can be neither maximum nor minimum, but the derivative $f'(x)$ is zero at those points. These are called **saddle** points. The following figure explains the three scenarios where $f'(x) = 0$:
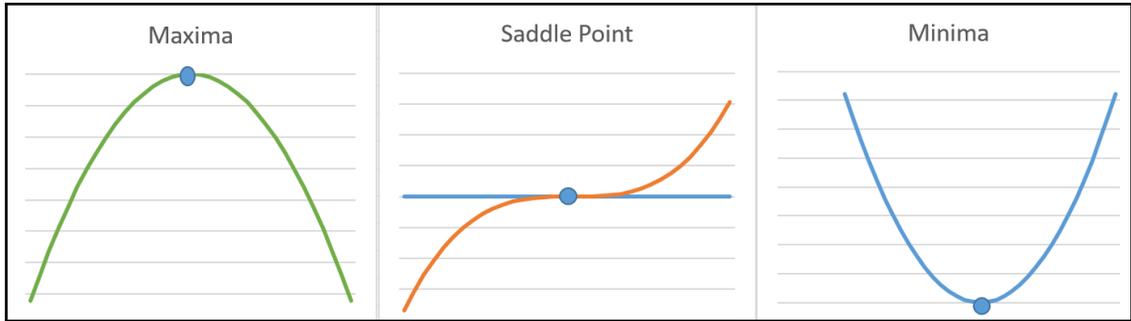
Min, max, and saddle point for a function of single variable. Derivative f'(x)=0 at all three highlighted points

A point that attains lowest value of *f* among all possible values of *x* is called a **global minimum**. A function can have one or many global minima. There could be local minima that are not a global minimum. However, if the function is convex, it's guaranteed that it has only one global minimum and no local minima.

Generally in ML, we are interested in minimizing a real valued function of several variables f: $\mathbb{R}^n \to \mathbb{R}$. A simple example of a real valued function of several variables is the hotplate temperature function, $f(x_1, x_2) = 50 - x_1^2 - 2x_2^2$ at a point with coordinates $\vec{x} = (x_1, x_2)$ on the plate. In deep learning, we generally minimize a loss function, which is a function of several variables, such as the weights in the neural network. These functions have many local minima, many saddle points that are surrounded by very flat regions, and they may or may not have any global minima. All this makes it very difficult to optimize such functions.

The derivatives of a function of several variables are expressed as partial derivatives, which measure the rate of change of the function as we change one of the input variables, x_i, keeping all other input variables constant. The vector of partial derivatives with regards to all the variables is called the **gradient vector** of *f* and is denoted by ∇f. We can also find out how fast the function might be changing with respect to some arbitrary direction, $\vec{v}$ (a unit vector). This is computed by projecting the gradient vector, ∇f, in the direction of the unit vector, $\vec{v}$, that is, the dot product, $\nabla f \bullet \vec{v}$. This is called the **directional derivative** of *f* in the direction of $\vec{v}$ and is often denoted by $\nabla_{\vec{v}}$. To minimize *f*, we need to find a direction, $\vec{u}$, in which to change $\vec{x}$, such that the value of *f* is reduced maximally.

Let $\overrightarrow{x_a}$ be a point very close to $\vec{x}$, that is, $\left\| \vec{x} - \overrightarrow{x_a} \right\|$ is very small. First, the order expansion of the Taylor series around $\vec{x}$ is given by:

$$ f\left(\overrightarrow{x_a} \right) = f\left(\vec{x} \right) + \nabla f(\vec{x}) \bullet \left(\overrightarrow{x_a} - \vec{x} \right) + o\!\left(\left\| \overrightarrow{x_a} - \vec{x} \right\| \right) $$

The last term in the preceding equation is negligible for $\overrightarrow{x_a}$ to be close enough to $\vec{x}$. The second term represents the directional derivative of f along $\overrightarrow{x_a} - \vec{x}$. Here we have:

$$ \nabla f(\vec{x}) \bullet \left(\overrightarrow{x_a} - \vec{x} \right) = \left\| \nabla f(\vec{x}) \right\| \left\| \overrightarrow{x_a} - \vec{x} \right\| cos(\theta) $$

Therefore, $f(\vec{x})$ is maximally reduced if cos (θ) is minimum, that is, -1, which will happen if $\theta = \pi$, that is, $\overrightarrow{x_a} - \vec{x}$ should point in a direction opposite to the gradient vector, ∇f. This is the direction of steepest descent: $-\nabla f$ or the direction of **steepest gradient descent**. We explain this in the following figure:
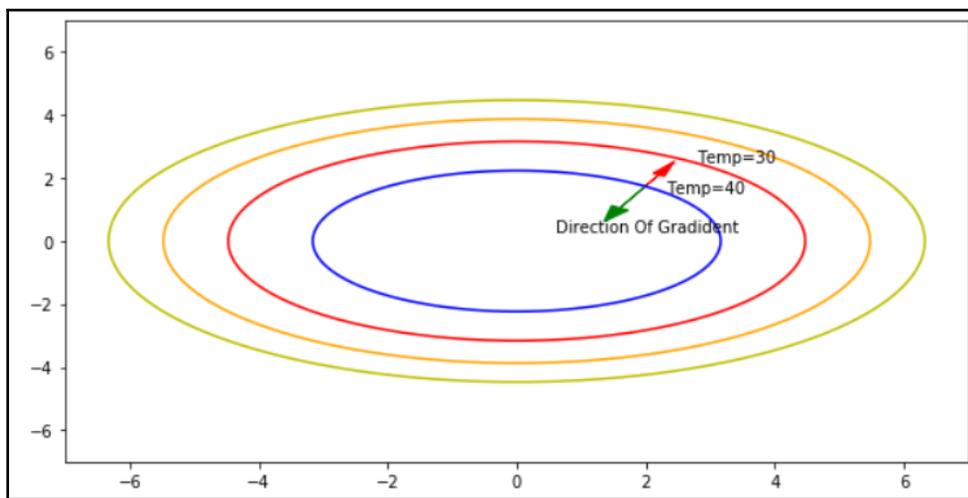
Hot plate

The *Hot plate* example: the temperature at a given coordinate (x and y) is given by the function $f(x, y) = 50 - y^2 - 2x^2$. The plate is hottest at the center (0, 0), with a temperature of 50. The gradient vector at a point (x, y) is given by $\nabla f = (-4x, -2y)$. The temperature at point (2.3, 2) on the plate is **40**. This point lies on a contour of constant temperature. Clearly, moving in a direction opposite to the gradient, as shown by the red arrow, with step size ε the temperature reduces to **30**.

Let's implement gradient-descent optimization for the hot plate temperature function using `tensorflow`. We need to initialize the gradient descent, so let's start at $x = y = 2$:

```
import tensorflow as tf
#Initialize Gradient Descent at x,y =(2, 2)
x = tf.Variable(2, name='x', dtype=tf.float32)
y = tf.Variable(2, name='y', dtype=tf.float32)

temperature = 50 - tf.square(y) - 2*tf.square(x)

#Initialize Gradient Descent Optimizer
optimizer = tf.train.GradientDescentOptimizer(0.1) #0.1 is the learning
rate
train = optimizer.minimize(temperature)
grad = tf.gradients(temperature, [x,y]) #Lets calculate the gradient vector

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print("Starting at coordinate x={}, y={} and temperature there is
            {}".format(
                    session.run(x),session.run(y),session.run(temperature)))
    grad_norms = []
    for step in range(10):
        session.run(train)
        g = session.run(grad)
        print("step ({}) x={},y={}, T={}, Gradient={}".format(step,
                    session.run(x), session.run(y),
session.run(temperature), g))
        grad_norms.append(np.linalg.norm(g))

plt.plot(grad_norms)
```

The following is the output of the preceding code. In each step, new values of `x`, `y` are calculated as suggested by the gradient vector such that the overall temperature decreases maximally. Note that the gradient calculated exactly matches the formula stated previously. We are also calculating the gradient norm after each step. Here is how the gradient changes in 10 iterations:

```
Starting at coordinate x=2.0, y=2.0 and temperature there is 38.0
step (0)  x=2.79,y=2.40000, T=28.55, Gradient=[-11.2, -4.8000002]
step (1)  x=3.92,y=2.88000, T=10.97, Gradient=[-15.68, -5.7600002]
..........
step (9)  x=57.85,y=12.38347, T=-6796.81, Gradient=[-231.40375, -24.766947]
```

The Jacobian and Hessian matrices

Sometimes, we need to optimize functions whose input and output are vectors. So, for each component of the output vector, we need to compute the gradient vector. For $f : \mathbb{R}^n \to \mathbb{R}^m$, we will have m gradient vectors. By arranging them in a matrix form, we get n x m matrix of partial derivatives $J_{i,j} = \dfrac{\partial f(\vec{x})_i}{\partial x_j}$, called the **Jacobian matrix**.

For a real-valued function of a single variable, if we want to measure the curvature of the function curve at a point, then we need to compute how first the derivative will change as we change the input. This is called the **second-order derivative**. A function whose second-order derivative is zero has no curvature and is a flat line. Now, for the function of several variables, there are many second derivatives. These derivatives can be arranged in a matrix called the **Hessian matrix**. Since the second-order partial derivatives are symmetric, that is, $\dfrac{\partial^2 f(\vec{x})}{\partial x_i \partial x_j} = \dfrac{\partial^2 f(\vec{x})}{\partial x_j \partial x_i}$, the Hessian matrix is real symmetric and thus has real eigen values. The corresponding eigen vectors represent different directions of curvature. The ratio of the magnitude of the largest and smallest eigen value is called the **condition number** of the Hessian. It measures how much the curvatures along each eigen dimension differ from each other. When the Hessian has a poor condition number, the gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change and, due to this, may take a very long time to converge.

For our temperature example, the Hessian is $\begin{bmatrix} -2 & 0 \\ 0 & -4 \end{bmatrix}$. The direction of most curvature is two times more than the direction of least curvature. So, traversing along *y*, we will reach the minimum point much faster. This is also evident from the temperature contours, shown in the preceding *Hot plate* figure.

We can use the second-derivative curvature information to check whether an optimal point is minimum or maximum. For a single variable, *f′(x) = 0, f′′(x) > 0* implies *x* is a local minimum of *f*, and *f′ (x) = 0, f′′ (x) < 0* implies *x* is a local maximum. This is called a **second derivative test** (refer to the following figure *Explaining curvature*). Similarly, for functions of several variables, if the Hessian is positive definite (that is, all eigen values are positive) at $\vec{x}$, then *f* attains a local minimum at $\vec{x}$. If the Hessian is negative definite at *x*, then *f* attains local maximum at *x*. If the Hessian has both positive and negative eigen values, then *x* is a saddle point for *f*. Otherwise, the test is inconclusive:
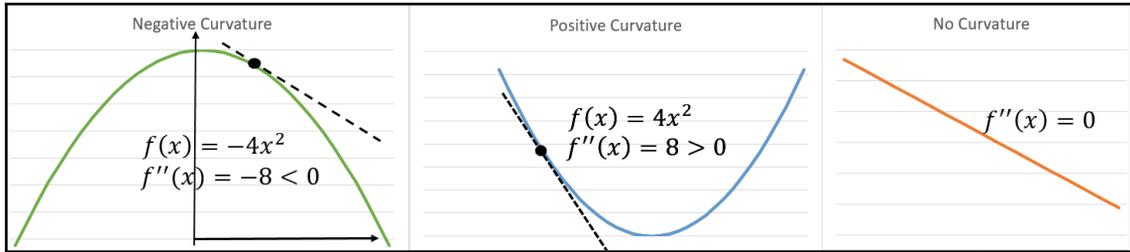
Explaining curvature

There are optimization algorithms based on second-order derivatives that use the curvature information. One such method is Newton's method, which can reach the optimum point in just one step for convex functions.

Chain rule of derivatives

Let f and g both be real-valued functions of a single variable. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$.

Then, the chain rule of derivatives states that:

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}$$

Similarly for the function of several variables, let $\vec{x} \in \mathbb{R}^m, \vec{y} \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n, f : \mathbb{R}^n \to \mathbb{R}, \vec{y} = g(\vec{x}), z = f(\vec{y})$, then, $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$

Therefore, the gradient of z with respect to $\vec{x}$, $\nabla_x(z) = \frac{\partial z}{\partial \vec{x}}$ is represented as a multiplication of the Jacobian $\frac{\partial \vec{y}}{\partial \vec{x}}$ with the $\nabla_y(z) = \frac{\partial z}{\partial \vec{y}}$ gradient vector. So, we have the chain rule of derivatives for a function of several variables as follows:

$$\nabla_x(z) = \frac{\partial \vec{y}}{\partial \vec{x}}\nabla_y(z)$$

The neural network learning algorithm consists of several such Jacobian gradient multiplications.

Stochastic Gradient Descent

Almost all neural network learning is powered by one very important algorithm: SGD. This is an extension of the normal gradient-descent algorithm. In ML, the loss function is often written as a sum over per-example loss function as the squared error *E* in the *cafeteria example*. Thus, if we have *m* training examples, the gradient function will also have *m* additive terms.

The computational cost of the gradient increases linearly with *m*. For a billion-size training set, the preceding gradient computation will take very long time and the gradient-descent algorithm will proceed very slowly toward convergence, making learning impossible in practice.

SGD depends on a simple insight that the gradient is actually an expectation. We can approximate the expectation by computing it on small sample sets. A **mini-batch** of *m'*(much smaller than *m*) sample size can be drawn uniformly at random from the training set and the gradient can be approximated to compute a single gradient descent step. Let's consider our *cafeteria example* again. Applying the chain rule, the gradient of the error function (a function of three variables) is given by:

$$\nabla E_i = \frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{\partial E^n}{\partial y^n} = \frac{1}{2} \sum_n -2x_i^n(t^n - y^n) = -\sum_n x_i^n(t^n - y^n)$$

Now, instead of using all *n* training example to calculate the derivative, if we take a small random sample of the training example, we can still reasonably approximate the derivative.

The ∇E gradient gives an estimate of the weight update. We can control it further by multiplying it by a constant, ε, called the **learning rate**. Taking a very high learning rate may increase of the optimization-objective function value instead of minimizing it.

In SGD, weights are updated after each mini-batch is presented to the algorithm. It takes a number of data points/batch-size steps to present the entire training data once to the training algorithm. An epoch describes the number of times the algorithm sees the *entire* dataset.

The following is the `keras` code for the cafeteria problem. Let the actual prices for fish be 150 cents, chips 50 cents, and ketchup 100 cents. We have generated sample portions of items in the meal randomly. Let the initial guess for the set of weight/price be 50 cents per portion. After 30 epochs, we get the very close estimates to the true price of the items:

```
#The true prices used by the cashier
p_fish = 150; p_chips = 50; p_ketchup = 100
```

```
#sample meal prices: generate data meal prices for 10 days.
np.random.seed(100)
portions = np.random.randint(low=1, high=10, size=3 )
X = []; y = []; days = 10
for i in range(days):
    portions = np.random.randint(low=1, high=10, size=3 )
    price = p_fish * portions[0] + p_chips * portions[1] + p_ketchup *
    portions[2]
    X.append(portions)
    y.append(price)
X = np.array(X)
y = np.array(y)

#Create a linear model
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import SGD

price_guess = [np.array([[ 50 ], [ 50],[ 50 ]]) ] #initial guess of the
price
model_input = Input(shape=(3,), dtype='float32')
model_output = Dense(1, activation='linear', use_bias=False,
    name='LinearNeuron',
    weights=price_guess)(model_input)
    sgd = SGD(lr=0.01)
model = Model(model_input, model_output)

#define the squared error loss E stochastic gradient descent (SGD)

optimizer
model.compile(loss="mean_squared_error", optimizer=sgd)
model.summary()
_____
Layer (type) Output Shape Param #
================================================================
input_4 (InputLayer) (None, 3) 0
_____
LinearNeuron (Dense) (None, 1) 3
================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_____

#train model by iterative optimization: SGD with mini-batch of size 5.

history = model.fit(X, y, batch_size=5, epochs=30,verbose=2)
```

In the following figure, we show the effect of the learning rate on convergence of the iterative SGD algorithm:
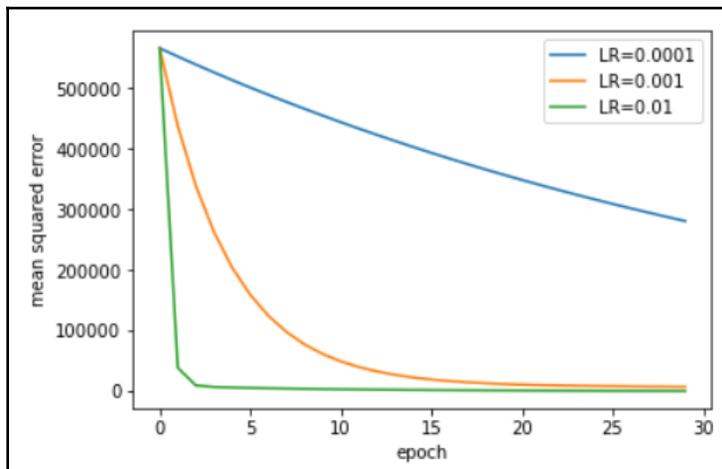
Effect of learning rate on the rate of convergence of SGD for the cafeteria problem

The following table shows how the price guesses are updated by SGD over successive epochs for LR = 0.01:

Epoch	w_fish	w_chips	w_ketchup
0(initial)	50	50	50
1	124.5	96.3	127.4
5	120.6	81.7	107.48
10	128.4	74.7	104.6
15	133.8	68.9	103.18
30	143.07	58.2	101.3
50	148.1	52.6	100.4

Non-linear neural units

A linear neuron is simple but computationally limited. Even if we use a deep stack of multiple layers of linear units, we still have a linear network capable of learning only linear transformations. To design networks that can learn much richer sets of transformations (non-linear), we need a way to introduce non-linearity in the design of neural nets. By passing the linear weighted sum of input through a non-linear function, we can induce non-linearity in the neural unit.

Although the non-linear function is fixed, it can adapt to the data through the weights of the linear unit, which are arguments to this function. This non-linear function is called an **activation function** of a non-linear neuron. One simple activation function example is binary threshold activation, and the corresponding non-linear unit is called **McCulloch-Pitts unit**. This is a step function and it's not differentiable at zero. Also, at non-zero points, its derivative is zero. Other, popularly used activation functions are sigmoid, tanh, and ReLu. The definitions and plots of these functions are provided in the following figure:
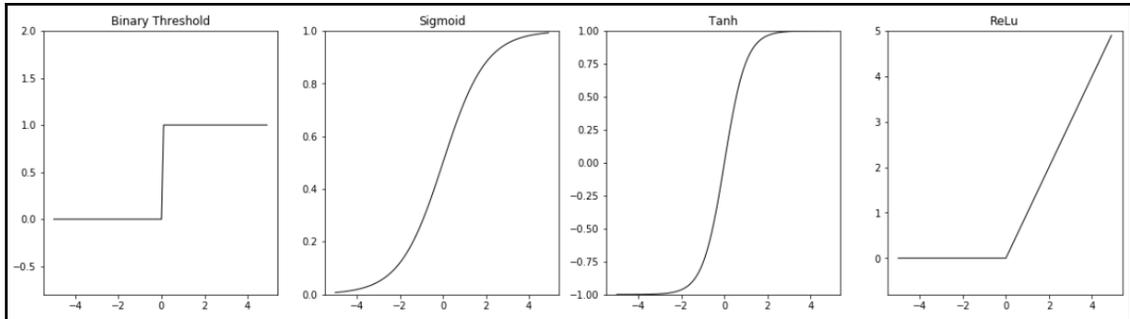
Activation function plots

Here are the activation function definitions:

Function name	Definition
Binary threshold	$f(x) = \begin{cases} 1 & , if\ x > 0 \\ 0 & , otherwise \end{cases}$
Sigmoid	$f(x) = \dfrac{1}{1 + e^{-x}}$
Tanh	$f(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLu	$f(x) = \begin{cases} x & , if\ x > 0 \\ 0 & , otherwise \end{cases}$ OR $\max\{0, x\}$

If we have a k-class (*k > 2*) classification problem, we basically want to learn the conditional probability distribution, *P(y|x)*. Thus, the output layer should have *k* neurons, and their value should sum to 1. To provide the network with the knowledge that the output of all *k* units should sum to 1, the **softmax activation** function is used. This is a generalization of the sigmoid activation. The softmax function squashes the output of each unit to be between 0 and 1, just like a sigmoid function.

Also, it divides each output such that the total sum of the outputs is equal to 1:
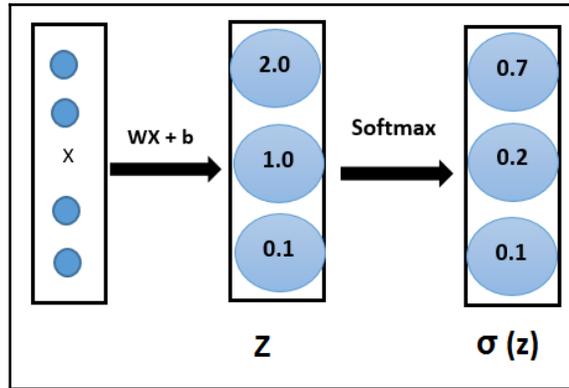
Mathematically the softmax function is shown as follows, where **z** is a vector of the input to the output layer (if you have 10 output units, then there are 10 elements in *z*). And again, *j* indexes the output units, so *j = 1, 2, ..., K:*

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Learning a simple non-linear unit – logistic unit

Let's suppose we have two class classification problems; that is, we need to predict the value of a binary outcome variable, *y*. In terms of probability, the outcome, *y*, is Bernoulli-distributed conditioned on the feature, *x*. The neural network needs to predict the probability, *P(y = 1 | x)*. For the output of the neural network to be a valid probability, it should lie in [0, 1]. We use a sigmoid activation function for this and get a non-linear logistic unit.

To learn the weights of the logistic unit, first we need a cost function and to find the derivatives of the cost function. From a probabilistic point of view, the cross-entropy loss arises as the natural cost function if we want to maximize the likelihood of the input data. Assuming we have a training dataset $X = \{x_n, t_n\}$, *n= 1, …, N*, the likelihood function can be written as:

$$p(\vec{t}|\vec{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

Where y_n is the output of the sigmoid unit after passing x_n as input data to the logistic unit. Note $\vec{t}$ *and* $\vec{w}$ represent the target vector (all N target values in the training set) and weight vector (set of all weights) of the sigmoid unit, respectively. The error function can be defined by taking the negative algorithm of the likelihood, and this gives the cross-entropy cost function:

$$E(\vec{w}) = -\ln p\left(\vec{t}\,|\vec{w}\right) = -\sum_{n=1}^{N} \{t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n)\}$$

To learn the weights of the logistic neural unit, we need the derivative of the output with regards to each of the weights. We will be using the *chain rule of derivatives* to derive the derivative of the error for the logistic unit:

Let
$$z = \sum_{k=1}^{d} w_i x_i, \ y = sigmoid(z), \ \frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i}\frac{\partial y}{\partial z} = x_i y(1 - y)$$

Therefore,
$$\frac{\partial E}{\partial w_i} = \sum_{n=1}^{N} \frac{\partial E}{\partial y_n}\frac{\partial y_n}{\partial w_i} = -\sum_{n=1}^{N} \frac{t_n - y_n}{y_n(1 - y_n)} x_i^n y_n(1 - y_n) = -\sum_{n=1}^{N} x_i^n(t_n - y_n)$$

The derivative we found looks very similar to the derivative with regards to the squared-error loss for the linear unit, but they are not the same. Let's take a closer look at the cross-entropy loss and see how it differs from the squared error. We can rewrite the cross-entropy loss as follows:

$$E(\vec{w}) = \begin{cases} -\ln(y_n) & if \ t_n = 1 \\ -\ln(1 - y_n) & if \ t_n = 0 \end{cases}$$

- Thus, for t_n = 1, $y_n = 1 \Rightarrow E(\vec{w}) = 0$, but $y_n = 0 \Rightarrow E(\vec{w}) = \infty$
- For *tn = 0,* $y_n = 1 \Rightarrow E(\vec{w}) = \infty$, but $y_n = 0 \Rightarrow E(\vec{w}) = 0$
- That is, the algorithm will be penalized with a very large cost if the prediction and the true value of the class label differ

Now, let's try using the *squared-error loss* with the logistic output. So, our cost function is:

$$E(\vec{w}) = \tfrac{1}{2} \sum_n (t_n - y_n)^2$$

Therefore, $\dfrac{\partial E}{\partial w_i} = \sum_{n=1}^{N} \dfrac{\partial E}{\partial y_n} \dfrac{\partial y_n}{\partial w_i} = -\sum_{n=1}^{N}(t_n - y_n)x_i^n y_n(1 - y_n) = -\sum_{n=1}^{N} x_i^n (t_n - y_n)\sigma^{'}(y_n)$

This error derivative depends directly on the derivative of the sigmoid function $\sigma^{'}(y_n)$. Now, the sigmoid function tends to 0 when y_n is highly negative and tends to 1 when y_n is highly positive. It's clear from the flat horizontal regions in the sigmoid curve that the gradient can shrink to be too small for such values of y_n. Thus, the squared-error derivative will have very small updates for such data points even if the t_n and y_n disagree a lot; that is, they are heavily misclassified by the network. This is called the **vanishing gradient problem**. For this reason, maximum likelihood-based cross-entropy loss is almost always the preferred loss function for training logistic units.

Loss functions

The loss function compares the output of a neural network with the target values in the training, producing a loss value/score to measure how well the predictions of the network match the expected value. In the last section, we saw the need for different types of loss functions for different tasks, such as regression and binary classification. Here are a few other popular loss functions:

- **Binary cross-entropy**: The log loss or cross-entropy loss for the two-class classification problem discussed in last section on logistic units.
- **Categorical cross-entropy**: If we have a *K* class classification problem, then we use the generalized cross-entropy for *K* classes.
- **Mean Squared Error**: This is the mean sum squared error we've discussed several times. This is widely used for various regression tasks.
- **Mean Absolute Error**: Measures the average magnitude of the errors in a set of predictions, without considering their direction. It's the average over the test sample of the absolute differences between prediction and actual observation. **Mean Absolute Error** (**MAE**) gives a relatively high weight to large errors as it squares the error.

- **Mean Absolute Percentage Error**: Measures the size of the error in percentage terms. It is calculated as the average of the unsigned percentage error. **Mean Absolute Percentage Error** (**MAPE**) is used because of the ease of interpretability of percentages.
- **Hinge loss/squared-hinge loss**: Hinge loss is used in SVMs. They penalize marginally misclassified points differently. They are good alternatives to cross-entropy loss and lead to faster training for neural networks as well. Higher-order hinge losses, such as squared-hinge losses, are even better for some classification tasks.
- **Kullback-Leibler (KL) divergence**: KL divergence is a measure of how one probability distribution diverges from a second expected probability distribution.

Data representations

All input and targets in the training set for a neural network training must be represented as tensors (or multi-dimensional arrays). Tensors are actually generalizations of two-dimensional matrices to an arbitrary number of dimensions. Typically, these are floating point tensors or integer tensors. Whatever the raw input data type—image, sound, text—it should be first converted to a suitable tensor representation. This step is called **data vectorization**. The following are tensors of different dimensions that we will be using frequently in this book:

- **Zero-D tensor or scalar**: A tensor that contains one single number is called a zero-dimensional tensor, zero-dimensional tensor, or scalar.
- **One-dimensional tensor or vector**: A tensor that contains an array of numbers is called a vector or one-dimensional tensor. The number of dimensions of the tensor is also called the **axes** of the tensor. One-dimensional tensor has exactly one axis.
- **Matrices (two-dimensional tensors)**: A tensor that contains an array of vectors is a matrix, or a two-dimensional tensor. A matrix has two axes (denoted by *rows* and *columns*).
- **Three-dimensional tensors**: By stacking a set of matrices (of same dimensions) in an array, we get a three-dimensional tensor.

By putting three-dimensional tensors in an array, four-dimensional tensors can be created. And so on. In deep learning, generally we play with zero-dimensional to four-dimensional tensors.

Tensors have three key attributes:

- Dimension or number of axes
- Shape of the tensor, that is, how many elements the tensor has along each axes
- Data type—whether it's an integer tensor or a floating-type tensor

Tensor examples

Following are a few example tensors which we will be frequently using while discussing the transfer learning use cases.

- **Time-series data**: A typical time-series data will have a time dimension, and the dimension will correspond to features at each time step. For example, an hourly measurement of temperature and humidity in a day is a time-series data that can be represented by a 2D tensor of shape (24, 2). So, a data batch will be represented by a 3D tensor.
- **Image data**: Images generally have three dimensions: width, height, and color channels. So, it can be represented by a 3D tensor. An image batch is represented by a 4D tensor, as shown in the following figure.
- **Video data**: A video consists of frames of images. So, to represent a single video, we need one more dimension. A single frame is a color image, and three dimensions are needed to represent a frame. This a video is represented by a 4D tensor of shape (frames, width, height, color channels):
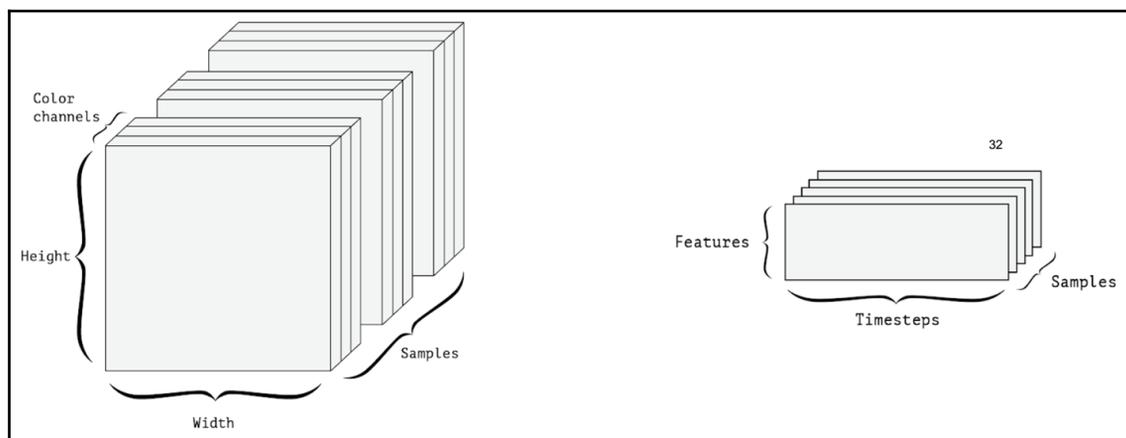
- **A data batch as tensor**: Suppose we have a batch of 10 images. A binary image such as in MNIST data can be represented by a 2D tensor. For a batch of 10 images, it can be represented by a 3D tensor. The first axes (axis = 0) of this 3D tensor is called the **batch dimension**.

Tensor operations

All computation done for training/testing a deep neural network can be formulated by a set of tensor operations. For example, adding, multiplying, and subtracting tensors. Following are some of the frequently used tensor operations in this book:

- **Element-wise operations**: Applying a function independently to all elements of a tensor is very commonly used in deep learning. For example, applying an activation function to all the units in a layer. Other element-wise operations include applying basic math operators, such as +, -, and *, element wise to two tensors of identical shape.
- **Tensor dot**: The dot product of two tensors is different from the elementwise product of two tensors. The dot product of two vectors is a scalar that is equal to sum of the elementwise products of the two vectors. The dot product of a matrix and a vector of compatible shape is a vector, and the dot product of two matrices of compatible shapes is another matrix. For two matrices x, y, with compatible shapes, we mean for dot (x, y) to be defined we should have *x.shape[1] = y.shape[0]*.
- **Broadcasting**: Suppose we want to add two differently shaped tensors. This commonly arises in every layer of the neural network. Let's take a ReLU layer, for example. The ReLU layer can be expressed by tensor operations as follows: *output = relu(dot(W, input) + b)*. Here, we are computing a dot product of the weight matrix with the input vector *x*. This will yield a vector, and then we add a scalar bias term. Actually, we want to add the bias term to each element of the output vector of the dot product. But, the bias tensor is a zero-dimensional tensor and the vector is one-dimensional. So, here we need to broadcast the smaller tensor to match the shape of the larger tensor. Broadcasting involves two steps: add axes to the smaller tensor to match the dimension of the larger tensor. Then, repeat the smaller tensor to match the shape of the larger tensor. We will explain this with a concrete example: Let x be of shape (32, 10) and y of shape (, 10). We want to compute x+y. Following the first step of broadcasting, we add the axis to y (the smaller tensor) and get a y1 tensor of shape (1, 10). To match the dimensions of x, we repeat y1 32 times and get a y2 tensor of shape (32, 10). Then, we calculate the elementwise addition x + y2.

- **Reshaping**: Tensor reshaping is an operation where elements of the tensor are rearranged along the axes. One simple example of reshaping is transposing a 2D tensor. In a transpose operation on a matrix, the *rows* and *columns* are interchanged. Another example of reshaping is flattening the tensor. A multidimensional tensor can be reshaped to a vector or 1D tensor by putting all the elements of the tensor along one axis. Here are a few example tensor operation implementations in TensorFlow:

```
#EXAMPLE of Tensor Operations using tensorflow.
import tensorflow as tf

# Initialize 3 constants: 2 vectors, a scalar and a 2D tensor
x1 = tf.constant([1,2,3,4])
x2 = tf.constant([5,6,7,8])
b = tf.constant(10)
W = tf.constant(-1, shape=[4, 2])
# Elementwise Multiply/subtract
res_elem_wise_mult = tf.multiply(x1, x2)
res_elem_wise_sub = tf.subtract(x1, x2)

#dot product of two tensors of compatable shapes
res_dot_product = tf.tensordot(x1, x2, axes=1)

#broadcasting : add scalar 10 to all elements of the vector
res_broadcast = tf.add(x1, b)

#Calculating Wtx
res_matrix_vector_dot = tf.multiply(tf.transpose(W), x1)

#scalar multiplication
scal_mult_matrix = tf.scalar_mul(scalar=10, x=W)

# Initialize Session and execute
with tf.Session() as sess:
    output = sess.run([res_elem_wise_mult,res_elem_wise_sub,
                        res_dot_product,
                        res_broadcast,res_matrix_vector_dot,
                        scal_mult_matrix])
    print(output)
```

Multilayered neural networks

A single layered non-linear unit still has a limited capacity for the input output transformations it can learn. It can be explained by looking at the XOR problem. In the XOR problem, we want a neural network model to learn the XOR function. The XOR function takes two Boolean input, and outputs 1 if they differ, or 0 if the input is identical.

We can think of it as a pattern-classification problem with input patterns of *X = {(0,0), (0, 1), (1, 0), (1,1)}*. The first and fourth are in class 0 and the others in class 1. Let's treat this problem as a regression problem, with **Mean Square Error** (**MSE**) loss and try to model this with a linear unit. Solving it analytically, we arrive at the desired weights : $\vec{w} = (0,0)$ and bias: $b = \frac{1}{2}$. This model outputs 0.5 for all input values. Thus, the XOR function cannot be learned by a simple linear neuron.

One way to solve the XOR problem is to use a different representation of the input such that the linear model is able to find a solution. This can be achieved by adding a non-linear hidden layer to the network. We will use a ReLU layer with two hidden units. The output is Boolean so the most suitable output neuron is a logistic unit. We can use binary cross-entropy loss to learn the weights:
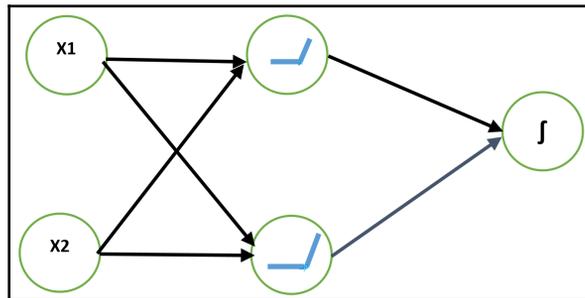
Let's learn the weights of this network using SGD. The following is the `keras` code for the XOR function learning problem:

```
model_input = Input(shape=(2,), dtype='float32')
z = Dense(2,name='HiddenLayer', kernel_initializer='ones')(model_input)
z = Activation('relu')(z) #hidden activation ReLu
z = Dense(1, name='OutputLayer')(z)
model_output = Activation('sigmoid')(z) #Output activation
model = Model(model_input, model_output)
model.summary()

#Compile model with SGD optimization, with learning rate = 0.5
sgd = SGD(lr=0.5)
```

```
model.compile(loss="binary_crossentropy", optimizer=sgd)
#The data set is very small - will use full batch - setting batch size = 4
model.fit(X, y, batch_size=4, epochs=300,verbose=0)
#Output of model
preds = np.round(model.predict(X),decimals=3)
pd.DataFrame({'Y_actual':list(y), 'Predictions':list(preds)})
```

The output of the preceding code is as follows:
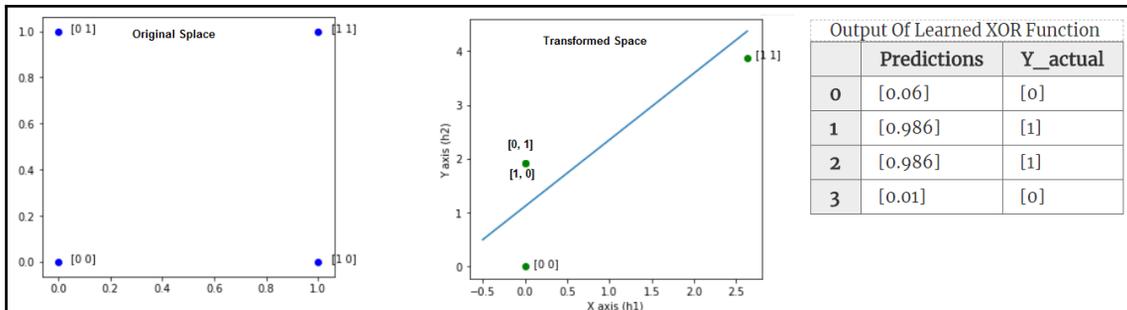
(Left) Shows the original space of 4 points—clearly no line can separate class 0 points {(0,0),(1,1)} from other class. (Center) Shows the transformed space learned by the hidden ReLU layer. (Right) The table shows the predicted values obtained by the function

The neural network with one hidden layer is able to learn the XOR function. This example explains the need for non-linear hidden layers for neural networks to do something meaningful. Let's take a closer look at what input transformation the hidden layer has learned that makes it possible for the output-logistic neuron to learn the function. In Keras, we can extract an intermediate hidden layer from a learned model and use it to extract the transformation of input it's doing before passing to the output layer. The preceding figure shows how the the input spaces of four points are transformed. Post-transformation, the class 1 and class 0 points can be easily separated by a line. Here is the code to generate the plot for the original space and the transformed space:

```
import matplotlib.pyplot as plt

#Extract intermediate Layer function from Model
hidden_layer_output = Model(inputs=model.input,
outputs=model.get_layer('HiddenLayer').output)
projection = hidden_layer_output.predict(X) #use predict function to
                                extract the transformations
```

```
#Plotting the transformed input

fig = plt.figure(figsize=(5,10))
ax = fig.add_subplot(211)
plt.scatter(x=projection[:, 0], y=projection[:, 1], c=('g'))
```

By stacking multiple non-linear hidden layers, we can build networks that are capable of learning very complex non-linear input-output transformations.

Backprop – training deep neural networks

For training deep multilayered neural networks, we can still use the gradient decent/SGD. But, SGD will require computing the derivatives of the loss function with regards to all the weights of the network. We have seen how to apply the chain rule of derivatives to compute the derivative for the logistic unit.

Now, for a deeper network, we can recursively apply the same chain rule layer by layer to obtain the derivative of the loss function with regards to the weights corresponding to layers at different depths in the network. This is called the backpropagation algorithm.

Backpropagation was invented in the 1970s as a general optimization method for performing automatic differentiation of complex nested functions or functions of a function. However, it wasn't until 1986, with the publishing of a paper by Rumelhart, Hinton, and Williams, titled *Learning representations by back-propagating errors* (`https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf`), that the importance of the algorithm was appreciated by the greater ML community. Backpropagation was one of the first methods able to demonstrate that artificial neural networks could learn good internal representations; that is, their hidden layers learned nontrivial features.

The backpropagation algorithm is an efficient way of computing the error derivative, $\frac{dE}{d\theta}$, for every weight on a single training example. To understand the backpropagation algorithm, let's first represent a neural network with a computational graph notation. A computational graph of a neural network will have nodes and directed edges, where the nodes represent variables (tensors) and the edges represent operations on variables connecting to the next variable. A variable, x, is connected to y by a directed edge if $y = f(x)$, for some function, f.

For example, the graph for the logistic unit can be represented as follows:
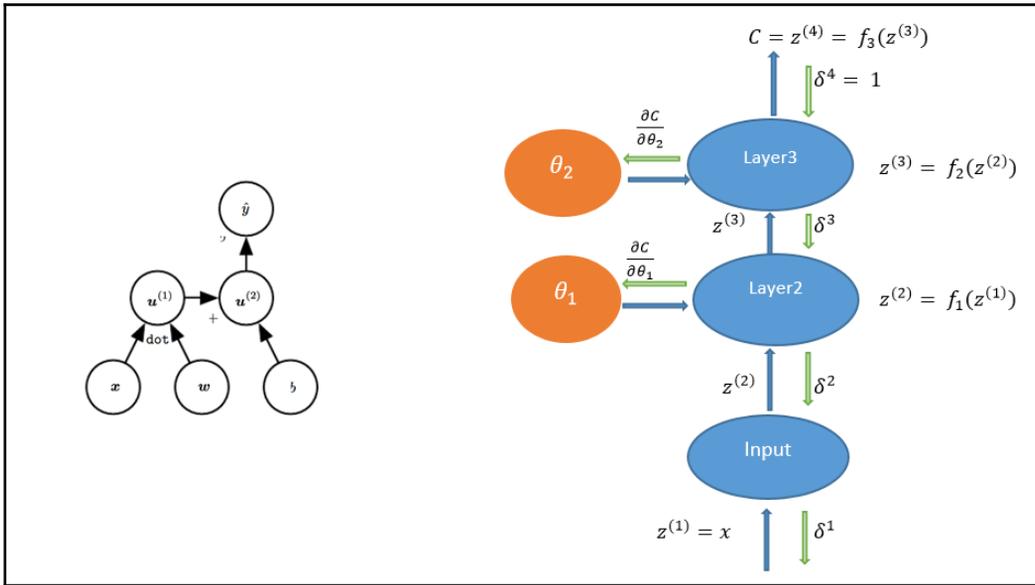
(Left) Logistic regression as a computation graph. (Right) Information flow of the BP algorithm for a three-layered network computation graph

We represent the computation nodes by u^1, u^2, ..., u^n. Also, we arrange the nodes in order so that they can be computed one after the other. Here, u^n is a scalar—the loss function. Let's denote the parameters or weights of the network by nodes, θ_k. To apply gradient descent, we need to compute all the derivatives, $\dfrac{\partial u^n}{\partial \theta_k}$. The forward computation for this graph can be calculated by following the directed path in the computation graph from input nodes to the final node, u^n. This is called **forward pass**.

Since the nodes in our graphs are tensors, to compute the partial derivatives, $\dfrac{\partial u^n}{\partial \theta_k}$, the chain rule of derivatives for functions of several variables will be used, which is represented by a multiplication of a Jacobian with a gradient. The backpropagation algorithm involves a sequence of such Jacobian-Gradient products.

Backpropagation algorithm is represented as follows:

1. Given input vectors, $X = \{\vec{x_n}\}$, target vectors, $Y = \{\vec{t_n}\}$, a cost function C to measure network error, and a set of initial weights of a network, compute a forward pass of the network and calculate loss, C

2. Backward pass—for each training example $\left(\vec{x}_n, t_n\right)$, calculate the derivative of loss, C, with respect to each layer parameters/weights—this step is discussed as follows:

    1. Combine individual gradients by taking an average over all the gradients for the input-target pairs or a mini-batch of them

    2. Update each of the parameters, $\triangle \theta_l = -\alpha \frac{\partial C}{\partial \theta_l}$, α being the learning rate

We will explain *backward pass* with a fully connected three-layered neural network. The computation graph for this is shown in the preceding figure. Let $z^{(i)}$ represent a computation node in the graph. To perform backpropagation, the computation of derivative $\frac{\partial C}{\partial z^{(i)}}$ will proceed in exactly the reverse order of the forward pass. These are indicated by the downward arrows. Let's denote the derivative of the cost function with regards to the layer l input $z^{(l)}$ $z^{(l)}$ by $\delta^{(l)}$. For the topmost layer, let $\delta^{(4)} = 1$. To compute recursively, let's consider a single layer. A layer has an input, $z^{(l)}$, and output, $z^{(l+1)}$. Also, the layer will take input $\delta^{(l+1)}$ and produce $\delta^{(l)}$ and $\frac{\partial C}{\partial \theta_l}$.

For layer l,
$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \sum_j \frac{\partial C}{\partial z_{l+1}^j} \frac{\partial z_{l+1}^j}{\partial z_i^l} = \sum_j \delta_i^{l+1} \frac{\partial z_{l+1}^j}{\partial z_i^l}$$

i represents the ith component of the gradient δ_i^l.

Thus, we arrive at a recursive formula for computing the backward messages. Using these, we can also compute the derivative of the cost with regards to the model parameters, as follows:

$$\frac{\partial C}{\partial \theta_l} = \sum_j \frac{\partial C}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial \theta_l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial \theta_l}$$

The backpropagation algorithm computes the gradient of the scalar cost function, z, with respect to its ancestor, x, in the computation graph. The algorithm begins by computing the derivative of the cost, z, with regards to itself, $\frac{\partial z}{\partial z} = 1$. The gradient with regards to the parent of z can be computed by multiplying the current gradient by the Jacobian of the operation that produced z. We keep traversing the compute graph backwards and multiplying the Jacobians until we reach the input, x.

Challenges in neural network learning

Optimization is, in general, a very difficult task. In this section, we discuss some common challenges involved in optimization methods for training deep models. Understanding these challenges is crucial to assessing the training performance of the neural network model and taking corrective measures to possibly alleviate issues.

Ill-conditioning

The condition number of a matrix is the ratio of the largest singular value to the smallest singular value. A matrix is ill-conditioned if the condition number is very high, usually indicating that the lowest singular value is orders of magnitude smaller than the highest one, and rows of the matrix are heavily correlated with each other. This is a very general problem in optimization. In fact, it makes even convex optimization problems difficult to solve. Generally, neural networks have this problem, which causes SGD to get stuck, that is, learning becomes very slow in spite of the existence of a strong gradient. For the dataset with a good condition number, close to 1, the error contours are nearly circular and the negative gradient always points straight at the minimum of the error surface. For a poor-conditioned dataset, the error surface is relatively flat in one or more directions and strongly curved in other directions. For complex neural networks, it may not be possible to find the Hessian and the ill-conditioning effect analytically. However, the effect of ill-conditioning can be monitored by plotting the squared gradient norm and $g^T H_g$ over training epochs.

Let's consider the second-order Taylor-series approximation of the *f(x)* function that we want to optimize. The second-order Taylor series at point x_0 is given by:

$$f(x) = f(x_0) + (x - x_0)^T g + \tfrac{1}{2}(x - x_0)^T H(x - x_0)$$

Where *g* is the gradient vector and *H* is the Hessian of *f(x)* at x_0. If ε is the learning rate we are using, then the new point according to gradient descent is x_0 - ϵ_g. Substituting this in the Taylor-series expansion, we get $f(x_0 - \epsilon g) \approx f(x_0) - \epsilon g^T g + \tfrac{1}{2}\epsilon^2 g^T H g$.

Note that if $-\epsilon g^T g + \frac{1}{2}\epsilon^2 g^T H_g > 0$ then the value of the function at the new point will increase compared to x_o. Also, in the presence of strong gradients, we will have a high squared gradient norm, $\|g\|^2 = g^T g$, but, simultaneously if the other quantity, $g^T H_g$, grows by an order of magnitude, then we will see *f(x)* decreasing at a very slow rate. However, if we could shrink the learning rate, ε, at that point, then maybe this effect can be nullified to some extent, as the $g^T H_g$ quantity is multiplied by ϵ^2. The effect of ill-conditioning can be monitored by plotting the squared gradient norm and $g^T H_g$ over training epochs. We have seen in the *Hot plate* example of how to calculate the gradient norm.

Local minima and saddle points

DNN models are essentially guaranteed to have an extremely large number of local minima. If a local minimum has a high cost compared to a global minima, this could be problematic. For a long time, it was assumed that, due to the existence of such local minima, neural network training is plagued. This is still an active research area, but now it's suspected that for DNNs, most of the local minima have low cost value and it's not necessary to a find global minimum, but rather a point in the weight space where we have a sufficiently low value of the cost function. The existence of strong local minima can be detected by monitoring the gradient norm. If the gradient norm decreased to an insignificantly small order, then it indicates the existence of local minima.

Saddle points are points that are neither maxima nor minima, but they are surrounded by a flat region, on one side of which the objective function value increases, and on the other side, the objective function decreases. Because of this flat region, the gradients become very small. However, it has been observed that empirically gradient descent escapes such regions rapidly.

Cliffs and exploding gradients

Objective functions for highly non-linear DNNs have extremely steep, regions resembling cliffs, as shown in the following figure. Moving in the direction of a negative gradient of an extremely steep cliff structure can move the weights so far off that we jump off the cliff structure altogether. Thus, missing the minima at a point when we are very close.

Thus, nullifying much of the work that had been done to reach the current solution:
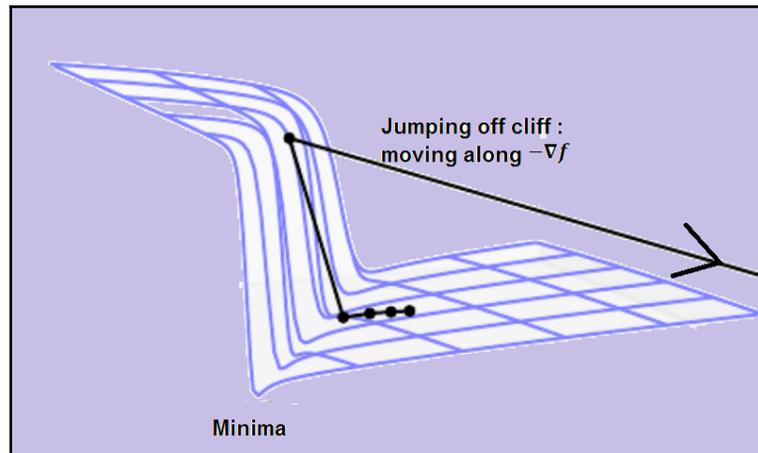
Explaining when we need to clip the gradient norm

We can avoid such bad moves in gradient descent by clipping the gradient, that is, setting an upper bound on the magnitude of the gradient. We recall that the gradient descent is based on the first-order Taylor approximation of the function. This approximation holds good in an infinitesimal region around the point where the gradient is computed. The cost function may begin to increase or curve back upward if we jump outside this region. So, we need to restrict the length of the move. The gradient can still give the approximately correct direction. The update must be chosen to be small enough to avoid traversing too much upward curvature. One way to achieve this is by **clipping the norm of the gradient** by setting a threshold for the upper bound on the norm:

$$if \; \|g\| > v, \; g \leftarrow \frac{gv}{\|g\|}$$

In Keras, this can be implemented as follows:

```
#The parameters clipnorm and clipvalue can be used with all optimizers #to
control gradient clipping:

from keras import optimizers

# All parameter gradients will be clipped to max norm of 1.0
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
#Similarly for ADAM optmizer
adam = optimizers.Adam(clipnorm=1.)
```

Initialization – bad correspondence between the local and global structure of the objective

To start a numerical optimization algorithm, such as SGD, we need to initialize the weights. If we have an objective function, as in following figure, by making small local moves suggested by SGD, we will be wasting a lot of time if we start on the side of the mountain across from which the true minima lie. This is a scenario where the local structure of the objective function does not give any hint about where the minima lie. It can be avoided by proper initialization. If we could start the SGD at some point on the other side of the hill, the optimization would be much faster:
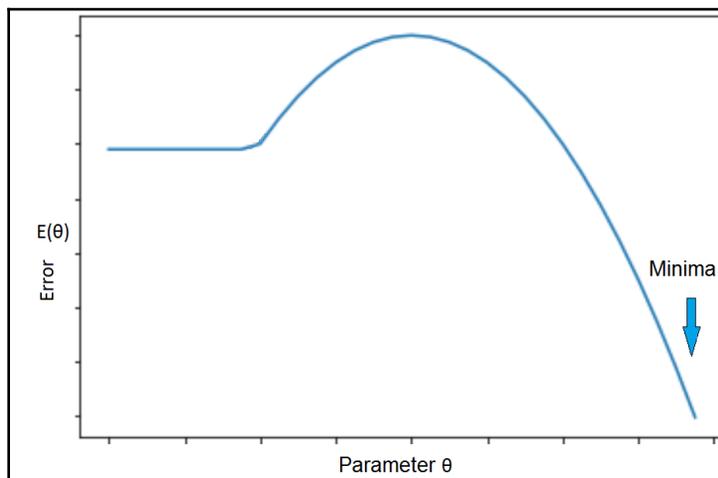
Explaining a scenario of bad initialization and the effect on gradient-based optimization

Inexact gradients

Most of the optimization algorithms are based on assumptions that we have the exact gradient known at a given point. But, actually we only have an estimate of the gradient. How good is this estimate? In SGD, the batch size significantly influences the behavior of the stochastic-optimization algorithm, since it determines the variance of the gradient estimates.

In summary, the different problems that are faced in neural network training can be addressed by the following four tricks:

- Choosing a proper learning rate—possibly an adaptive learning rate for each parameter

  - Choosing a good batch size—gradient estimates are dependent on this
  - Choosing a good initialization of the weights
  - Choosing the right activation functions for hidden layers

Now, let's briefly discuss the various heuristics/strategies that made learning DNNs practically feasible and continue to make deep learning a huge success.

Initialization of model parameters

Here's how the choice of initial points impacts the performance of the iterative learning algorithm for deep neural networks:

  - Initial points can determine whether the learning will converge or not
  - Even if learning converges, how quickly it converges depends on the initial point
  - Initial points of comparable costs may have varying generalization errors

Initialization algorithms are mostly heuristics. The whole point of a good initialization is having one that somehow enables faster learning. One of the important aspects of initialization is to *break symmetry* of the initial set of weights to the units of hidden layers. Two units with same activation function at the same level of the network will be updated equally if they are initialized with the same weight. The reason multiple units are kept in the hidden layer is that they should learn different functions. So, getting equally updated will not leaf to a different function learning.

A simple way to break symmetry is use random initialization—sampling from a Gaussian or uniform distribution. Bias parameters in the models can be heuristically chosen constants. Choosing the magnitude of weights depends on a trade-off between optimization and regularization. Regularization demands that the weights should not be very large—this may lead to poor generalization performance. Optimization needs the weights to be large enough to propagate information through the network successfully.

Initialization heuristics

Let's consider a dense layer with *m* input and *n* output units:

  1. Sample each weight from the uniform distribution $\left[\frac{-1}{\sqrt{m}}, \frac{1}{\sqrt{m}} \right]$.

2. Glorot and Bengio suggested a normalized version of the uniform distribution initialization: $\left[-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right]$. This is designed to have same variance of the gradients in each layer, called **Glorot Uniform**.

3. Sampling each weight from normal distribution with a mean of 0 and a variance of $\sqrt{2/(m+n)}$. This is similar to Glorot Uniform and called **Glorot Normal**.

4. For very large layers, individual weights will become extremely small. To fix that, an alternative is to initialize only k non-zero weights. This is called **sparse initialization**.

5. Initialize the weights to be random orthogonal matrices. Gram-Schmidt orthogonalization can be used on an initial weight matrix.

Initialization schemes can also be treated as a hyperparameter in neural network training. If we have enough computing resources, different initialization schemes can be evaluated and we can chose the one with the best generalization performance and faster convergence rate.

Improvements of SGD

Different optimization algorithms have been proposed in recent years, which use different equations to update a model's parameters.

The momentum method

Cost function may have regions of high curvature and small but consistent gradients. This is due to the poor conditioning of the Hessian matrix and variance in the stochastic gradient. SGD may slow down a lot in such regions. The momentum algorithm accumulates the **exponentially weighted moving-average** (**EWMA**) of previous gradients and makes a move in that direction instead of the local gradient direction suggested by SGD. The exponential weighting is controlled by the hyperparameter, $\alpha \in [0,1)$ which determines how quickly the effect of the previous gradient decays. The momentum method damps the oscillations in directions of high curvature by combining gradients of opposite signs.

Nesterov momentum

Nesterov momentum is a variant of the momentum algorithm that differs from the momentum method only at the point the gradient is calculated. The standard momentum method computes the gradient first at the current location and then takes a big jump in the direction of the accumulated gradient. The Nesterov momentum first makes a big jump along the direction of the previously accumulated gradient and then computes the gradient at the new point. The new gradient is corrected by again taking the EWMA of all previous gradients:
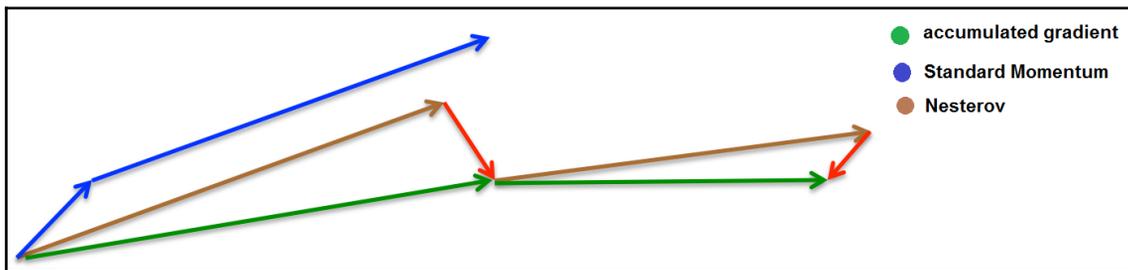
Adaptive learning rate – separate for each connection

In the preceding methods the same learning rate is applied to all parameter updates. Having sparse data, we may instead want to update the parameters in different extent. Adaptive gradient descent algorithms, such as AdaGrad, AdaDelta, RMSprop, and Adam, provide an alternative to classical SGD by keeping per parameter learning rates.

AdaGrad

The AdaGrad algorithm adapts the learning rate for each connection by scaling them *inversely* proportional to the square root of all previous gradients' sum-squared values. Thus, larger moves are made in the gently sloped direction of the error surface. However, applying this trick from very beginning of training may cause some of the learning rates to diminish drastically. But, AdaGrad has still performed very well on several deep learning tasks.

RMSprop

RMSprop modifies the AdaGrad algorithm by taking the EWMA of previous squared gradients. It has the moving average parameter, ρ, which controls the length and scale of the moving average. This is one of the most successful algorithms for deep neural network training.

Adam

Adaptive Moments (**Adam**). It has taken the best of both of the momentum-based and adaptive-learning-rate algorithms and combined them. Here, the momentum algorithm is applied to rescaled gradients computed by RMSprop.

Overfitting and underfitting in neural networks

As in any other ML training, the dataset for training a deep learning model is split into train, test, and validation. During iterative training of the model, typically the validation error is slightly more than the training error. If the gap between the test error and the validation error increases over iterations, it's a case of **overfitting**. If the training error stops decreasing to a sufficiently low value, we can conclude that the model is **underfitting**.

Model capacity

The capacity of a model describes the complexity of the input-output relationships it can model; that is, how large a set of functions is allowed in the hypothesis space of the model. For example, a linear regression model can be generalized to include polynomials rather than only linear functions. This can be done by taking n integral powers of x as input along with x while building the model. The capacity of the model can also be controlled by adding multiple hidden nonlinear layers to the network. So, we can make the neural network model either wider or deeper, or both, to increase the capacity of the model.

However, there is a trade-off between the model capacity and the generalization error of the model:
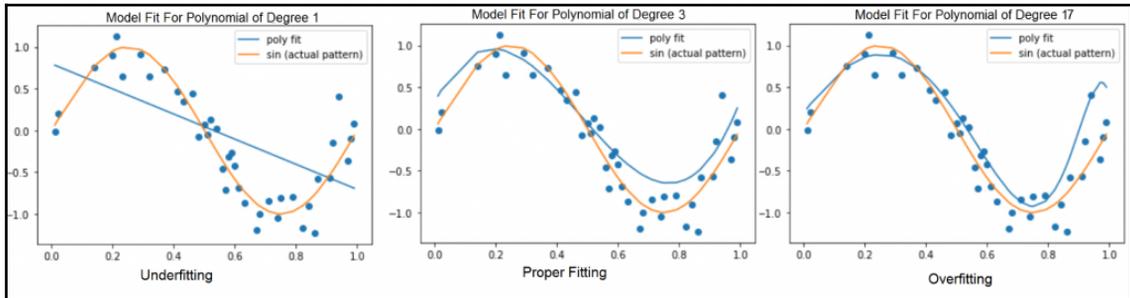
(Left): A linear function fit to the data suffers from underfitting. (Center): A quadratic function fit to the data generalizes well to unseen points
(Right) A polynomial of degree 9 fit to the data suffers from overfitting

Models with very high capacity can overfit the training set by learning patterns in training sets that may not generalize well to unseen test sets. Also, it can fit very well on small amounts of training data. On the other hand, models with low capacity may struggle to fit the training set:
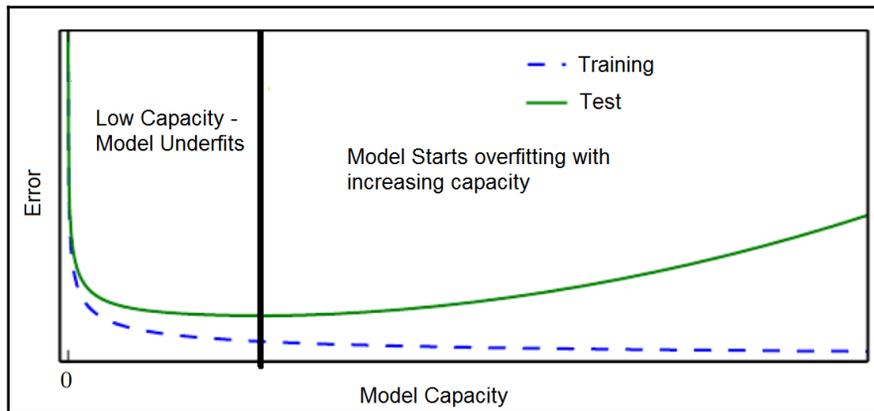
Overfitting/underfitting in terms of training and validation loss

How to avoid overfitting – regularization

Overfitting is a central problem in ML. For neural networks, many strategies are developed to avoid overfitting and reduce generalization errors. These strategies are collectively known as **regularization**.

Weight-sharing

Weight-sharing means the same set of weights is being used in different layers in the network and thus we have fewer parameters to optimize. This is seen in some popular deep learning architectures, such as the Siamese network and RNNs. Using shared weights in a few layers helps the model generalize better by controlling the model capacity. Backpropagation can easily incorporate linear weight constraints such as weight-sharing. Another type of weight-sharing is used in CNNs where, unlike a fully connected hidden layer, a convolution layer has connections between local regions. In CNN, it's assumed that the input (such as image or text) to be processed by the network can be decomposed into a set of local regions with the same nature and thus each of them can be processed with the same set of transformations, that is, shared weights. An RNN can be treated as a feed-forward network where each successive layer shares the same set of weights.

Weight-decay

It can be observed that an overfitted model—like the polynomial in the preceding example—has very large weights. To prevent this, a penalty term, Ω, can be added to the objective function, which will drive the weights closer to the origin. Thus, the penalty term should be a function of the norm of the weights. Also, the effect of the penalty term can be controlled by multiplying with a hyperparameter, α. So our objective function becomes: $E(w) + \alpha\Omega(w)$. The popularly-used penalty terms are:

- **L^2 regularization**: Penalty term is given by $\Omega = \frac{1}{2}\|w\|^2$. In regression literature, this is called **ridge regression**.
- **L^1 regularization**: Penalty term is given by $\Omega = \|w\|_1 = \sum_i w_i$. This is called **lasso regression**.

L^1 regularization leads to a sparse solution; that is, it sets many of the weights to zero and thus acts as a good feature-selection method for regression problems.

Early stopping

As the training for a large neural network proceeds, training errors decrease steadily over time, but as shown in the following figure, validation set errors starts to increase beyond some iterations:
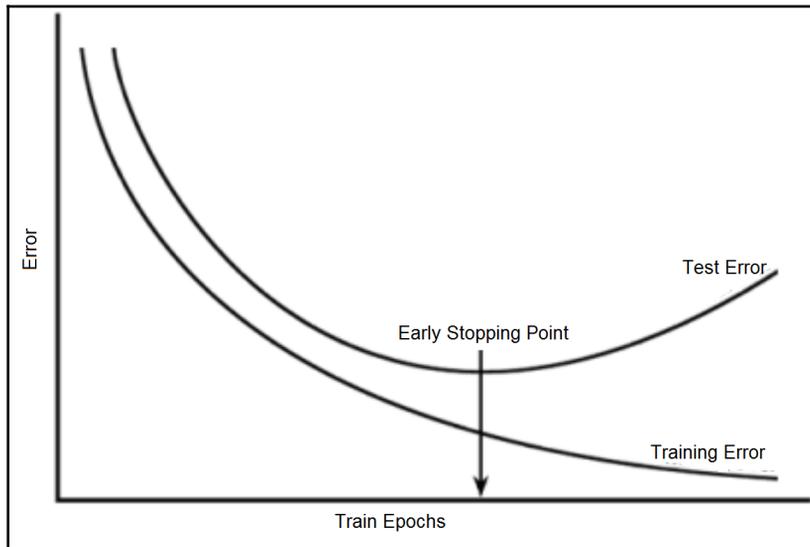
Early stopping: training versus validation error

If the training is stopped at the point where the validation errors start increasing, we can have a model with better generalization performance. This is called **early stopping**. It's controlled by a patience hyperparameter, which sets the number of times to observe increasing validation set error before training is aborted. Early stopping can be used either alone or in conjunction with other regularization strategies.

Dropout

Dropout is a computationally inexpensive but powerful method of regularizing in deep neural networks. It can be applied separately to both the input layer and the hidden layers. Dropout randomly masks the output of a fraction of nodes from a layer by setting their output to zero during the forward pass. It's equivalent to removing a fraction of nodes from a layer and creating a new neural network with fewer nodes. Typically, at the input layer 0.2 nodes are dropped out and in hidden layers up to 0.5 fraction of nodes are dropped.

Model averaging (ensemble methods) is heavily used in ML for reducing generalization error by combining the output of various models. Bagging is one ensemble method where k different datasets are constructed by random sampling with replacement from the training set and separate k models are trained on each of them. In particular, for a regression problem, the final output of the model is an average of the outputs of the k models. There are other combination strategies as well.

Dropout can be also thought as a model-averaging method where many models are created by changing the number of active nodes at various layers of the base model on which dropout is applied.

Batch normalization

It's common practice in ML to scale and standardize the input training data before feeding it to a model for training. For neural networks as well, scaling is one of the preprocessing steps and has shown some improvements in the model performance. Can we apply the same trick before we feed the data to a hidden layer? Batch normalization is based on this idea. It normalizes the previous layer's activations by subtracting the mini-batch mean, μ, of activations and dividing by the mini-batch standard deviation, σ. While making predictions, we may have only one example at a time. So, calculating batch mean μ and batch σ is not possible. These are replaced by an average over all the values collected at training time.

Do we need more data?

The best way to make a neural model have better generalizations or test performance is by training it with more data. In practice, we have very limited training data. The following are a few popular strategies used to get more training data:

- **Synthetically generate some training samples**: It's not always easy to generate fake training data. But, for some types of data, such as image/video/speech, transformations can be applied to the original data to generate new data. For example, images can be translated, rotated, or scaled to generate new image samples.
- **Training with noise**: Adding controlled random noise to the training data is another popular data-augmentation strategy. Noise can also be added to the hidden layers of the neural network.

Hyperparameters of the neural network

The architecture-level parameters of a neural network, such as the number of hidden layers, number of units per hidden layer, and training-related parameters, such as learning rate, optimizer algorithm, optimizer parameter—momentum, L1/L2 regularizer, and dropout, are collectively called the **hyperparameters** of the neural network. The weights of the neural network are called the **parameters** of the neural network. Some hyperparameters affect the time and cost of training the algorithms and some affect the generalization performance of the model.

Automatic hyperparameter tuning

There are various methods developed for the hyperparameter tuning. But, for most of them, one needs to specify a certain range of values for each of the hyperparameters. Most hyperparameters can be set by understanding the effect they will have on model capacity.

Grid search

A grid search is an exhaustive search through a manually-specified subset of the hyperparameter space. The grid search algorithm requires a performance metric, such as cross-validation error or validation-set error to evaluate the best possible parameter. Typically, a grid search involves selecting parameters in logarithmic scale. For example, a learning rate taken within the set {0.1, 0.01, 0.001, 0.0001} or a number of hidden units can be chosen from the set {50, 100, 200, 500, 1000, ...}. The computational cost for the grid search grows exponentially with the number of hyperparameters. So, another popular technique is the randomized grid search. The randomized search samples parameters a fixed number of times from all the specified ranges of parameters. This is found to be more effective than an exhaustive search when we have high-dimensional hyperparameter spaces. It is better because there could be some hyperparameters that do not significantly affect the loss.

Summary

We covered a lot of ground in this chapter as regards the fundamentals of deep learning. We really commend your efforts on getting this far! The idea of this chapter was to introduce you to the core concepts and terminology pertaining to the domain of deep learning. We started with a brief introduction of deep learning and then looked at the popular frameworks in today's deep learning landscape. Detailed step-by-step guides have also been included for setting up your own deep learning environments to develop and train large-scale deep learning models on GPUs.

Finally, we covered essential concepts around neural networks including linear and non-linear neurons, data representation, chain rule, loss functions, multilayer networks, and SGD. The challenges of learning in neural networks were also covered, including popular caveats surrounding local minima and exploding gradients. We looked at overfitting and underfitting issues in neural networks, along with strategies to handle them. Then we covered popular initialization heuristics for neural network units. Besides this, we explored some newer optimization techniques, which are improvements over vanilla SGD, including popular methods such as RMSprop and Adam.

In the next chapter, we will explore the different architectures around deep learning models that are leveraged to solve different types of problems.

3
Understanding Deep Learning Architectures

This chapter will focus on understanding the various architectures present today in deep learning. A lot of the success of neural networks lies in the careful design of the neural network architecture. We have come a long way since the traditional **Artificial Neural Networks** (**ANNs**) of the 1960s. In this book, essential model architectures such as fully connected deep neural networks, **Convolutional Neural Networks** (**CNNs**), **recurrent neural networks** (**RNNs**), **Long Short-Term Memory** (**LSTM**) networks, and the most recent capsule networks will be covered.

In this chapter, we will cover the following topics:

- Why neural network architecture design is important
- Various popular architecture designs and applications

Neural network architecture

The word **architecture** refers to the overall structure of the neural network, including how many layers it can have and how units in the layers should be connected to each other (for instance, units across successive layers can be fully connected, partially connected, or may even skip the next layer altogether and then make connections to a layer at a much higher level in the network). With the availability of modular deep learning frameworks, such as Caffe, Torch, and TensorFlow, complex neural network designs have been revolutionized. Now we can compare neural network designs to Lego blocks, where you can build almost any structure that you can imagine. However, these designs are not just random guesses. The intuitions behind these designs are usually driven by the domain knowledge the designer has about the problem, along with some trial and error to fine-tune the final design.

Why different architectures are needed

A feedforward multilayered neural network has the capacity to learn a huge hypothesis space and extract complex features in each of the nonlinear hidden layers. So, why do we need different architectures? Let's try to understand this.

Feature engineering is one of the most important aspects in **machine learning** (**ML**). With too few or irrelevant features, we may have underfitting; and with too many features, we may be overfitting the data. Creating a good, hand-crafted set of features is a tedious, time-consuming, and iterative task.

Deep learning comes with a promise that, given enough data, the deep learning model is capable of automatically figuring out the right set of features—that is, a hierarchy of features of increasing complexity. Well, the promise of deep learning is simultaneously true and slightly misleading. Deep learning has indeed simplified feature engineering in many scenarios, but it certainly hasn't completely eradicated the need for it. As manual feature engineering has decreased, the architectures of neural network models themselves have become increasingly more complex. Specific architectures are designed to solve specific problems. Architecture engineering is a much more generic approach than hand-crafted feature engineering. In architecture engineering, unlike in feature engineering, domain knowledge is not hardcoded as specific features but is instead only used at an abstract level. For example, if we were dealing with image data, one very high-level piece of information about the data is two-dimensional locality of object pixels and another is translation invariance. In other words, translating a cat's image by a few pixels still keeps it a cat.

In the feature engineering approach, we have to use very specific features, such as edge detectors, corner detectors, and various smoothing filters, to build a classifier for any image processing/computer vision task. Now, for neural networks, how can we code two-dimensional locality and translation invariance information? If a dense, fully connected layer is placed preceding the input data layer, then every pixel in the image is connected to each unit in the dense layer. But, pixels from two spatially distant objects need not be connected to same hidden unit. It may be that a neural network with strong L1 regularization would be able to sparsify the weights after prolonged training with lots of data. We can design the architecture to restrict only local connections to the next layer. A small set of neighboring pixels (say, a 10 x 10 sub-image of pixels) may have connections to one unit of the hidden layer. Because of translation invariance, the weights used in these connections can be reused. This is what a CNN does. This weight-reuse strategy has other benefits, such as dramatically lowering the number of model parameters. This helps the models to generalize.

Let's take another example of how abstract domain knowledge can be hardcoded into neural networks. Suppose we have temporal data or sequential data. A normal feedforward network would treat each input example as independent of the previous input. So, any hidden feature representation that is learned should also depend on the recent history of the data rather than just the current data. So, a neural network should have some feedback loops or memory. This key idea gave rise to the recurrent neural network architecture and its modern robust variants, such as LSTM networks.

Other advanced ML problems, such as speech translation, question-answering systems, and relationship modeling have demanded the development of varied deep learning architectures.

Various architectures

Let's now take a look at some popular neural network architectures and their applications. We will start with **multilayered perceptron** (**MLP**) networks. We have already covered single-layered perceptron networks, which are the most basic neural network architectures.

MLPs and deep neural networks

MLP or simple **deep neural networks** (**DNNs**) are the most basic form of neural network architectures. The neural units are arranged layer after layer, and adjacent network layers are fully connected to one another. We have already discussed this in detail in the previous chapter:
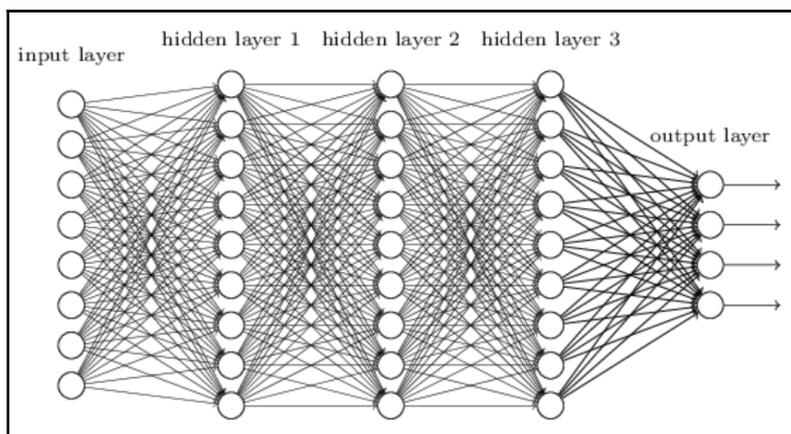
Autoencoder neural networks

Autoencoders are typically used for reducing the dimensionality of data in neural networks. Autoencoders are also successfully used for anomaly detection and novelty detection problems. **Autoencoder neural networks** come under the unsupervised learning category. Here the target values are set to be equal to the inputs. In other words, we want to learn the identity function. By doing so, we can gain a compact representation of the data.

The network is trained by minimizing the difference between input and output. A typical autoencoder architecture is a slight variant of the DNN architecture, where the number of units per hidden layer is progressively reduced until a certain point before being progressively increased, with the final layer dimension being equal to the input dimension. The key idea behind this is to introduce bottlenecks in the network and force it to learn a meaningful compact representation. The middle layer of hidden units (the bottleneck) is basically the dimension-reduced encoding of the input. The first half of the hidden layers is called the **encoder**, and the second half is called the **decoder**. The following depicts a simple autoencoder architecture. The layer named **z** is the representation layer here:
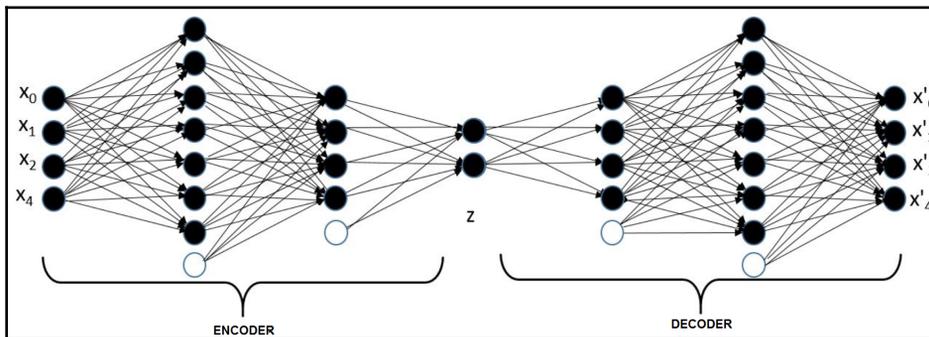
Source: https://cloud4scieng.org/manifold-learning-and-deep-autoencoders-in-science/

Variational autoencoders

Very deep autoencoders are hard to train and are prone to overfitting. There have been many developments that have improved how autoencoders are trained, such as generative pretraining with a **Restricted Boltzmann Machine** (**RBM**). The **variational autoencoders** (**VAEs**) are also generative models, and compared to other deep generative models, VAEs are computationally tractable and stable and can be estimated by the efficient backpropagation algorithm. They are inspired by the idea of variational inference in Bayesian analysis.

The idea of variational inference is as follows: given input distribution x, the posterior probability distribution over output y is too complicated to work with. So, let's approximate that complicated posterior, $p(y \mid x)$, with a simpler distribution, $q(y)$. Here, q is chosen from a family of distributions, Q, that best approximates the posterior. For example, this technique is used in training **latent Dirichlet allocation** (**LDAs**) (they do topic modeling for text and are Bayesian generative models). However, a key limitation of classical variational inference is the need for the likelihood and the prior to be conjugate in order to be tractably optimized. VAEs introduce the use of neural networks to output the conditional posterior (Kingma and Welling, 2013) and thus allow the variational inference objective to be optimized using **Stochastic Gradient Descent** (**SGD**) and backpropagation. This method is known as the **reparametrization trick**.

Given a dataset, X, VAE can generate new samples similar but not necessarily equal to those in X. Dataset X has N **Independent and Identically Distributed** (**IID**) samples of some continuous or discrete random variable, x. Let's assume that the data is generated by some random process, involving an unobserved continuous random variable, z. In the this example of a simple autoencoder, the variable z is deterministic and is a stochastic variable. Data generation is a two step process:

1. A value of z is generated from a prior distribution, $\varrho_\theta(z)$
2. A value of x is generated from the conditional distribution, $\varrho_\theta(x|z)$

So, *p(x)* is basically the marginal probability, calculated as:

$$\int p_\theta(x|z)p(z)dz$$

The parameter of the distribution, θ, and the latent variable, z, are both unknown. Here, x can be generated by taking samples from the marginal *p(x)*. Backpropagation cannot handle stochastic variable z or stochastic layer z within the network. Assuming the prior distribution, *p(z)*, is Gaussian, we can leverage the *location-scale* property of Gaussian distribution, and rewrite the stochastic layer as $z = \mu + \sigma\varepsilon$, where μ is the location parameter, σ is the scale, and ε is the white noise. Now we can obtain multiple samples of the noise, ε, and feed them as the deterministic input to the neural network.

Then, the model becomes an end-to-end deterministic deep neural network, as shown here:
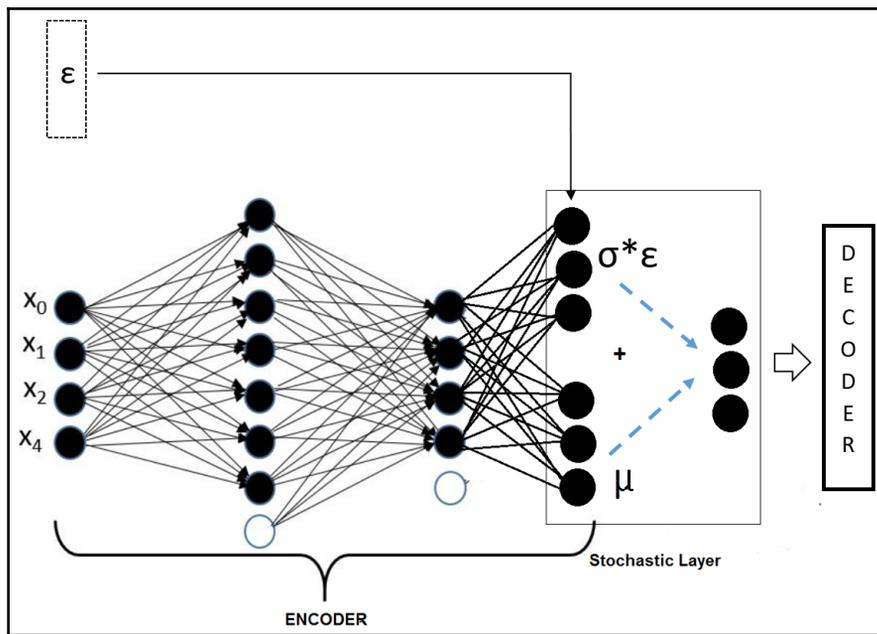
Here, the decoder part is same as in the case of the simple autoencoder that that we looked at earlier. What about the loss function for training this network? As this is a probabilistic model, the most straightforward method is to derive the loss function by the maximum likelihood of the marginal, $p(x)$. However, that function becomes computationally intractable to find. As such, we apply a variational inference technique to the lower bound, L. The marginal likelihood is derived, and then the loss function is derived by maximizing the lower bound, L. The theoretical details of this can be found in the paper by Kingma and their co-authors, *Auto-Encoding Variational Bayes*, ICLR, 2014 (`https://arxiv.org/abs/1312.6114`). VAEs are successfully used in various domains. For example, deep semantic hashing of text is done by VAEs, where a text document is converted into binary code. Also, similar documents have similar binary addresses. As such, these codes can be used for faster and more efficient retrieval, and also for the clustering and classification of documents.

Generative Adversarial Networks

Generative Adversarial Networks (**GANs**) have gained a lot of popularity since their first introduction in a 2014 NIPS paper by Ian Goodfellow and their co-authors (`https://arxiv.org/pdf/1406.2661.pdf`). Now we see applications of GANs in various domains. Researchers from Insilico Medicine proposed an approach to artificial drug discovery using GANs. They also found applications in image processing and video processing problems, such as image style transfers and **deep convolutional generative adversarial networks** (**DCGAN**).

As the name suggests, this is another type of generative model that uses neural networks. GANs have two main components—a generator neural network and a discriminator neural network. The generator network takes a random noise input and tries to generate a sample of data. The discriminator network compares the generated data with the real data and solves the binary classification problem of whether generated data is fake or not, using sigmoid output activation. Both the generator and the discriminator are constantly competing and trying to fool each other—that's the reason that GANs are also known as **adversarial networks**. This competition drives both the networks to improve their weights until the discriminator starts outputting a probability of 0.5; that is, until the generator starts generating real images. Both the networks are trained simultaneously by backpropagation. Here is a high-level structural diagram of a GAN:
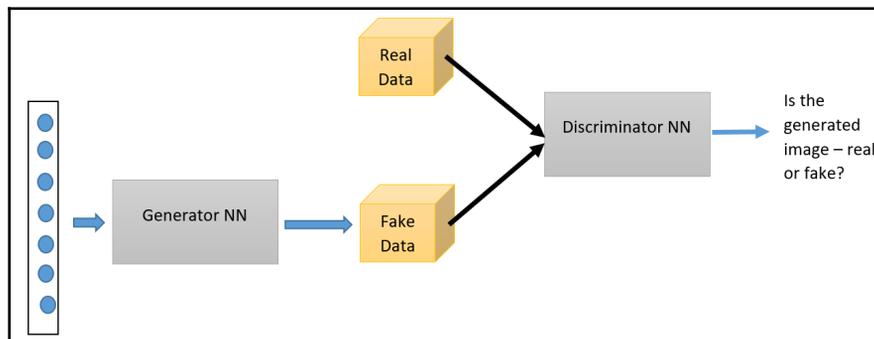
The loss function for training these networks can be defined as follows. Let *pdata* be the probability distribution of the data and *pg* be the generator distribution. *D(x)* represents the probability that *x* came from *pdata* and not from *pg*. *D* is trained to maximize the probability of assigning the correct label to both training examples and samples from *G*. Simultaneously, *G* is trained to minimize *log(1 − D(G(z)))*. So, *D* and *G* play a two-player minimax game with value function *V(D, G)*:

$$min_G \ max_D \mathrm{V}(D,G) = E_{x \sim p_{data}(x)}[log(D(x))] + E_{x \sim p_g(x)}[log(1 - D(G(x)))]$$

It can be proved that this minimax game has a global optimum for $p_g = p_{data}$.

The following is the algorithm for training a GAN with backpropagation to obtain the desired result:

```
for N epochs do:

    #update discriminator net first

    for k steps do:

        Sample minibatch of m noise samples {z(1), , . . . , z
        (m)} from noise prior pg(z).

        Sample minibatch of m examples {x(1), . . . , x(m)} from
        data generating distribution pdata(x).

        Update the discriminator by:
```

$$\nabla_{\theta_d} [\frac{1}{m} \sum_{i=1}^{m} [log(D(x^i)) + log(1 - D(G(z^i)))]]$$

```
    end for

        Sample minibatch of m noise samples {z(1) , . . . , z (m)}
        from noise prior pg(z).
        Update the generator by descending its stochastic gradient:
```

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [log(1 - D(G(z^i)))]$$

```
    end for
```

Text-to-image synthesis using the GAN architecture

Let's look at image generation from a textual description using a GAN. The following image shows the full architecture of such a GAN:

Generative Adversarial Text to Image Synthesis
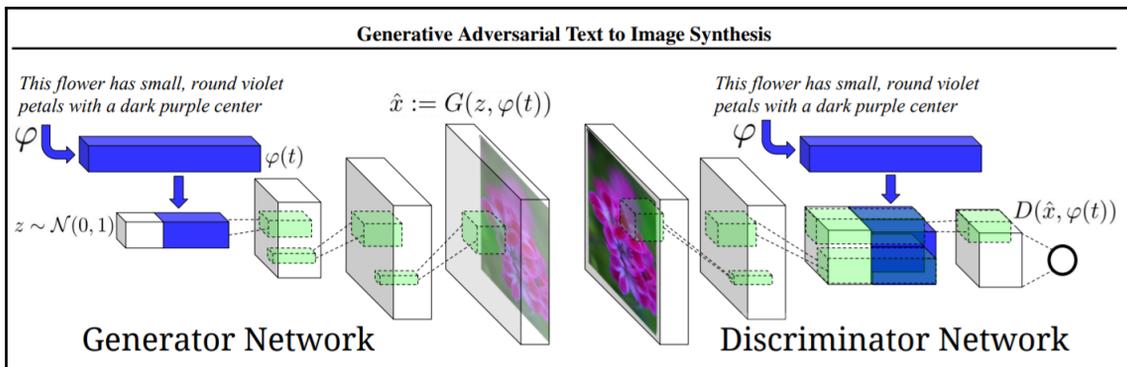
This is a type of conditional GAN. The generator network takes input text with a noise vector to generate the image. The generated image is conditioned on the input text. The image description is converted into a dense vector using embedding layer $\varphi(t)$. It's compressed using a fully connected layer and is then concatenated with the noise vector. The detector network is a CNN, and the architecture of the generator network uses deconvolution layers with the same filters as the ones used in CNN networks. Deconvolution is basically a transposed convolution—we'll discuss that later.

CNNs

CNNs are multilayered neural networks designed specifically for identifying shape patterns with a high degree of invariance to translation, scaling, and rotation in two-dimensional image data. These networks need to be trained in a supervised way. Typically, a labeled set of object classes, such as MNIST or ImageNet, is provided as a training set. The crux of any CNN model is the convolution layer and the subsampling/pooling layer. So, let's understand the operations performed in these layers in detail.

The convolution operator

The central idea behind CNNs is the mathematical operation of **convolution**, which is a specialized kind of linear operation. It's heavily used in a variety of domains, including physics, statistics, computer vision, and image and signal processing. To understand this, let's start with an example. A *noisy* laser sensor is tracking the position of a spaceship. To get a better estimate of the location of the spaceship, we can take an average of a few readings, giving more weights to recent observations. Let $x(t)$ represent the location in time, t, and let $w(t)$ be the weighting function.

The estimate of the position can be written as:

$$s(t) = \sum_{a=-\infty}^{\infty} \mathrm{x}(a)\mathrm{w}(t-a)$$

Here, the weight function, *w(t)*, is called the **kernel** of the convolution. We can compute a **simple moving average** (**SMA**) of the location sensor data using convolution. Let *m* be the window size of the SMA.

The kernel is defined as:

$$\mathrm{w}(t) = \begin{cases} 1/m & ,0 \le t \le m \\ 0 & otherwise \end{cases}$$

Here is a NumPy implementation of the SMA using convolution:

```
x = [1, 2, 3, 4, 5, 6, 7]
m = 3 #moving average window size
sma = np.convolve(x, np.ones((m,))/m, mode='valid')
#Outputs
#array([ 2., 3., 4., 5., 6.])
```

In deep learning, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are learned by the training algorithm. Although we have an infinite summation in the convolution formula, for practical implementation, the value of the weight function is non-zero only at a finite subset of values, as in the case of SMA. Hence, the summation in the formula becomes a finite summation. Convolution can be applied over more than one axis. If we have a two-dimensional image, *I*, and a two-dimensional smoothing kernel, *K*, the convoluted image would be calculated as follows:

$$\mathrm{S}(i,j) = \sum_{m}\sum_{n} I(m,n)K(i-m,j-n)$$

Alternatively, it could also be calculated as follows:

$$\mathrm{S}(i,j) = \sum_{m}\sum_{n} I(i-m,j-n)K(m,n)$$

The following figure explains how with a kernel of size 2 and stride 1, the convolution layer output is calculated:
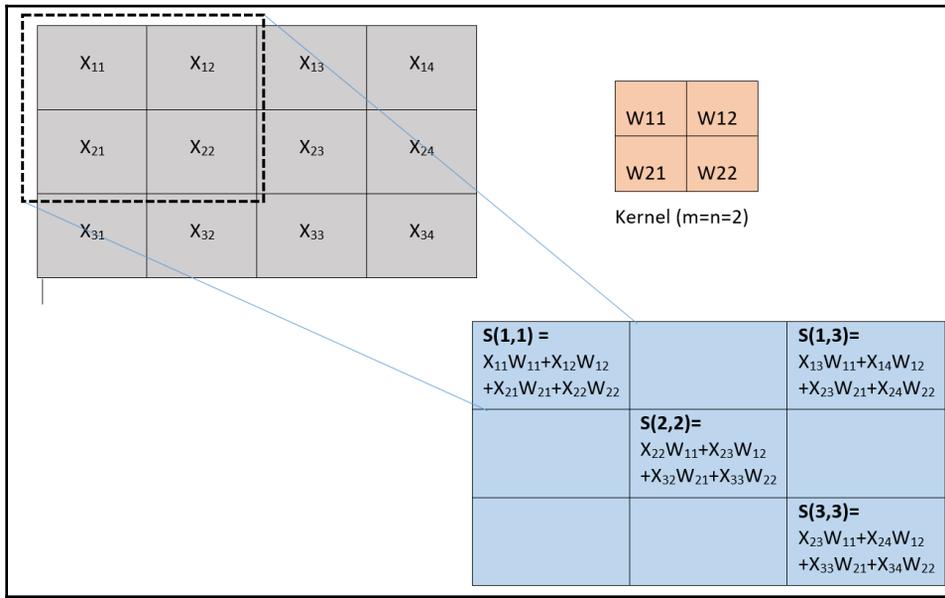
Stride and padding mode in convolution

The convolution kernel convolves around the input volume by shifting one column/row at a time. The amount by which the filter shifts is called the **stride**. In the preceding scenario, the stride was implicitly set to 1. If we shift the kernel by a stride of 2 (two columns or two rows), the number of output units shrinks:
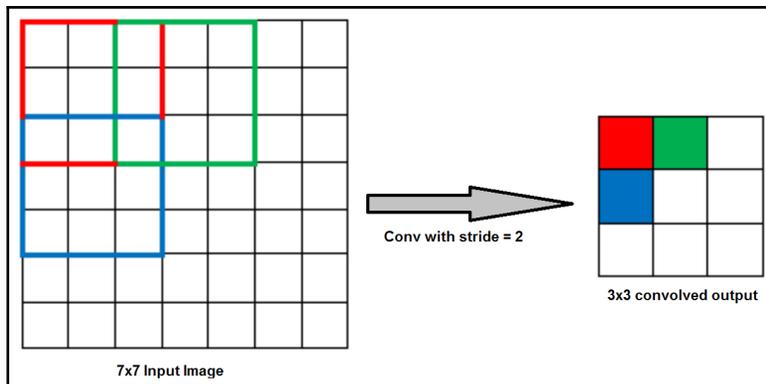
The convolution operator reduces the size of the input. If we want to preserve the size of the input, we need to pad zeros evenly around the input. For a two-dimensional image, this means adding a border of zero pixels around the four sides of the image. The thickness of the border—that is, the number of rows of pixels added—depends on the kernel size being applied. Any convolution operator implementation typically takes a mode argument that specifies the padding type. There are two such arguments:

- `SAME`: Specifies that the output size is to be the same as the input size. This requires the filter window to slip outside the input map, hence the need for padding.
- `VALID`: Specifies that the filter window stays at a valid position inside the input map, so the output size shrinks by `filter_size` minus one. No padding occurs.

In the preceding one-dimensional convolution code, we have set the mode as `VALID`, and thus no padding occurs. You may try with *same* padding.

The convolution layer

A convolution layer consists of three major stages, each of which poses some structural constraints over a multilayered network:

- **Feature extraction**: Each unit makes connections from a locally receptive field in the previous layer, thus forcing the network to extract local features. If we have a 32 x 32 image and the receptive field size is 4 x 4, then one hidden layer will be connected to 16 units in the previous layers, and we will have 28 x 28 hidden units in total. Thus, the input layer makes 28 x 28 x 16 connections to the hidden layer, and this is the number of parameters (weight on each connection) between these two layers. Had it been a fully connected dense hidden layer, there would be 32 x 32 x 28 x 28 parameters. So, we get a dramatic reduction in the number of parameters with this architectural constraint. Now, the output of this local linear activation is run through a nonlinear activation function, such as ReLU. This stage is sometimes called the **detector stage**. Once a feature detector is learned, the exact location of the feature in an unseen image is not important, as long as its position relative to other features is preserved. The synaptic weights associated with the receptive field of a hidden neuron is the kernel of the convolution.

- **Feature mapping**: The feature detector creates a feature map that is in the form of a plane (green plane shown as follows). To extract the different types of local features and have a richer representation of the data, several convolutions are performed in parallel to produce several feature maps, as shown here:
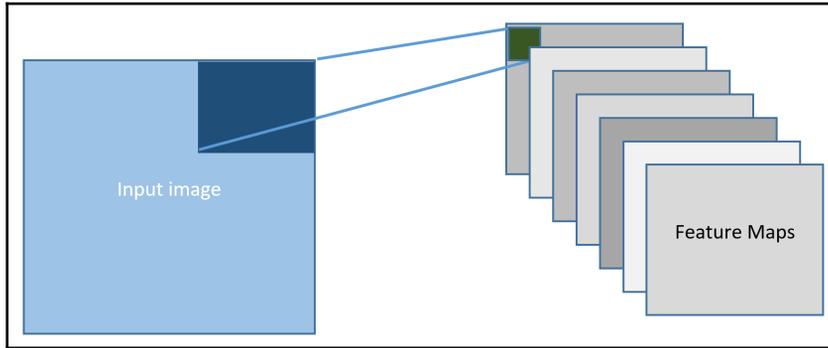
- **Subsampling by pooling**: This is done by a computational layer that subsamples the output of the feature detector by replacing the feature detector units at certain locations with summary statistics of the nearby units. The summary statistics can be maximums or averages. This operation reduces the sensitivity of the feature maps output to simple distortions, such as linear shifts and rotations; pooling introduces invariance:
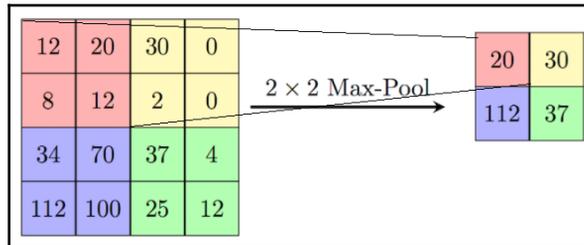
Combining all of these three stages gives us one complex layer in the CNN—each of the three stages are simple layers in their own right:
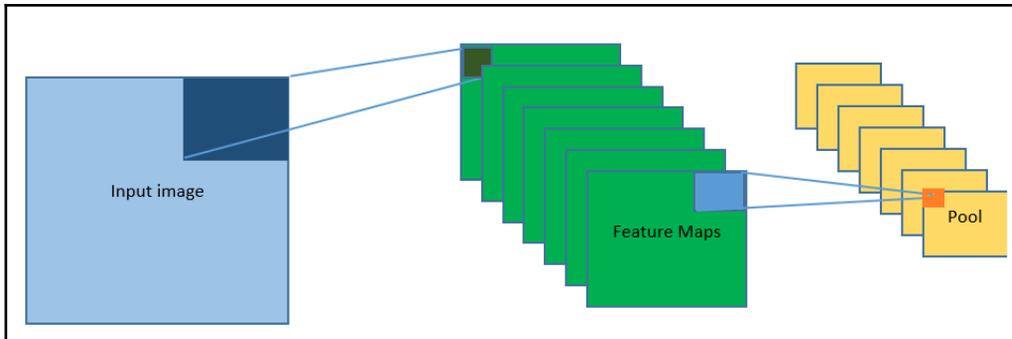
The pooled feature maps can be arranged in a volume by stacking them together side by side, as follows. Then we can again apply the next level of convolution to this. Now, the receptive field for a hidden unit in a single feature map will be a volume of neural units, as shown in the following figure. However, the same set of two-dimensional weights will be used across the depth. The depth dimension is typically made up of channels. If we have an RGB input image, then our input itself will have three channels. But, the convolutions are applied in two-dimensional, and the same weights are shared across all channels:
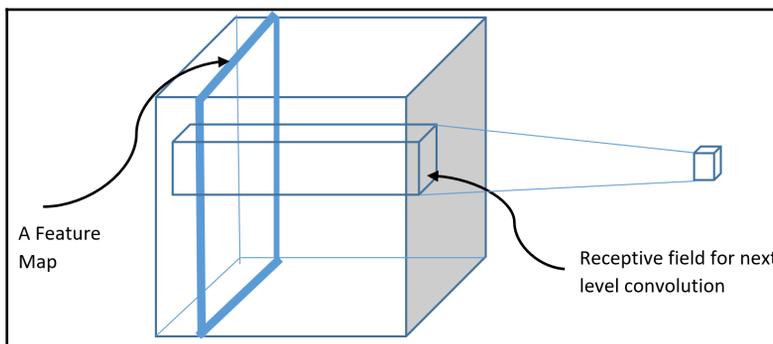
LeNet architecture

This is a pioneering seven-level convolutional network, designed by LeCun and their co-authors in 1998, that was used for digit classification. Later, it was applied by several banks to recognize handwritten numbers on cheques. The lower layers of the network are composed of alternating convolution and max pooling layers.

The upper layers are fully connected, dense MLPs (formed of hidden layers and logistic regression). The input to the first fully connected layer is the set of all the feature maps of the previous layer:
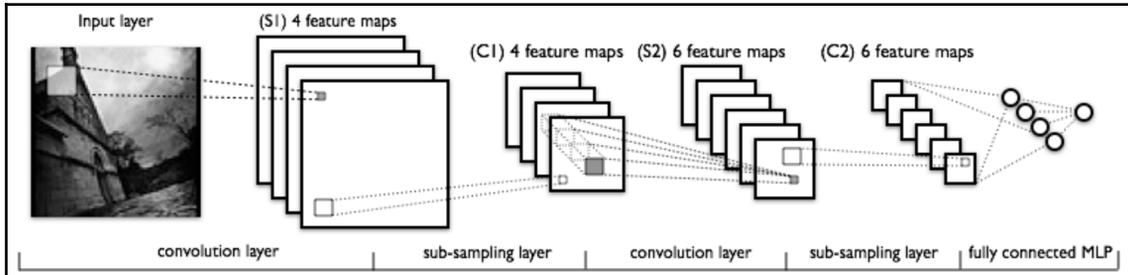
After the successful application of CNNs for digit classification, researchers focused on building more complex architectures that can classify images from ImageNet. ImageNet is an image database organized according to the WordNet hierarchy (currently only that of the nouns), in which each node of the hierarchy is depicted by hundreds or thousands of images. The ImageNet project runs an annual software contest, **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**), which evaluates algorithms for object detection and image classification on a large scale. The evaluation criteria is either the top-five/top score. These are calculated by taking the predictions of the CNN from the final dense softmax layer. If the target label is one of the top-five predictions (the five predictions with the highest probabilities), then it's considered a success. The top-five score is computed by taking the times a predicted label (in the top 5) matched the target label and dividing that by the number of images evaluated.

The top score is calculated in a similar way.

AlexNet

In 2012, AlexNet significantly outperformed all the prior competitors and won the ILSVRC by reducing the top-5 error to 15.3%, compared to the runner-up with 26%. This work popularized the application of CNNs in computer vision. AlexNet has a very similar architecture to that of LeNet, but has more filters per layer and is deeper. Also, AlexNet introduces the use of stacked convolution, instead of always using alternative convolution pooling. A stack of small convolutions is better than one large receptive field of convolution layers, as this introduces more non-linearities and fewer parameters.

Suppose we have three 3 x 3 convolution layers on top of each other (with non-linearity or pooling layer in between). Here, each neuron on the first convolution layer has a 3 x 3 view of the input volume. A neuron on the second convolution layer has a 3 x 3 view of the first convolution layer, and hence a 5 x 5 view of the input volume. Similarly, a neuron on the third convolution layer has a 3 x 3 view of the second convolution layer, and hence a 7 x 7 view of the input volume. Clearly, the number of parameters for a 7 x 7 receptive field is a factor of 49, compared to three stacked 3 x 3 convolutions with a factor of 3 x (3 x 3) = 27 parameters.

ZFNet

The ILSVRC 2013 winner was a CNN from Matthew Zeiler and Rob Fergus. It became known as **ZFNet**. It improved on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller, going from 11 x 11 stride 4 in AlexNet to 7 x 7 stride 2 in ZFNet. The intuition behind this was that a smaller filter size in the first convolution layer helps to retain a lot of the original pixel information. Also, AlexNet was trained on 15 million images, while ZFNet was trained on only 1.3 million images:
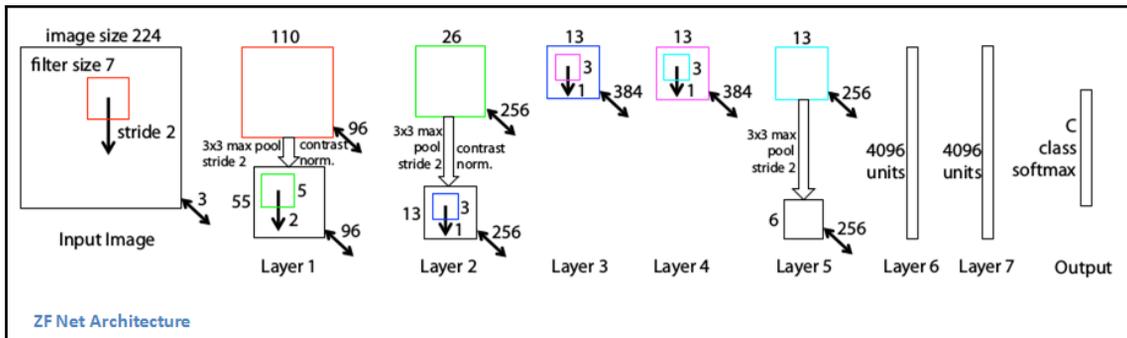
GoogLeNet (inception network)

The ILSVRC 2014 winner was a convolutional network called **GoogLeNet** from Google. It achieved a top-5 error rate of 6.67%! This was very close to human-level performance. The runner up was the network from Karen Simonyan and Andrew Zisserman known as **VGGNet**. GoogLeNet introduced a new architectural component using a CNN called the **inception layer**. The intuition behind the inception layer is to use larger convolutions, but also keep a fine resolution for smaller information on the images.

So, we can convolve in parallel with different kernel sizes, starting from 1 x 1 to a bigger one, such as 5 x 5, and the outputs are concatenated to produce the next layer:
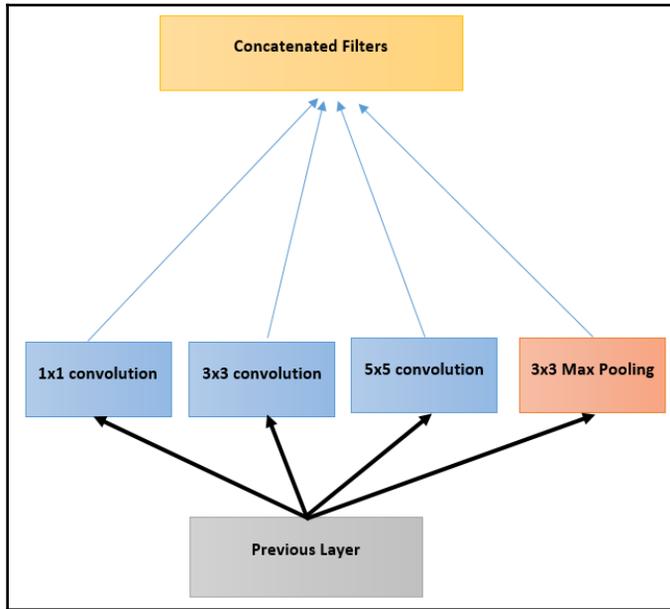
Clearly, adding more layers explodes the parameter space. To control this, a dimensionality-reduction trick is used. Note that the 1 x 1 convolution basically does not reduce the spatial dimensions of the image. But, we can reduce the number of feature maps with 1 x 1 filters and reduce the depth of the convolution layer, as shown in the following diagram:
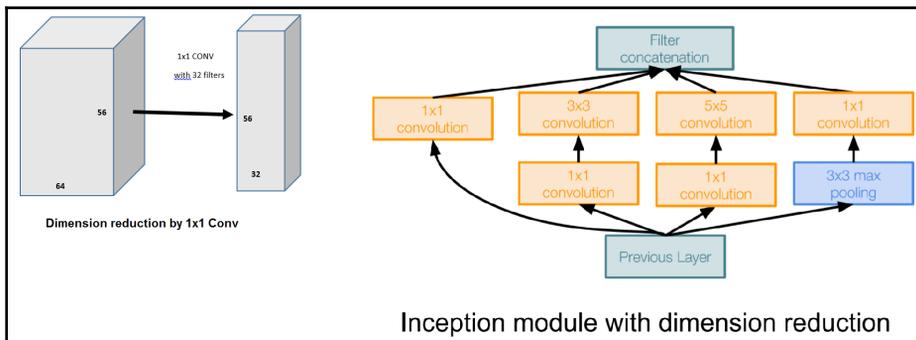
The following diagram describes the full GoogLeNet architecture:
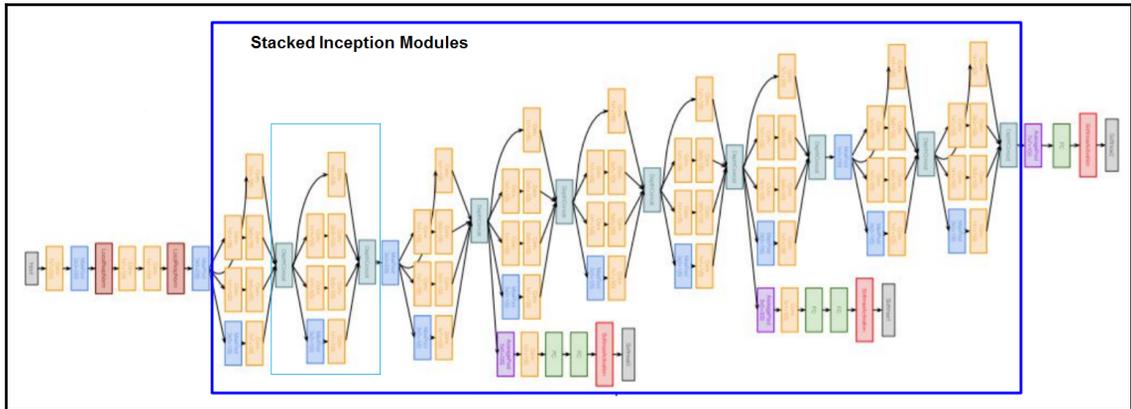
VGG

Researchers from the Oxford Visual Geometry Group, or the **VGG** for short, developed the VGG network, which is characterized by its simplicity, using only 3 x 3 convolutional layers stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. At the end, two fully connected layers, each with 4,096 nodes, are then followed by a softmax layer. The only preprocessing done to the input is the subtraction of the mean RGB value, computed on the training set, from each pixel.

Pooling is carried out by max pooling layers, which follow some of the convolution layers. Not all the convolution layers are followed by max pooling. Max pooling is performed over a 2 x 2 pixel window, with a stride of 2. ReLU activation is used in each of the hidden layers. The number of filters increases with depth in most VGG variants. The 16-layered architecture VGG-16 is shown in the following diagram. The 19-layered architecture with uniform 3 x 3 convolutions (VGG-19) is shown along with ResNet in the following section. The success of VGG models confirms the importance of depth in image representations:
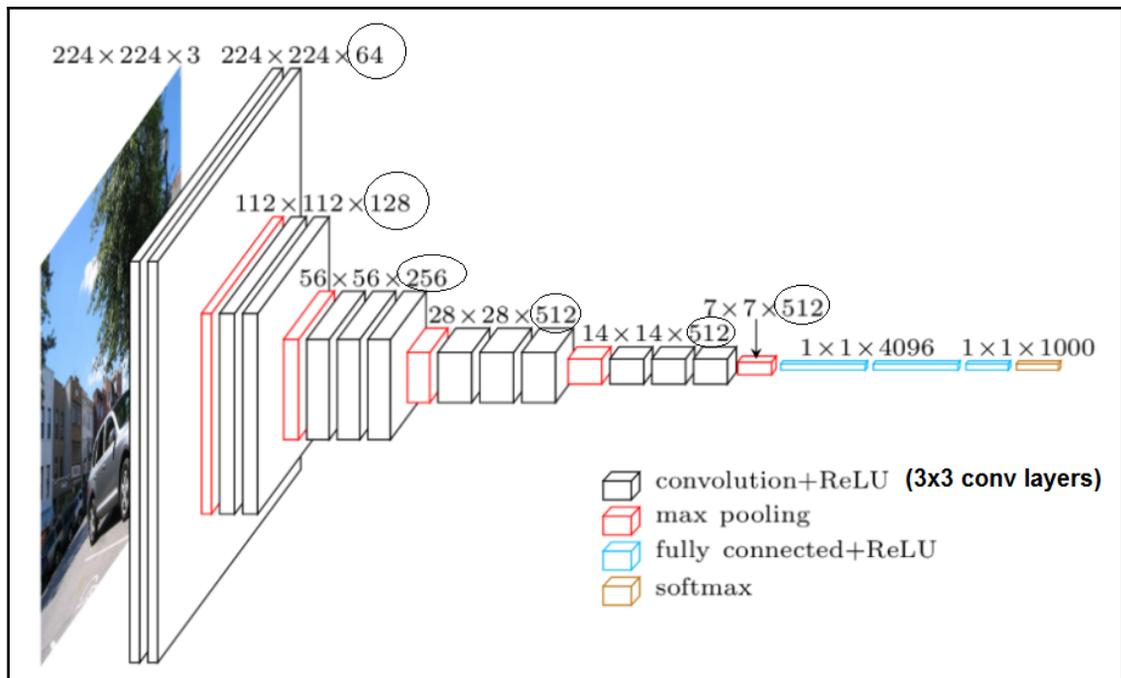
VGG-16: Input RGB image of size 224 x 224 x 3, the number of filters in each layer is circled

Residual Neural Networks

In ILSVRC 2015, a novel CNN architecture with *skip connections* and *batch normalization* was introduced by Kaiming He and their co-authors from Microsoft Research Asia called the **Residual Neural Network** (**ResNet**). With this, they were able to train a neural network with 152 layers (eight times deeper than the VGG network) while still having lower complexity than the VGG network. It achieves a top-5 error rate of 3.57%, which beats human-level performance on this dataset.

The main idea in this architecture is as follows. Instead of hoping that a set of stacked layers would directly fit a desired underlying mapping, $H(x)$, they tried to fit a residual mapping. More formally, they let the stacked set of layers learn the residual $R(x) = H(x) - x$, with the true mapping later being obtained by a skip connection. The input is then added to the learned residual, $R(x) + x$.

Also, batch normalization is applied right after each convolution and before activation:
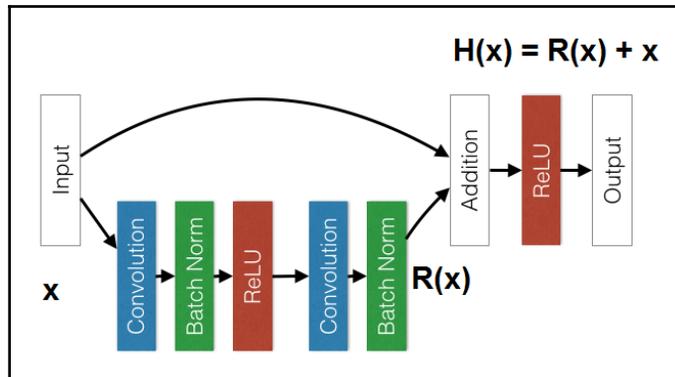
One building block of residual network

Here is the full ResNet architecture compared to VGG-19. The dotted skip connections show an increase in dimensions; hence, for the addition to be valid, no padding is done. Also, increases in dimensions are indicated by changes in color:
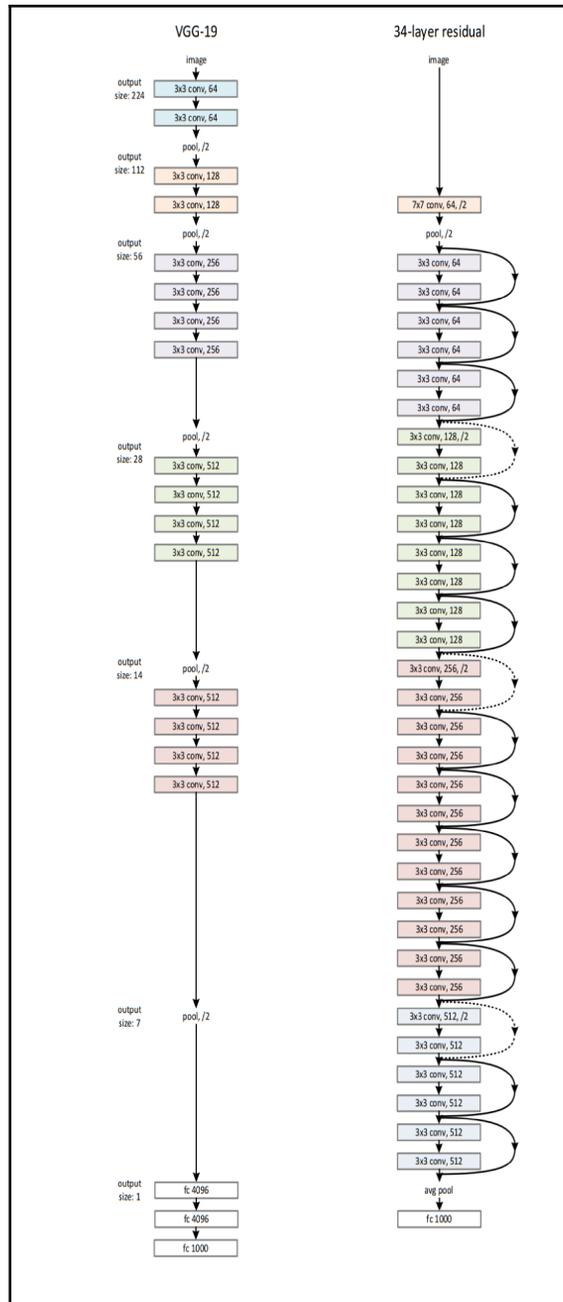
All the variants of CNN models that we have talked about so far are available as pretrained models in Keras and TensorFlow. We will be using them heavily in our transfer learning applications. Here is the Keras code snippet used to load various VGG models. More can be found here `https://keras.io/applications/`:

```
from keras.applications.vgg16 import VGG16
model = VGG16()
print(model.summary())
```

Capsule networks

We have discussed how various CNN architectures evolved, we and have looked at their successive improvements. Can we now apply CNNs for more advanced applications, such as **Advanced Driver Assistance Systems** (**ADAS**) and self-driving cars? Can we detect an obstacle, a pedestrian, and other overlapping objects on roads in real-world scenarios and in real time? Maybe not! We are still not quite there. In spite of the tremendous success of CNNs in ImageNet competitions, CNNs still have some severe limitations that restrict their applicability to more advanced, real-world problems. CNNs have poor translational invariance and lack information about orientation (or *pose*).

Pose information refers to three-dimensional orientation relative to the viewer, but also to lighting and color. CNNs do have trouble when objects are rotated or when lighting conditions are changed. According to Hinton, CNNs cannot do *handedness* detection at all; for instance, they can't tell a left shoe from a right shoe, even if they are trained on both. One of the reasons for these limitations of CNNs is the use of max pooling, which is a crude way to introduce invariance. By crude invariance, we mean the output of max pooling does not change much if the image is *slightly* shifted/rotated. We actually need more than invariance—we need **equivariance**; that is, invariance under **symmetric transformation** of images.

Edge detectors are the first layer in CNNs, and perform much the same function as the visual cortex system in the human brain. The difference between the brain and a CNN occurs in the higher levels. Efficiently routing low-level visual information to higher-level information, such as objects in various poses and colors, or at various scales and velocities, is believed to be done by cortical micro columns—which Hinton names *capsules*. This *routing* mechanism makes the human visual system more powerful than the CNN.

Capsule Networks (**CapsNets**) make two fundamental changes to the CNN architecture—firstly, they replace the scalar-output feature detectors of CNNs with vector-output capsules; secondly, they use max pooling with routing-by-agreement. Here is a simple CapsNet architecture:
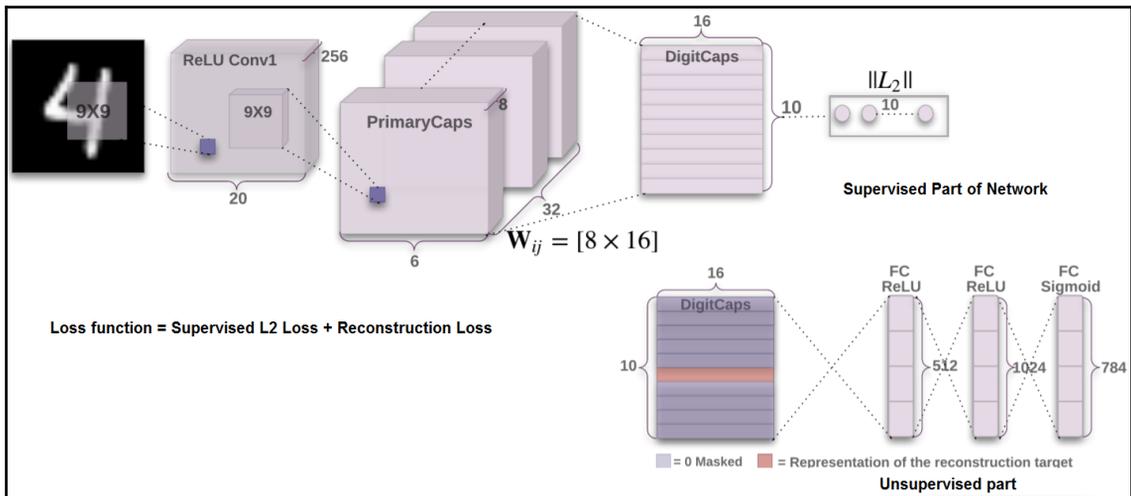
This is a shallow architecture to be trained on MNIST data (28 x 28 handwritten digit images). This has two convolutional layers. The first convolution layer has 256 feature maps with 9 x 9 kernels (stride =1) and ReLu activation. So, each feature map is (28-9+1) x (28-9+1), or 20 x 20. The second convolution layer again has 256 feature maps with 9 x 9 kernels (stride =2) and ReLu activation. Here each feature map is 6 x 6, 6=((20-9)/2+1). This layer is reshaped—or, rather, the feature maps are regrouped into 32 groups, with each group having 8 feature maps (256=8 x 32). The grouping process aims to create feature vectors each of size 8. To represent pose, a vector representation is a more natural representation. The grouped feature maps from the second layer is called the **primary capsule** layer. We have (32 x 6 x 6) eight-dimensional capsule vectors where each capsule contains eight convolutional units with a 9 x 9 kernel and a stride of 2. The final capsule layer (**DigitCaps**) has one sixteen-dimensional capsule per digit class (10 classes), and each of these capsules receives input from all the capsules in the primary capsule layer.

The length of the output vector of a capsule represents the probability that the entity represented by the capsule is present in the current input. The length of the capsule vector is normalized and kept between 0 and 1. Also, a squashing function is used on the norm of the vector such that short vectors get shrunk to almost zero length and long vectors get shrunk to a length slightly below 1.

The squashing function is :

$$f(x) = \frac{x}{1 + x^2}$$

Where x is the norm of a vector and hence $x > 0$ (see the following plot):

$$v_j = \frac{||s_j||^2}{1 + ||s_j||^2} \frac{s_j}{||s_j||}$$

v_j is the vector output of capsule j and s_j is its total input
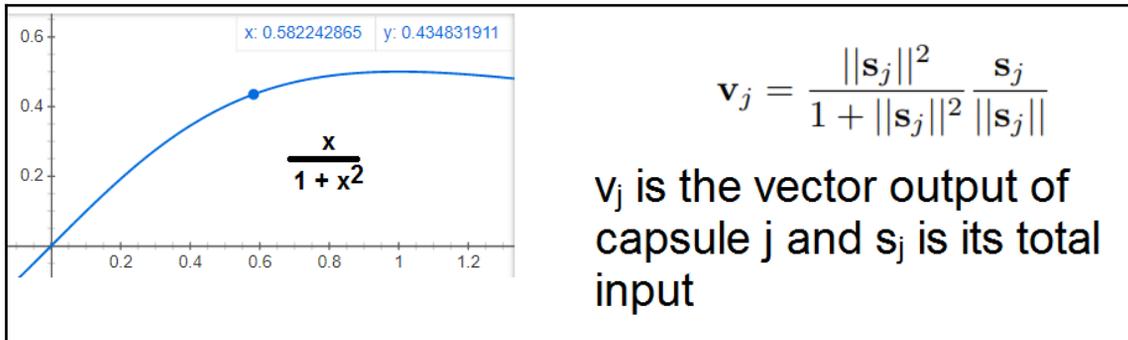
W_{ij} is a weight matrix between each u_i, $i \in (1, 32 \times 6 \times 6)$ in primary capsules and v_j, $j \in (1, 10)$ in DigitCaps. Here, $\hat{u}_{j|i} = W_{ij}u_i$ is called the prediction vector and is like a transformed (rotated/translated) input capsule vector, u_i. The total input to a capsule, s_j, is a weighted sum over all prediction vectors from the capsules in the layer below. These weights, c_{ij}, add up to 1 and are called **coupling coefficients** by Hinton. At the beginning, its assumed that log prior probabilities that capsule i should be coupled with parent capsule j is the same for all, i, j, and is denoted by b_{ij}. So, the coupling coefficients can be calculated by this well-known softmax transform:

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})}$$

These coupling coefficients are iteratively updated along with the weights of the network by an algorithm called **routing-by-agreement**. In short, it does the following: if the primary capsule i's prediction vector has a large scalar product with the output of possible parent j, coupling coefficient b_{ij} is increased for that parent and decreased for other parents.

The full routing algorithm is given here:
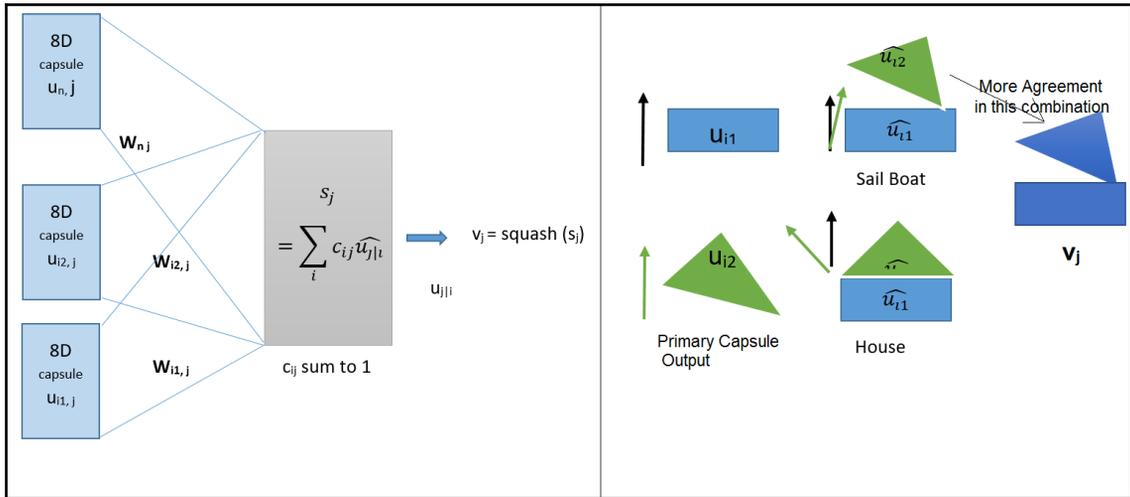
The left side—shows how all the primary capsules are connected to a digit capsule with the weight matrix W_{ij}. Also, it depicts how the coupling coefficients are calculated and how the sixteen-dimensional output of DigitCaps is calculated by the non-linear squashing function. On the right side—suppose primary capsules capture two basic shapes: a triangle and a rectangle in the input image. Aligning them after rotation gives either a **House** or a **Sail Boat**, based on the amount of rotation done. It's clear that, with minimal or almost no rotation, the two objects combine to form a sail boat. That is, the two primary capsules are more aligned to form a boat than a house. Thus, the routing algorithm should update the coupling coefficient for $b_{i,\,boat}$:

```
procedure routing (ûⱼ|ᵢ, r, l):
  for all capsule i in layer l and capsule j in layer (l + 1): bij <- 0
for r iterations do:
  for all capsule i in layer l: ci <- softmax (bi)
```
$$s_j \leftarrow \sum_i c_{ij} \hat{u_{j|i}}$$
```
  for all capsule j in layer (l + 1):
  for all capsule j in layer (l + 1): vj <- squash (sj)
  for all capsule i in layer l and capsule j in layer (l + 1): bᵢⱼ ← bᵢⱼ + ûⱼ|ᵢ • vⱼ
return vj
```

Finally, we need a proper loss function to train this network. Here, *margin loss for digit existence* is used as the loss function. It also considers the case of overlapping digits. A separate margin loss, L_k, for each digit capsule, *k*, is used to allow the detection of multiple overlapping digits. L_k looks at the length of the capsule vector—for a digit of class *k*, the length of the kth capsule vector should be maximum compared to others:

$$L_k = T_k max(0, m^+ - ||v_k||)^2 + \lambda(1 - T_k)max(0, ||v_k|| - m^-)^2$$

Here, $T_k = 1$, if the kth digit is present. $m^+ = 0.9$ and $m^- = 0.1$. The λ is used for down-weighting of the loss for absent digit classes. Along with L_ks, an image reconstruction error loss is used as a regularization of the network. As depicted in the CapsNet architecture, the output of the digit capsule is fed into a decoder consisting of three fully connected layers. The sum of squared differences between the outputs of the logistic units and the original image pixel intensities are minimized. The reconstruction loss is scaled down by a factor of 0.0005 so that it does not dominate the margin loss during training.

> The TensorFlow implementation of CapsNet is available here: `https://github.com/naturomics/CapsNet-Tensorflow`.

Recurrent neural networks

An **recurrent neural Network** (**RNN**) is specialized for processing a sequence of values, as in x(1), . . . , x(t). We need to do sequence modeling if, say, we wanted to predict the next term in the sequence given the recent history of the sequence, or maybe translate a sequence of words in one language to another language. RNNs are distinguished from feedforward networks by the presence of a feedback loop in their architecture. It is often said that RNNs have memory. The sequential information is preserved in the RNNs hidden state. So, the hidden layer in the RNN is the memory of the network. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps.

We will explain this later:

$$H(t) = ReLu(W * x(t) + U * H(t-1))$$

RNN with single hidden unit

RNN with distributed hidden state
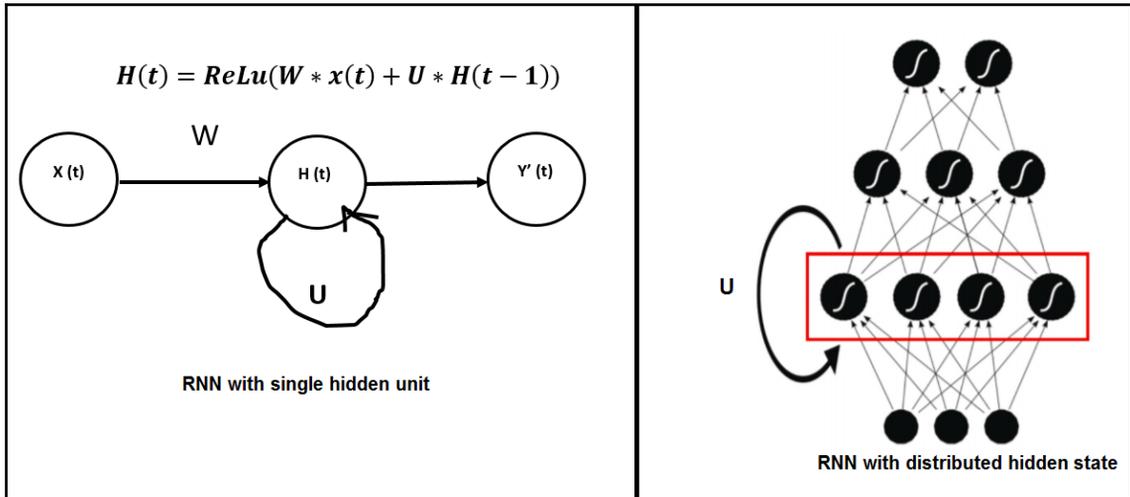
By unrolling the feedback loop in the network, we can obtain a feedforward network. For example, if our input sequence is of length 4, then the network can be unrolled as follows. After unrolling the same set of weights, U, V, and W are shared across all the steps, unlike in a traditional DNN, which uses different parameters at each layer. So, we are actually performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn. Now, to learn these shared weights, we need a loss function. At each time step, we can compare the network output, $y(t)$, with the target sequence, $s(t)$, and get an error, $E(t)$. So, the total error is $E = \sum_{s} E_s$:
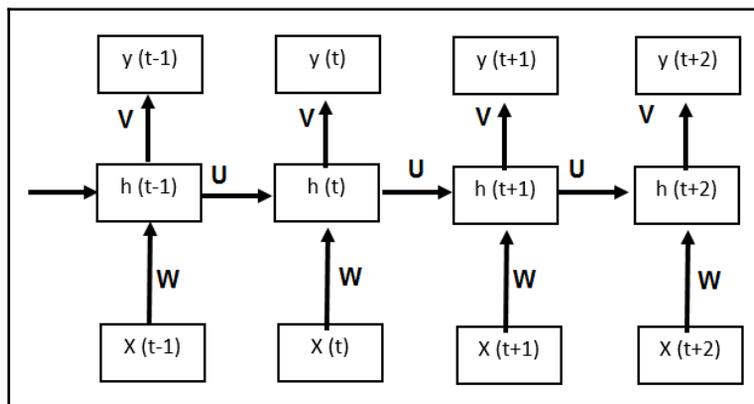
Let's take a look at the total error derivatives that we will need for learning weights with gradient-based optimization algorithms. We have $h_t = U\phi(h_{t-1}) + W x_t$, Φ being the non-linear activation and $y_t = V\phi(h_t)$.

Now, $\dfrac{\partial E}{\partial U} = \sum_{t=1}^{S} \dfrac{\partial E_t}{\partial U}$, where by chain rule we have $\dfrac{\partial E_t}{\partial U} = \sum_{k=1}^{t} \dfrac{\partial E_t}{\partial y_t} \dfrac{\partial y_t}{\partial h_t} \dfrac{\partial h_t}{\partial h_k} \dfrac{\partial h_k}{\partial U}$.

Here, the Jacobian $\dfrac{\partial h_t}{\partial h_k}$ that is, the derivative of layer *t*, with regards to a previous layer, *k*—is itself a product of Jacobian $\prod_{s=k+1}^{t} \dfrac{\partial h_s}{\partial h_{s-1}}$.

Using the preceding equation of h$_t$,we have $\dfrac{\partial h_s}{\partial h_{s-1}} = U^T diag(\phi'(h_{s-1}))$. So, the norm of the Jacobian $\dfrac{\partial h_t}{\partial h_k}$ is given by the product $\prod_{s=k+1}^{t} \left\| \dfrac{\partial h_s}{\partial h_{s-1}} \right\|$. If the quantities $\left\| \dfrac{\partial h_s}{\partial h_{s-1}} \right\|$ are less than one, then over a long sequence of, say, 100 steps, the product of these norms will tend to zero. Similarly, if the norm is greater than one, then the product over long sequences will grow exponentially. These problems are called **vanishing gradients** and **exploding gradients** in RNNs. So, in practice, RNNs cannot have very long-term memories.

LSTMs

RNNs start losing historical context over time in the sequence, and hence are hard to train for practical purposes. This is where LSTMs come into the picture! Introduced by Hochreiter and Schmidhuber in 1997, LSTMs can remember information from really long sequence-based data and prevent issues such as the vanishing gradient problem. LSTMs usually consist of three or four gates, including input, output, and forget gates.

The following diagram shows a high-level representation of a single LSTM cell:
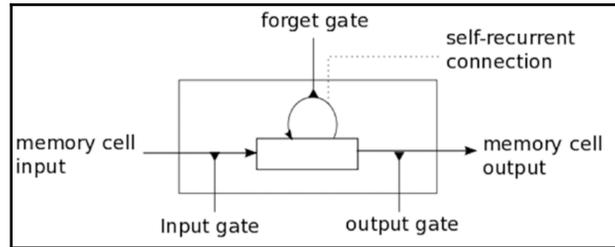
The input gate can usually allow or deny incoming signals or inputs to alter the memory cell state. The output gate usually propagates the value to other neurons as necessary. The forget gate controls the memory cell's self-recurrent connection to remember or forget previous states as necessary. Multiple LSTM cells are usually stacked in any deep learning network to solve real-world problems, such as sequence prediction. In the following diagram, we compare the basic structures of RNNs and LSTMs:
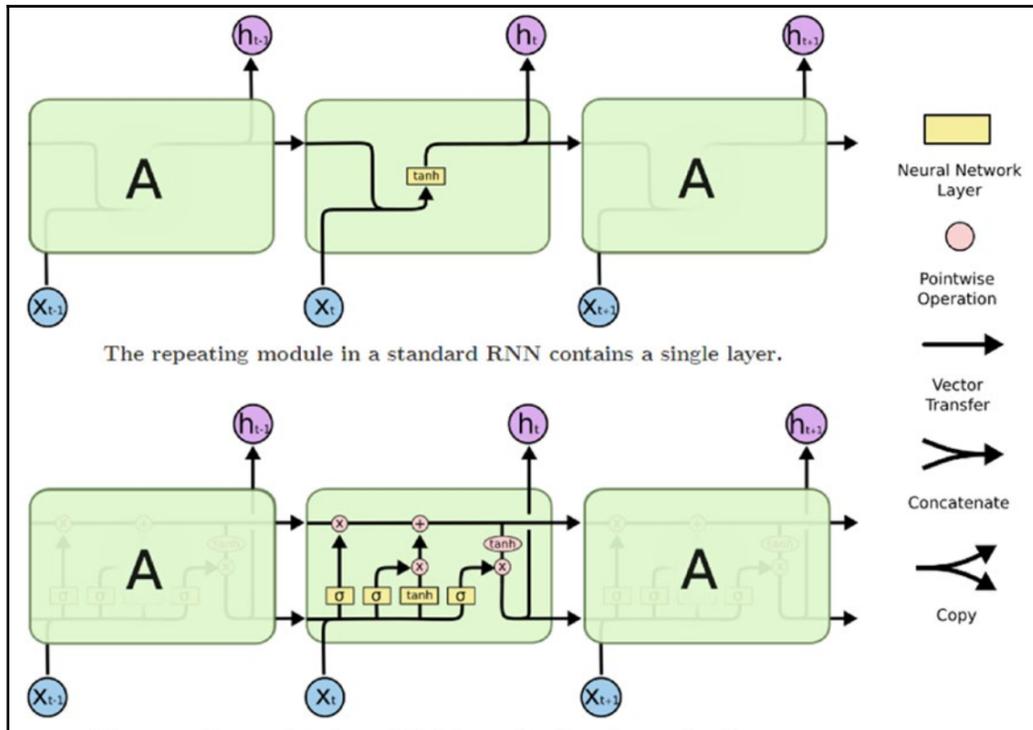
A detailed architecture of an LSTM cell and information flow is presented in the following diagram. Let t indicate one time step; C, the cell states; and h, the hidden states. The LSTM cell has the ability to remove or add information to the cell state, by structures called *gates*. The gates i, f, and o represent the input, forget, and output gates, respectively, and each of them is modulated by the sigmoid layer, which outputs numbers from zero to one, controlling how much of the output from these gates should pass. Thus this helps in protecting and controlling the cell state:
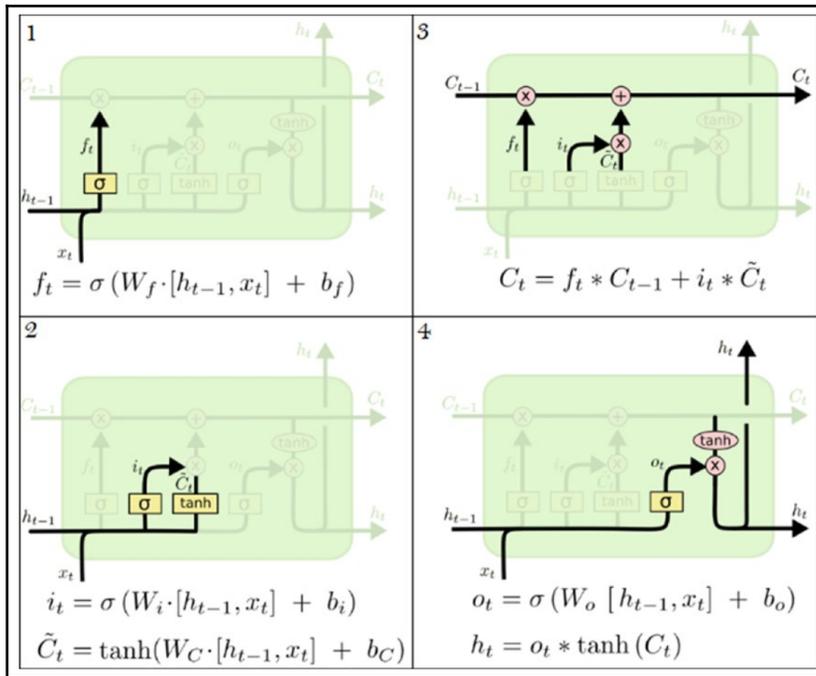
The information flow through LSTMs has four steps:

1. **Decide what information to throw away from the cell state**: This decision is made by a sigmoid layer called the **forget gate layer**. An affine transformation is applied on h_t, x_{t-1}, and the output of that is passed through a sigmoid squashing function to get a number between zero and one for each number in the cell state, C_{t-1}. One indicates that memory should be kept and zero indicates that memory should be erased completely.

2. **Decide what new information to write to memory**: This is a two-step process. Firstly, a sigmoid layer called the **input gate layer**, i_t, is used to decide which locations to write the information in. Next, a tanh layer creates the new candidate information to be written.

3. **Update memory state**: The old memory state is multiplied by f_t, erasing things that were determined forgettable. Then, the new state information computed in *step 2* is added after scaling them by i_t.

4. **Output memory state**: The final output of the cell state depends on the current input and the updated cell start. Firstly, a sigmoid layer is used to decide which parts of the cell state we're going to output. Then, the cell state is passed through tanh and multiplied by the output of the sigmoid gate.

I recommended that you check out Christophers's blog at `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` for a more detailed explanation of the LSTM steps. Most of the diagrams we have looked at here are taken from this.

LSTMs can be used for sequence forecasting as well as sequence classification. For example, we can forecast future stock prices. Also, we can use LSTMs to build classifiers that will predict whether an input signal from some health-monitoring system is a fatal or non-fatal signal—a binary classifier. We can even build a text document classifier with LSTMs. The sequence of words would go as input to the LSTM layer, and the hidden state of the LSTM would be connected to a dense softmax layer as a classifier.

Stacked LSTMs

If we want to learn about the hierarchical representation of sequential data, a stack of LSTM layers can be used. Each LSTM layer outputs a sequence of vectors rather than a single vector for each item of the sequence, which will be used as an input to a subsequent LSTM layer. This hierarchy of hidden layers enables a more complex representation of our sequential data. Stacked LSTM models can be used for modeling complex multivariate time series data.

Encoder-decoder – Neural Machine Translation

Machine translation is a sub-field of computational linguistics, and is about performing translation of text or speech from one language to another. Traditional machine translation systems typically rely on sophisticated feature engineering based on the statistical properties of text. Recently, deep learning has been used to solve this problem, with an approach known as **Neural Machine Translation** (**NMT**). An NMT system typically consists of two modules: an encoder and a decoder.

It first reads the source sentence using the *encoder* to build a thought vector: a sequence of numbers that represents the sentence's meaning. A *decoder* processes the sentence vector to emit a translation to other target language. This is called an encoder-decoder architecture. The encoders and decoders are typically forms of RNN. The following diagram shows an encoder-decoder architecture using stacked LSTMs. Here the first layer is an embedding layer used to represent the words in the source language by dense real vectors. The vocabulary is predefined for both the source language and the target language. Words not in the vocabulary are denoted by a fixed word, *<unknown>*, and are represented by a fixed embedding vector. The input to this network is the source sentence at first, then an end of sentence marker *<s>*, which indicates the transition from the encoding to the decoding mode, and then the target sentence is fed in:

Source: https://www.tensorflow.org/tutorials/seq2seq
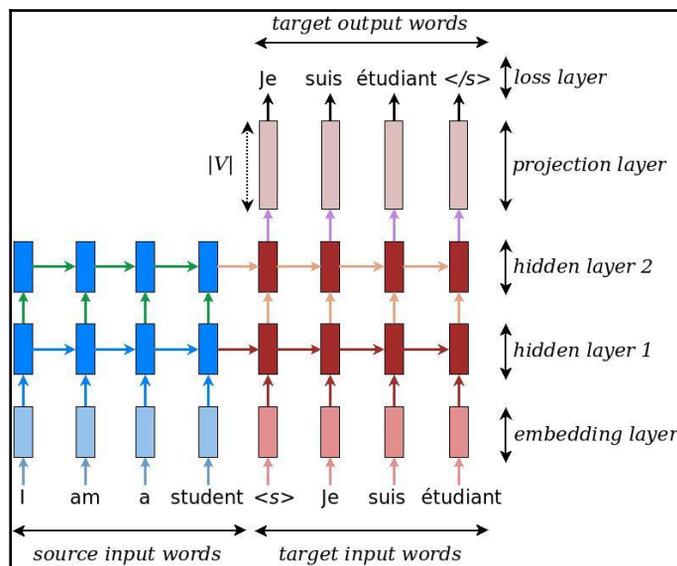
The input embedding layer is followed by two stacked LSTM layers. Then, the projection layer turns the top hidden states into logit vectors of dimension V (the vocabulary size for the target language). Here cross-entropy loss is used to train the network with backpropagation. We see in training mode that both the source and target sentences are input to the network. In inference mode, we will have only the source sentence. In that case, the decoding can be done by several methods, such as greedy decoding, attention mechanisms combined with greedy decoding, and beam search decoding. We will cover the first two methods here:
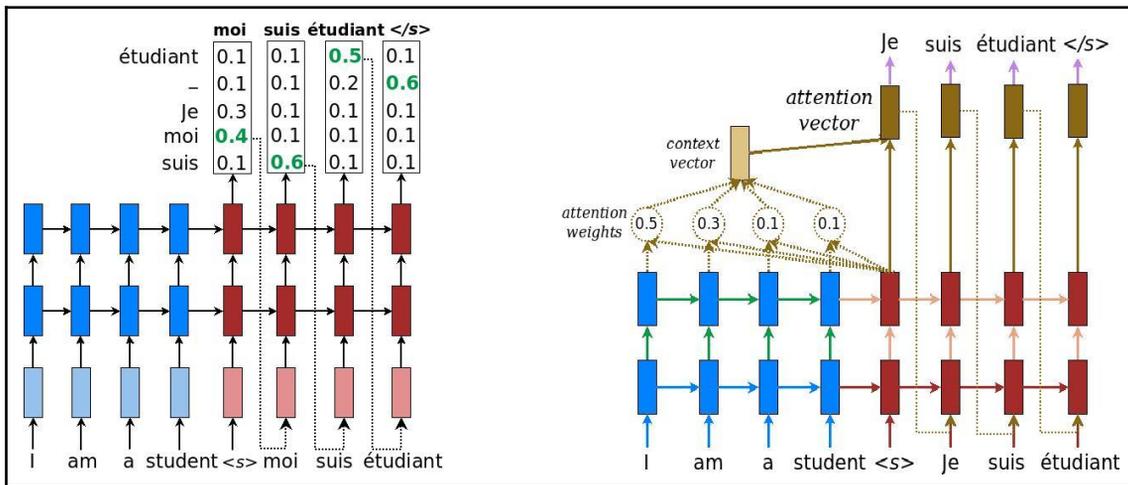
In the greedy decoding scheme (see the left-hand diagram of the two preceding diagrams), we choose the most likely word, depicted by maximum logit value as the emitted word, before feeding it back to the decoder as the input. This decoding process continues until the end-of-sentence marker, *</s>*, is produced as an output symbol.

The context vector generated by the source sentence's end-of-sentence marker must encode everything we need to know about the source sentence. It must fully capture its meaning. For long sentences, this means we need to store very long-term memory. Researchers have found that reversing the source sequence or feeding the source sequence twice helps the network remember things better. For languages like French and German, which are quite similar to English, reversing the input makes sense. For Japanese, the last word of a sentence may be highly predictive of the first word in an English translation. So, here reversing can degrade the quality of translation. So, an alternative solution is to use an **attention mechanism** (shown in the right-hand diagram of the preceding two diagrams).

Now, instead of trying to encode the full source sentence into a fixed-length vector, the decoder is allowed to *attend* to different parts of the source sentence at each step of the output generation. Hence, we represent the attention-based context vector, c_t, for the tth target language word as a weighted sum of all the previous source hidden states:

$$c_t = \sum_s \alpha_{ts} h_s$$

. The attention weights are $\alpha_{ts} = \dfrac{exp(score(h_t, h_s))}{\sum_{s'=1}^{S} exp(score(h_t, h_{s'}))}$, and the *score* is calculated as follows: $score(h_t, h_s) = h_t W h_s$.

Here W is a weight matrix that will be jointly learned with the RNN weights. This score function is called **Luong's** multiplicative style score. There are a few other variants of this score as well. Finally, the **attention vector, a_t**, is calculated by combining the context vector with the current target hidden state as follows: $a_t = f(c_t, h_t) = tanh(W_c[c_t; h_t])$.

The attention mechanism is like read-only memory, where all previous hidden states of source are stored and then read at the time of decoding. The source code for NMT in TensorFlow is available here: `https://github.com/tensorflow/nmt/`.

Gated Recurrent Units

Gated Recurrent Units (**GRUs**) are related to LSTMs, as both utilize different ways of gating information to prevent the vanishing gradient problem and store long-term memory. A GRU has two gates: a reset gate, *r*, and an update gate, *z*, as shown in the following diagram. The reset gate determines how to combine the new input with the previous hidden state, h_{t-1}, and the update gate defines how much of the previous state information to keep. If we set the reset to all ones and update gate to all zeros, we arrive at a simple RNN model:

Initially, for $t = 0$, the output vector is $h_0 = 0$.

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)$$

Variables

- x_t: input vector
- h_t: output vector
- z_t: update gate vector
- r_t: reset gate vector
- W, U and b: parameter matrices and vector
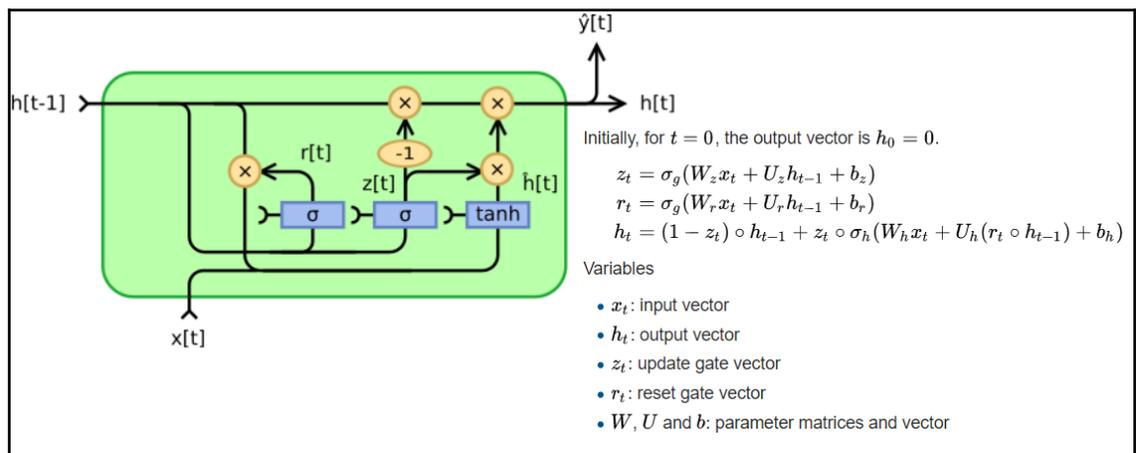
GRUs are relatively new, and their performance is on a par with that of LSTMs, but they are computationally more efficient because of a simpler structure and fewer parameters. Here are a few structural differences between LSTMs and GRUs:

- A GRU has two gates, while an LSTM has three gates. GRUs don't have the output gate that is present in LSTMs.
- GRUs don't have additional internal memory, C_t, other than the hidden state.
- In GRUs, non-linearity (tanh) is *not* applied when computing the output.

If enough data is available, using LSTMs is advisable as the greater expressive power of LSTMs may lead to better results.

Memory Neural Networks

Most machine learning models are not able to read and write to a long-term memory component, nor are they able to combine old memories seamlessly with inference. RNNs and their variants, such as LSTMs, do have a memory component. However, their memory (encoded by hidden states and weights) is typically too small, and is not like the large arrays of blocks that we find in modern computers (in the form of RAM). They try to compress all past knowledge into one dense vector—the memory state. This may be very restrictive for a complex application such as a virtual assistance or **question-answering** (**QA**) system where the long-term memory effectively acts as a (dynamic) knowledge base, and the output is a textual response. To address this problem, **Memory Neural Networks** (**MemNNs**) were developed by the Facebook AI Research group. The central idea of MemNNs is to combine the successful learning strategies developed in the deep learning literature for inference with a memory component that can be read and written to just like RAM. Also, the model is trained to learn how to operate effectively with the memory component. A memory network consists of memory, **m**, an indexed array of objects (for example, vectors or arrays of strings) and four components that are to be learned, **I, G, O** and **R**:

- **I**: An input feature map, **I**, which converts the incoming input into the internal feature representation.
- **G**: A *generalization* component, **G**, which updates old memories given new input. This is called **generalization**, as there is an opportunity for the network to compress and generalize its memories at this stage for some desired future use.
- **O**: An output feature map, **O**, which produces a new output in the feature representation space given the new input and the *current memory state*.

- **R**: A *response* component, **R**, which converts the output into the response format desired—for example, a textual response or an action:
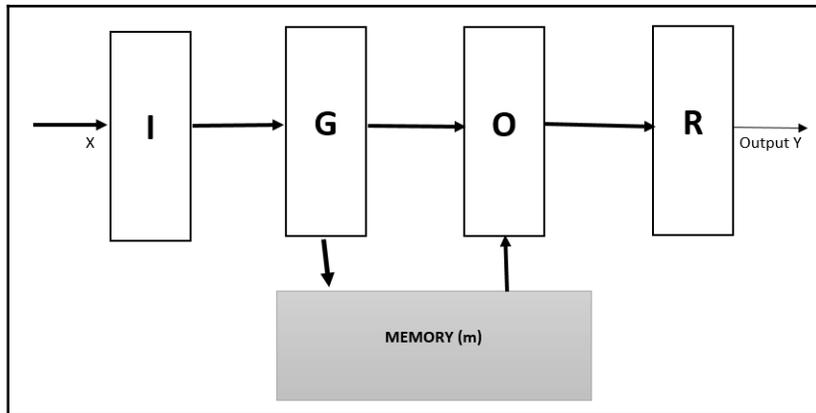
When the components I,G,O, and R are *neural networks*, then resulting system is called a **MemNN**. Let's try to understand this with an example QA system. The system will be given a set of facts and a question. It will output the answer to this question. We have the following six textual facts and a question, Q: *where is the milk now?*:

- Joe went to the kitchen
- Fred went to the kitchen
- Joe picked up the milk
- Joe traveled to the office
- Joe left the milk
- Joe went to the bathroom

Note that only some subsets of the statements contain information needed for the answer, and the others are essentially irrelevant distractions. We will represent this in terms of the MemNN modules I, G, O, and R. The module **I** is a simple embedding module that converts the text into binary bag-of-words vectors. The text is stored in the next available memory slot in its original form, and thus the **G** module is simple. The vocabulary of words used in given facts is V = {Joe, Fred, traveled, picked, left, went, office, bathroom, kitchen, milk}, once stopwords are removed. Now, here is the memory state after all text is stored:

Memory Slot#	Joe	Fred	...	Office	bathroom	kitchen	milk
1	1	0		0	0	1	0
2	0	1		0	0	1	0
3	1	0		0	0	0	1

4	1	0	1	0	0	0
5	1	0	0	0	0	1
6	1	0	0	1	0	0
7						

The **O** module produces output features by finding k supporting memories given question q. For k = 2, the highest scoring supporting memory is retrieved with:

$$o1 = \underbrace{arg\ max}_{i=1,\ldots,N} s_0(q, m_i)$$

Where s_0 is a function that scores the match between an input, q, and m_i. o1 is the index to the memory, m, with the best match. Now, using the query and the first retrieved memory, we can retrieve the next memory, m_{o2}, which is close to both of them:

$$o2 = \underbrace{arg\ max}_{i=1,\ldots,N} s_0([q, m_{o1}], m_i)$$

The combined query and memory results are o = [q, m_{o1}, m_{o2}] = [where is the milk now, Joe left the milk., Joe traveled to the office.]. Finally, module **R** needs to produce a textual response, r. The R module can output one-word answers, or maybe an RNN module generating a complete sentence. For a single-word response, let s_r be another function that scores the match between [q,m_{o1},m_{o2}] and a word, w. So, the final response, r, is the word office:

$$r = \underbrace{arg\ max}_{i=1,\ldots,N} s_r([q, m_{o1}, m_{o2}], V)$$

This model is hard to train end-to-end using backpropagation, and requires supervision at each module of the network. There is a slight modification of this that is effectively a continuous version of memory nets called an **End-To-End Memory Network** (**MemN2N**). This network can be trained by backpropagation.

MemN2Ns

We start with a query: where is the milk now? It is encoded with bags of words using a vector of size *V*. In the simplest case, we use embedding *B (d x V)* to convert the vector to a word embedding of size *d*. We have **u=embeddingB(q)**:
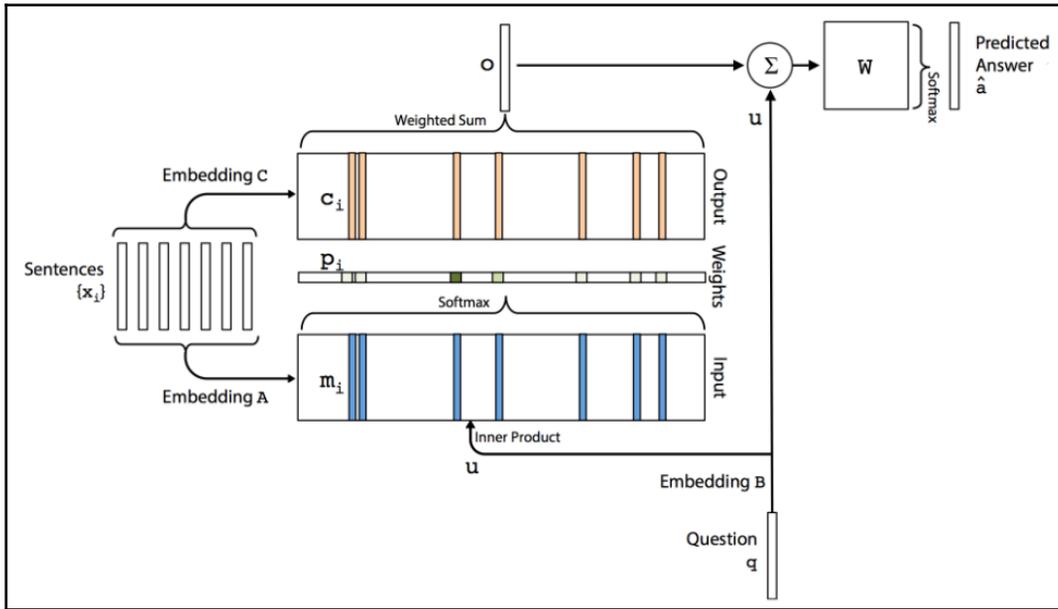
The input sentences x1, x2, ... , and xi are stored in memory by using another embedding matrix, A (d x Vd x V), with the same dimension as B **mi=embeddingA(x$_i$)**. The similarity between each embedded query, *u*, and each memory, m_i, is computed by taking the inner product followed by a softmax: **p$_i$= softmax(u^{T}m$_i$)**.

The output memory representation is as follows: each xi has a corresponding output vector, c_i, which can be represented by another embedding matrix, *C*. The response vector from the memory, *o*, is then a sum over the c_i, weighted by the probability vector from the input:

$$o = \sum_i p_i c_i$$

Finally, the sum of *o* and *u* is multiplied with a weight matrix, W (V x d). The result is passed to a softmax function to predict the final answer:

$$\hat{a} = softmax(W(o + u))$$

A TensorFlow implementation of MemN2N is available here: `https://github.com/carpedm20/MemN2N-tensorflow`.

Neural Turing Machine

A **Neural Turing Machine** (**NTM**) is inspired by the **Turing Machine** (**TM**): the fundamental mathematical model of computation that defines an abstract machine. A TM can manipulate symbols on a strip of tape according to a table of rules. Given any computer algorithm, a TM is capable of simulating that algorithm's logic. The machine positions its head over a cell and reads or writes a symbol there. Thereafter, based on the rules defined, it can move left or right or even halt the program.

An NTM architecture contains two basic components: a neural network controller and a memory. The following diagram shows a high-level representation of the NTM architecture:
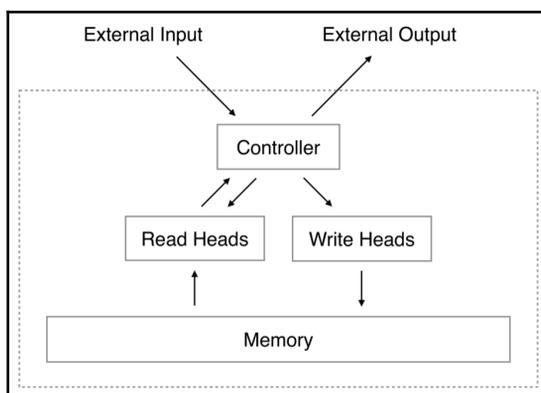
The controller interacts with the external world using input and output vectors. Unlike a standard neural network, here the controller also interacts with a memory matrix using selective read and write operations. The memory is a real-valued matrix. The memory interactions are differentiable end-to-end, making it possible to optimize them using gradient descent. NTMs can learn simple algorithms, such as copying, sorting, and associative recall from input and output examples. Also, unlike a TM, an NTM is a differentiable computer that can be trained by gradient descent, yielding a practical mechanism for learning programs.

The controller can be modeled by a LSTM, which has its own internal memory that can complement the larger memory in the matrix. The controller may be compared to the CPU in a computer, and the memory matrix can be compared to the RAM of a computer.

The read and write heads select the portions of memory to read or write to. They can be modeled by a hidden layer in the neural network—possibly a softmax layer—such that they can be treated as weights on the external memory cell that add up to 1. Also, note that the number of parameters of the model is controlled by the heads, and it does not grow with memory capacity.

Selective attention

The controller output is used to figure out which location of memory to read from or write to. This is defined by a set of weights spread over all memory locations, which add up to 1. The weights are defined by the following two mechanisms. The idea is to give the controller several different modes of reading or writing to memory, corresponding to different data structures:

- **Content based**: Compares the key **k** output of the controller with all the memory locations using a similarity measure, say, cosine similarity (S), and then all the distances are normalized by softmax to get weights that add up to 1:

$$w_i = \frac{exp(\beta S(k, M_i))}{\sum_j exp(\beta S(k, M_j))}$$

In this, $\beta \geq 1$ is called the sharpness parameter, and controls the focus on a particular location. It also gives the network a way to decide how precise it wants memory location access to be. It's like the fuzziness coefficient in fuzzy c-means clustering.

- **Location based**: The location-based addressing mechanism is designed for simple iterations across memory locations. For example, if the current weighting focuses entirely on a single location, a rotation of 1 would shift the focus to the next location. A negative shift would move the weighting in the opposite direction. The controller outputs a shift kernel, *s* (say, a softmax on [-n,n]), which is convolved with the memory weighting calculated previously to produce a shifted memory location, as in the following diagram. The shift is circular; that is, it wraps around the borders. The following diagram is a heat map representation of memory—the darker shades represent more weights:

$$\widetilde{w_i} = \sum_j \boldsymbol{w_j} s(i - j)$$
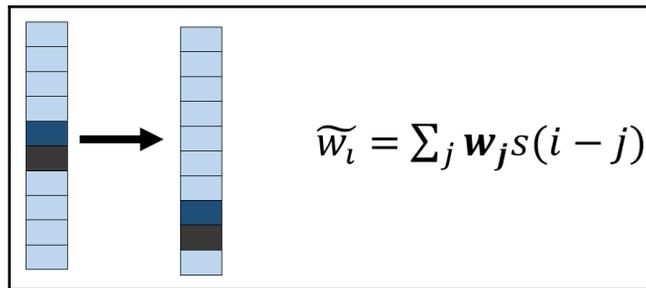
Before applying rotational shift, the weight vector given by content addressing is combined with the previous weight vector, w_{t-1}, as follows: $w_t^g = g_t w_t^{content} + (1 - g_t) w_{t-1}$.
Here, g_t is a scalar *interpolation gate* in the range (0, 1), emitted by the controller head. If g_t=1, the weighting from the previous iteration is ignored.

Read operation

Let $\mathbf{M}_t$ be the contents of the N x M memory matrix at time *t*, where *N* is the number of memory locations, and *M* is the vector size at each location. The read head at time *t* is given by vector $\mathbf{w}_t$, where:

$$\sum_{i=1}^{N} w_{ti} = 1$$

The length *M* read vector $\mathbf{r}_t$ is defined as a convex combination of the row vectors $\mathbf{M}_t(\mathbf{i})$ in memory:

$$r_t = \sum_i w_{ti} M_{ti}$$

Write operation

Each write head receives an *erase vector*, $\mathbf{e}_t$, and an *add* vector, $\mathbf{a}_t$, to reset and write to memory, just like an LSTM cell, as follows: $M_t(i) \leftarrow M_t(i) [1 - e_{t(i)} w_t(i)] + w_t(i) a_t(i)$.

Here is the pseudo code for the preceding operations:

```
mem_size = 128 #The size of memory
mem_dim = 16 #The dimensionality for memory
shift_range = 1 # defining shift[-1, 0, 1]

## last output layer from LSTM controller: last_output
## Previous memory state: M_prev
def Linear(input_, output_size, stddev=0.5):
  '''Applies a linear transformation to the input data: input_
  implements dense layer with tf.random_normal_initializer(stddev=stddev)
  as weight initializer
  '''
def get_controller_head(M_prev, last_output, is_read=True):
  k = tf.tanh(Linear(last_output, mem_dim))
  # Interpolation gate
  g = tf.sigmoid(Linear(last_output, 1)
  # shift weighting
  w = Linear(last_output, 2 * shift_range + 1)
    s_w = softmax(w)
  # Cosine similarity
  similarity = smooth_cosine_similarity(M_prev, k) # [mem_size x 1]
  # Focusing by content
  content_focused_w = softmax(scalar_mul(similarity, beta))

  # Convolutional shifts
  conv_w = circular_convolution(gated_w, s_w)

  if is_read:
    read = matmul(tf.transpose(M_prev), w)
    return w, read
  else:
    erase = tf.sigmoid(Linear(last_output, mem_dim)
    add = tf.tanh(Linear(last_output, mem_dim))
    return w, add, erase
```

The full TensorFlow implementation of NTMs is available here: `https://github.com/carpedm20/NTM-tensorflow`. NTM algorithms can learn to copy—they can learn an algorithm to copy a sequence of random numbers. The following shows how NTMs use the memory read and write heads and shift them to implement a copy algorithm:
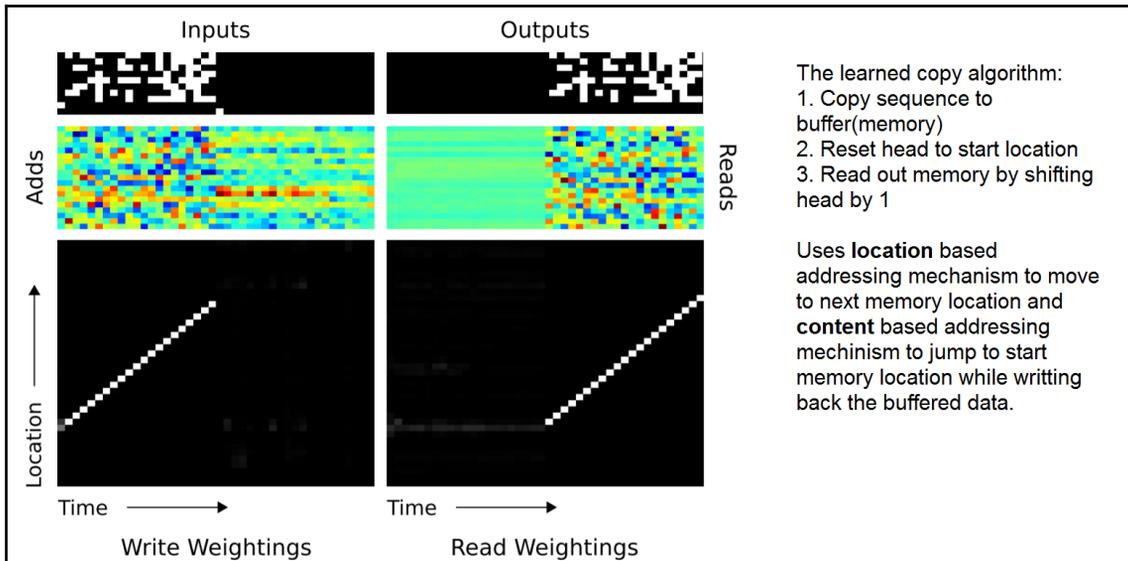
Similarly, NTMs can efficiently learn sorting algorithms from data, given a set of random sequences and corresponding sorted sequences.

The attention-based neural network model

We have discussed attention-based models for machine translation. The good part about attention-based models is that they give a way to interpret the model and understand how it works. The attention mechanism is a form of memorizing previous internal states. It's like internal memory. Unlike typical memory, the memory access mechanism here is soft, which means that the network retrieves a weighted combination of all memory locations, not a value from a single discrete location. Soft memory access makes it feasible to train the network by backpropagation. Attention-based architectures are used beyond machine translation in automatically generating captions for images.

This work was published in 2016 in the paper *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, by Kelvin Xu and it's co-authors (`https://arxiv.org/abs/1502.03044`). Here, looking at the attention weights, we see that as the model generates each word, its attention changes to reflect the relevant parts of the image. A TensorFlow implementation of this attention model is available here `https://github.com/yunjey/show-attend-and-tell`:
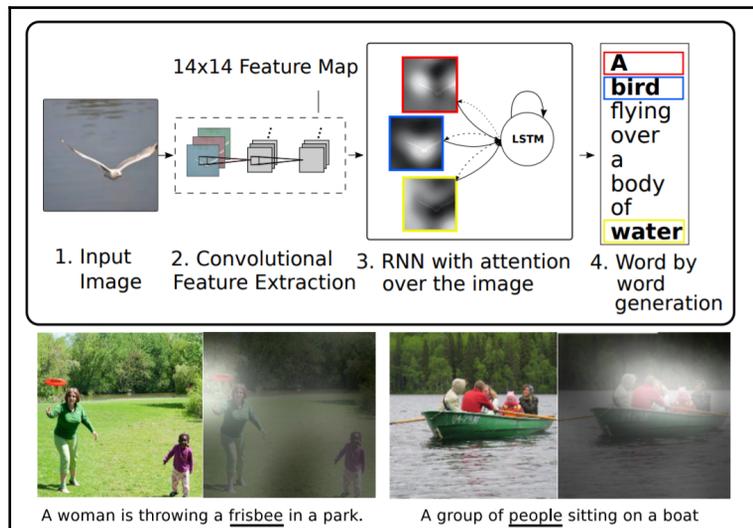
Summary

This chapter covered various advances in neural network architectures and their application to a varied set of real-world problems. We discussed the need for these architectures and why a simple deep multilayer neural network won't sufficiently solve all sorts of problems, given that it has great expressive power and a rich hypothesis space. Many of these architectures discussed will be used in later chapters when covering transfer learning use cases. References to the Python code for almost all the architectures is provided. We have also tried to clearly explain some of the very recent architectures, such as CapsNet, MemNNs, and NTMs. We will be frequently referring back to this chapter while walking you through the transfer learning use cases.

The next chapter will introduce the concepts of transfer learning.

4
Transfer Learning Fundamentals

I am still learning

– Michelangelo

Humans have an inherent ability to transfer knowledge across tasks. What we acquire as knowledge while learning about one task, we utilize in the same way to solve related tasks. The more related the tasks, the easier it is for us to transfer, or cross-utilize, knowledge. Machine learning and the deep learning algorithms, discussed so far, have been traditionally designed to work in isolation. These algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones. In this chapter, we will introduce the concept of transfer learning and concentrate on aspects of the context of deep learning. This chapter will cover the following topics:

- Introduction to transfer learning
- Transfer learning strategies
- Transferring knowledge with deep learning
- Types of deep transfer learning
- Challenges of transfer learning

Introduction to transfer learning

Traditionally, learning algorithms are designed to tackle tasks or problems in isolation. Depending upon the requirements of the use case and data at hand, an algorithm is applied to train a model for the given specific task. Traditional **machine learning** (**ML**) trains every model in isolation based on the specific domain, data and task as depicted in the following figure:
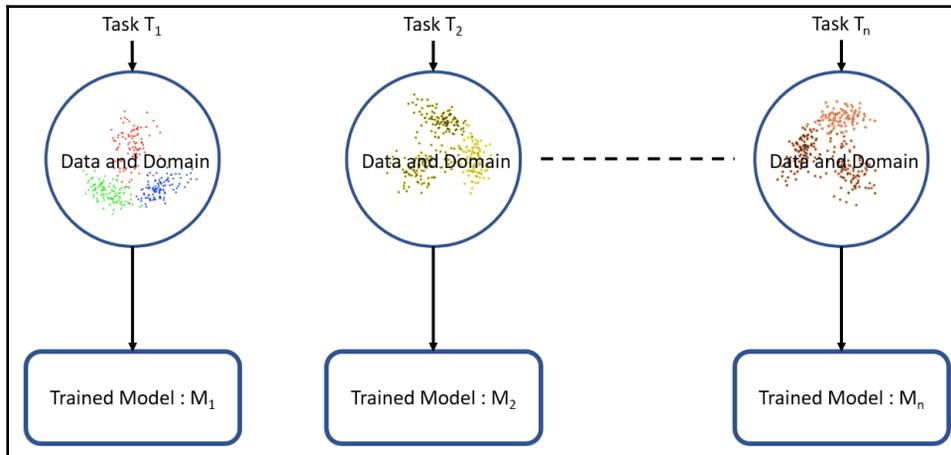
Traditional machine learning

Transfer learning takes the process of learning one step further and more inline with how humans utilize knowledge across tasks. Thus, transfer learning is a method of reusing a model or knowledge for another related task. Transfer learning is sometimes also considered as an extension of existing ML algorithms. Extensive research and work is being done in the context of transfer learning and on understanding how knowledge can be transferred among tasks. However, the **Neural Information Processing Systems** (**NIPS**) 1995 workshop *Learning to Learn: Knowledge Consolidation and Transfer in Inductive Systems* is believed to have provided the initial motivations for research in this field.

> All the workshops from NIPS 1995 are listed here:
> `http://www.cs.cmu.edu/afs/cs/project/cnbc/nips/NIPS95/Workshops.`
> `html`.

Since then, terms such as *Learning to Learn*, *Knowledge Consolidation*, and *Inductive Transfer* have been used interchangeably with transfer learning. Invariably, different researchers and academic texts provide definitions from different contexts. In their book, *Deep Learning*, Goodfellow et al. refer to transfer learning in the context of generalization. Their definition is as follows:

Situation where what has been learned in one setting is exploited to improve generalization in another setting.

Let's understand the preceding definition with the help of an example. Let's assume our task is to identify objects in images within a restricted domain of a restaurant. Let's mark this task in its defined scope as T_1. Given the dataset for this task, we train a model and tune it to perform well (generalize) on unseen data points from the same domain (restaurant). Traditional supervised ML algorithms break down when we do not have sufficient training examples for the required tasks in given domains. Suppose we now must detect objects from images in a park or a café (say, task T_2). Ideally, we should be able to apply the model trained for T_1, but in reality we face performance degradation and models that do not generalize well. This happens for a variety of reasons, which we can liberally and collectively term as the model's bias toward training data and domain. Transfer learning thus enables us to utilize knowledge from previously learned tasks and apply them to newer, related ones. If we have significantly more data for task T_1, we may utilize its learnings and generalize them for task T_2 (which has significantly less data). In the case of image classification, certain low-level features, such as edges, shapes, and lighting, can be shared across tasks and thus enable knowledge transfer among tasks.

The following diagram shows how transfer learning enables reusing existing knowledge for new related tasks:
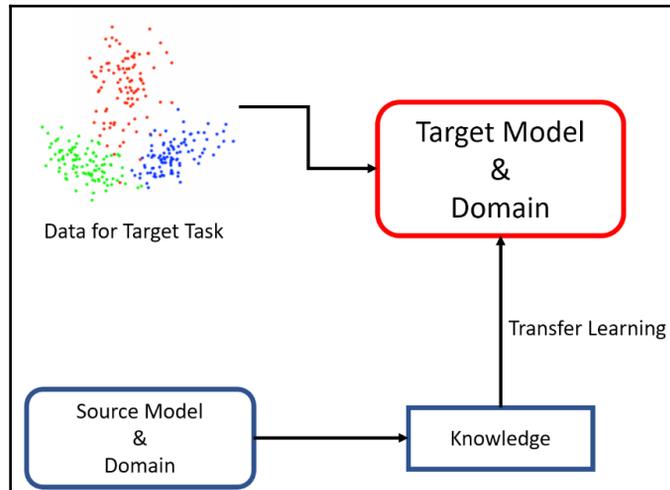
As shown in the preceding diagram, knowledge from an existing task acts as an additional input when learning a target task.

Advantages of transfer learning

We utilize knowledge from source models to improve learning in the target task. Apart from providing capabilities to reuse already-built models, transfer learning may assist learning the target task in the following ways:

- **Improved baseline performance**: When we augment the knowledge of an isolated learner (also known as an **ignorant learner**) with knowledge from a source model, the baseline performance might improve due to this knowledge transfer.
- **Model-development time**: Utilizing knowledge from a source model might also help in fully learning the target task, as compared to a target model that learns from scratch. This, in turn, results in improvements in the overall time taken to develop/learn a model.
- **Improved final performance**: Higher final performance might be attained by leveraging transfer learning.

Readers should note that one or more of these gains are possible, which we will discuss at length in the coming chapters. This is depicted in the following diagram, showcasing better baseline performance (**higher start**), efficiency gains (**higher slope**), and better final performance (**higher asymptote**):
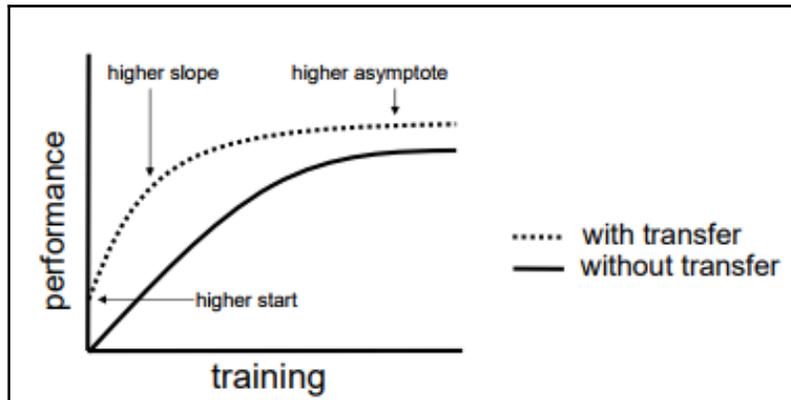
Possible benefits of using transfer learning (Source: Transfer Learning, Lisa Torrey and Jude Shavlik)

Transfer learning has been applied and studied in the context of inductive learners, such as neural networks, and Bayesian networks. Reinforcement learning is another area where possibilities with transfer learning are being explored. Thus, the concept of transfer learning is not limited to deep learning.

In this chapter and the subsequent ones, we will be limiting our scope of utilizing transfer learning to the context of deep learning.

Transfer learning strategies

Let's start by first looking at a formal definition for transfer learning and then utilize it to understand different strategies. In their paper, *A Survey on Transfer Learning* (`https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf`), Pan and Yang use domain, task, and marginal probabilities to present a framework for understanding transfer learning. The framework is defined as follows:

A domain, *D*, is defined as a two-element tuple consisting of feature space, χ, and marginal probability, *P(X)*, where *X* is a sample data point.

Here, $X = \{x_1, x_2....x_n\}$ with x_i as a specific vector and $X \in \mathcal{X}$. Thus:

$$D = \{\chi, P(X)\}$$

A task, T, on the other hand, can be defined as a two-element tuple of the label space, γ, and objective function, f. The objective function can also be denoted as $P(\gamma \mid X)$ from a probabilistic view point. Thus:

$$T = \{\gamma, P(\gamma|X)\}$$

Using this framework, we can define transfer learning as a process aimed at improving the target objective function, f_T (or target task, T_T), in the target domain, D_T, using knowledge from the T_S source task in the D_S domain. This leads to the following four scenarios:

- **Feature space**: The feature spaces of source and target domains are different from each other, such as $\chi_s \neq \chi_t$. For instance, if our tasks are related to document classification, this scenario refers to source and target tasks in different languages.
- **Marginal probability**: The marginal probabilities or source and target domains are different, such as $P(X_s) \neq P(X_t)$. This scenario is also known as **domain adaptation**.
- **Label space**: The label spaces of the source and target domains are different in this scenario, such as $\gamma_s \neq \gamma_t$. This also usually implies the presence of scenario four—different conditional probabilities.
- **Conditional probabilities**: In this case, $P(\Upsilon_s|X_s) \neq P(\Upsilon_t|X_t)$, such that the conditional probabilities are different in the source and target domains.

Transfer learning, as we have seen so far, is having the ability to utilize existing knowledge from the source learner in the target task. During the process of transfer learning, the following three important questions must be answered:

- **What to transfer**: This is the first and the most important step in the whole process. We try to seek answers about which part of the knowledge can be transferred from the source to the target in order to improve the performance of the target task. When trying to answer this question, we try to identify which portion of knowledge is source-specific and what is common between the source and the target.

- **When to transfer**: There can be scenarios where transferring knowledge for the sake of it may make matters worse than improving anything (also known as *negative transfer*). We should aim at utilizing transfer learning to improve target task performance/results and not degrade them. We need to be careful about when to transfer and when not to.
- **How to transfer**: Once the what and when have been answered, we can proceed toward identifying ways of actually transferring the knowledge across domains/tasks. This involves changes to existing algorithms and different techniques, which we will cover in later sections of this chapter. Also, specific use cases are lined up in the next section for a better understanding of how to transfer.

> The paper, *A Survey on Transfer Learning*, by Pan and Yang is available here: `https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf`.

Grouping techniques help us understand overall characteristics and provide a better framework for utilizing them. Transfer learning methods can be categorized based on the type of traditional ML algorithms involved, such as:

- **Inductive transfer**: In this scenario, the source and target domains are the same, yet the source and target tasks are different from each other. The algorithms try to utilize the inductive biases of the source domain to help improve the target task. Depending upon whether the source domain contains labeled data or not, this can be further divided into two subcategories, similar to *multitask learning* and *self-taught learning*, respectively.
- **Unsupervised transfer**: This setting is similar to inductive transfer itself, with a focus on unsupervised tasks in the target domain. The source and target domains are similar, but the tasks are different. In this scenario, labeled data is unavailable in either of the domains.
- **Transductive transfer**: In this scenario, there are similarities between the source and target tasks but the corresponding domains are different. In this setting, the source domain has a lot of labeled data while the target domain has none. This can be further classified into subcategories, referring to settings where either the feature spaces are different or the marginal probabilities.

The three transfer categories discussed in the previous section outline different settings where transfer learning can be applied and studied in detail. To answer the question of what to transfer across these categories, some of the following approaches can be applied:

- **Instance transfer**: Reusing knowledge from the source domain to the target task is usually an ideal scenario. In most cases, the source domain data cannot be reused directly. Rather, there are certain instances from the source domain that can be reused along with target data to improve results. In case of inductive transfer, modifications such as *AdaBoost* by Dai and their co-authors help utilize training instances from the source domain for improvements in the target task.
- **Feature-representation transfer**: This approach aims to minimize domain divergence and reduce error rates by identifying good feature representations that can be utilized from the source to target domains. Depending upon the availability of labeled data, supervised or unsupervised methods may be applied for feature-representation-based transfers.
- **Parameter transfer**: This approach works on the assumption that the models for related tasks share some parameters or prior distribution of hyperparameters. Unlike multitask learning, where both the source and target tasks are learned simultaneously, for transfer learning, we may apply additional weightage to the loss of the target domain to improve overall performance.
- **Relational-knowledge transfer**: Unlike the preceding three approaches, the relational-knowledge transfer attempts to handle non-IID data, such as data that is not independent and identically distributed. In other words, data where each data point has a relationship with other data points; for instance, social network data utilizes relational-knowledge-transfer techniques.

In this section, we studied different strategies for performing transfer learning under different contexts and settings in a very generic manner. Let's now utilize this understanding and learn how transfer learning is applied in the context of deep learning.

Transfer learning and deep learning

Deep learning models are representative of what is also known as **inductive learning**. The objective for inductive-learning algorithms is to infer a mapping from a set of training examples. For instance, in cases of classification, the model learns mapping between input features and class labels. In order for such a learner to generalize well on unseen data, its algorithm works with a set of assumptions related to the distribution of the training data. These sets of assumptions are known as **inductive bias**.

The inductive bias or assumptions can be characterized by multiple factors, such as the hypothesis space it restricts to and the search process through the hypothesis space. Thus, these biases impact how and what is learned by the model on the given task and domain.

Inductive transfer techniques utilize the inductive biases of the source task to assist the target task. This can be done in different ways, such as by adjusting the inductive bias of the target task by limiting the model space, narrowing down the hypothesis space, or making adjustments to the search process itself with the help of knowledge from the source task. This process is depicted visually in the following diagram:
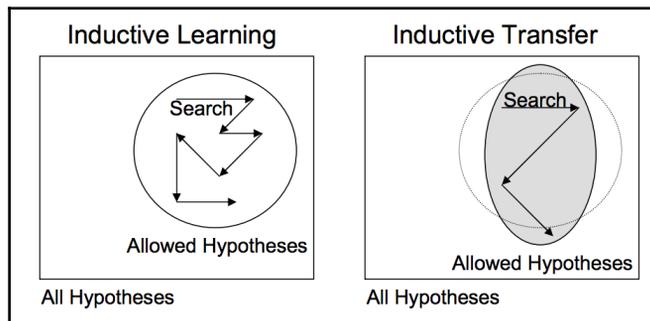
Inductive transfer (Source: Transfer learning, Lisa Torrey and Jude Shavlik)

Apart from inductive transfer, inductive-learning algorithms also utilize *Bayesian* and *hierarchical transfer* techniques to assist with improvements in the learning and performance of the target task.

Transfer learning methodologies

Deep learning has made considerable progress in recent years and the results are amazing. But the training time and the amount of data required for such deep learning systems is orders of magnitudes more than traditional ML systems.

There are various deep learning networks with state-of-the-art performance (sometimes as good or even better than human performance) that have been developed and tested across domains such as computer vision and **natural language processing** (**NLP**). In most cases, teams/people share the details of these networks for others to use (some of the popular ones have been shared in `Chapter 3`, *Understanding Deep Learning Architectures*). These pretrained networks/models form the basis of transfer learning in the context of deep learning.

Feature-extraction

As discussed in `Chapter 3`, *Understanding Deep Learning Architectures* deep learning systems are layered architectures that learn different features at different layers. These layers are then finally connected to a last layer (usually a fully connected layer, in the case of classification) to get the final output. This layered architecture allows us to utilize a pretrained network (such as Inception V3 or VGG) without its final layer as a fixed feature-extractor for other tasks. The following diagram represents deep transfer based on feature-extraction:
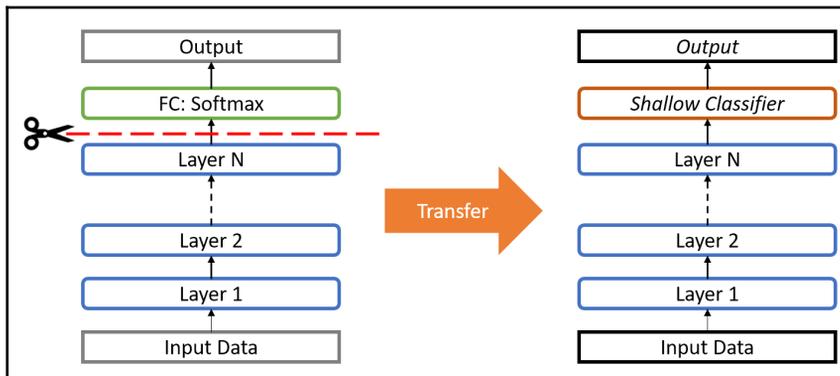
For instance, if we utilize AlexNet without its final classification layer, it will help us transform images from a new domain task into a 4,096-dimensional vector based on its hidden states, thus enabling us to extract features from a new domain task, utilizing the knowledge from a source-domain task. This is one of the most widely utilized methods of performing transfer learning using deep neural networks.

Fine-tuning

This is a more involved technique where we do not just replace the final layer (for classification/regression), but we also selectively retrain some of the previous layers. Deep neural networks are highly configurable architectures with various hyperparameters. As discussed earlier, the initial layers have been seen to capture generic features, while the later ones focus more on the specific task at hand. Using this insight, we may freeze (fix weights) certain layers while retraining, or fine-tune the rest of them to suit our needs. In this case, we utilize the knowledge in terms of the overall architecture of the network and use its states as the starting point for our retraining step. This, in turn, helps us achieve better performance with less training time.

Pretrained models

One of the fundamental requirements for transfer learning is the presence of models that perform well on source tasks. Luckily, the deep learning world believes in sharing. Many of the state-of-the art deep learning architectures have been openly shared by their respective teams. These span across different domains, such as computer vision and NLP. We looked at some of the most well-known and well-documented architectures in `Chapter 3`, *Understanding Deep Learning Architectures*. The teams behind those networks have not just shared the results, but also their pretrained models. Pretrained models are usually shared in the form of the millions of parameters/weights the model achieved while being trained to a stable state. Pretrained models are available for everyone to use through different means. The famous deep learning Python library, `keras`, provides an interface to download various available pretrained networks, such as **XCeption**, **VGG16**, and **InceptionV3**. Along the same lines, pretrained models are also available through TensorFlow and other deep learning libraries. An even more extensive collection of pretrained models developed over the years is available through Berkley's Model Zoo
(`http://caffe.berkeleyvision.org/model_zoo.html`).

Applications

Deep learning is one class of algorithms that has been utilized to reap the benefits of transfer learning very successfully. The following are a few examples:

- **Transfer learning with text data**: Textual data presents all sorts of challenges when it comes to ML and deep learning. These are usually transformed or vectorized using different techniques. Embeddings, such as Word2vec and fastText, have been prepared using different training datasets. These are utilized in different tasks, such as sentiment analysis and document classification, by transferring the knowledge from the source tasks.

- **Transfer learning with computer vision**: Deep learning has been very successfully utilized for various computer vision tasks, such as object identification, using different CNN architectures. In their paper, *How transferable are features in deep neural networks*, Yosinski and their co-authors (`https://arxiv.org/abs/1411.1792`) present their findings on how the lower layers act as conventional computer-vision feature extractors, such as edge detectors, while the final layers work toward task-specific features. Thus, these findings have helped in utilizing existing state-of-the-art models, such as **VGG**, **AlexNet**, and **Inceptions**, for target tasks, such as style transfer and face detection, that were different from what these models were trained for.

- **Transfer learning with speech/audio**: Similar to text and computer-vision domains, deep learning has been successfully used for tasks based on audio data. For instance, **Automatic Speech Recognition** (**ASR**) models developed for English have been successfully used to improve speech recognition performance for other languages, such as German. Also, automated-speaker identification is another example where transfer learning has greatly helped.

Deep transfer learning types

The literature on transfer learning has gone through a lot of iterations, and as mentioned at the start of this chapter, the terms associated with it have been used loosely and often interchangeably. Hence, it is sometimes confusing to differentiate between transfer learning, domain adaptation, and multitask learning. Rest assured, these are all related and try to solve similar problems. For the sake of consistency throughout this book, we will adopt the notion of transfer learning as being a general concept, where we will try to solve a target task using source task-domain knowledge.

Domain adaptation

Domain adaption is usually referred to in scenarios where the marginal probabilities between the source and target domains are different, such as $P\,(X_s) \neq P\,(X_t)$. There is an inherent shift or drift in the data distribution of the source and target domains that requires tweaks to transfer the learning. For instance, a corpus of movie reviews labeled as positive or negative would be different from a corpus of product-review sentiments. A classifier trained on movie-review sentiment would see a different distribution if utilized to classify product reviews. Thus, domain adaptation techniques are utilized in transfer learning in these scenarios.

Domain confusion

We learned different transfer learning strategies and even discussed the three questions of *what*, *when*, and *how* to transfer knowledge from the source to the target. In particular, we discussed how feature-representation transfer can be useful. It is worth reiterating that different layers in a deep learning network capture different sets of features. We can utilize this fact to learn domain-invariant features and improve their transferability across domains. Instead of allowing the model to learn any representation, we nudge the representations of both domains to be as similar as possible.

This can be achieved by applying certain preprocessing steps directly to the representations themselves. Some of these have been discussed by Baochen Sun, Jiashi Feng, and Kate Saenko in their paper *Return of Frustratingly Easy Domain Adaptation* (`https://arxiv.org/abs/1511.05547`). This nudge toward the similarity of representation has also been presented by Ganin et. al. in their paper, *Domain-Adversarial Training of Neural Networks* (`https://arxiv.org/abs/1505.07818`). The basic idea behind this technique is to add another objective to the source model to encourage similarity by confusing the domain itself, hence *domain confusion*.

Multitask learning

Multitask learning is a slightly different flavor of the transfer learning world. In the case of multitask learning, several tasks are learned simultaneously without distinction between the source and targets. In this case, the learner receives information about multiple tasks at once, as compared to transfer learning, where the learner initially has no idea about the target task.

This is depicted in the following diagram:
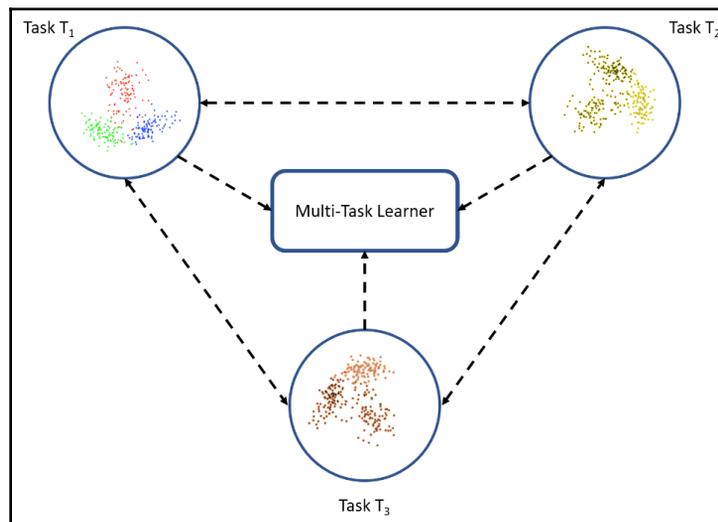
Multitask learning: Learner receives information from all tasks simultaneously

One-shot learning

Deep learning systems are data hungry by nature, such that they need many training examples to learn the weights. This is one of the limiting aspects of deep neural networks, though such is not the case with human learning. For instance, once a child is shown what an apple looks like, they can easily identify a different variety of apple (with one or a few training examples); this is not the case with ML and deep learning algorithms. One-shot learning is a variant of transfer learning where we try to infer the required output based on just one or a few training examples. This is essentially helpful in real-world scenarios where it is not possible to have labeled data for every possible class (if it is a classification task) and in scenarios where new classes can be added often.

The landmark paper by Fei-Fei and their co-authors, *One Shot Learning of Object Categories* (`https://ieeexplore.ieee.org/document/1597116/`), is supposedly what coined the term one-shot learning and the research in this subfield. This paper presented a variation on a Bayesian framework for representation learning for object categorization. This approach has since been improved upon, and applied using deep learning systems.

Zero-shot learning

Zero-shot learning is another extreme variant of transfer learning, which relies on no labeled examples to learn a task. This might sound unbelievable, especially when learning using examples is what most supervised learning algorithms are about. Zero-data learning, or zero-short learning, methods make clever adjustments during the training stage itself to exploit additional information to understand unseen data. In their book on *Deep Learning*, Goodfellow and their co-authors present *zero-shot learning* as a scenario where three variables are learned, such as the traditional input variable, *x,* the traditional output variable, *y,* and the additional random variable that describes the task, *T*. The model is thus trained to learn the conditional probability distribution of $P(y \mid x, T)$. Zero-shot learning comes in handy in scenarios such as machine translation, where we may not even have labels in the target language.

Challenges of transfer learning

Transfer learning has immense potential and is a commonly required enhancement for existing learning algorithms. Yet, there are certain pertinent issues related to transfer learning that need more research and exploration. Apart from the difficulty of answering the questions of what, when, and how to transfer, negative transfer and transfer bounds present major challenges.

Negative transfer

The cases we have discussed so far talk about improvements in target tasks based on knowledge transfer from the source task. There are cases when transfer learning can lead to a drop in performance. Negative transfer refers to scenarios where the transfer of knowledge from the source to the target does not lead to any improvements, but rather causes a drop in the overall performance of the target task. There can be various reasons for negative transfer, such as cases when the source task is not sufficiently related to the target task or if the transfer method could not leverage the relationship between the source and target tasks very well. Avoiding negative transfer is very important and requires careful investigation. In their work, Rosenstien and their co-authors present empirically how brute-force transfer degrades performance in target tasks when the source and target are too dissimilar. Bayesian approaches by Bakker and their co-authors, along with other techniques exploring clustering-based solutions to identify relatedness, are being researched to avoid negative transfers.

Transfer bounds

Quantifying the transfer in transfer learning is also very important affecting about the quality of the transfer and its viability. To gauge the amount for the transfer, Hassan Mahmud and their co-authors used Kolmogorov complexity to prove certain theoretical bounds to analyze transfer learning and measure relatedness between tasks. Eaton and their co-authors presented a novel graph-based approach to measure knowledge transfer. Detailed discussions of these techniques are outside the scope of this book. Readers are encouraged to explore more on these topics using the publications outlined in this section.

Summary

Having set the context and basics of ML and deep learning in Chapter 1 to Chapter 3 of the book, this chapter began the second phase of building the foundations of transfer learning. Before diving into actual use cases, it is imperative that we formalize our understanding of transfer learning and learn about different techniques and research, and the challenges associated with it. Throughout this chapter, we have presented the fundamentals behind the concept of transfer learning, how it has evolved over the years, and why it was required in the first place.

We began by understanding transfer learning in the broader context of learning algorithms and their associated advantages. We then discussed various strategies for understanding, applying, and categorizing transfer learning methods. Transfer learning in the context of deep learning was the next topic discussed, to set the tone for the rest of the chapter. We discussed different transfer learning methodologies, such as *feature-extraction* and *fine-tuning,* that are associated with deep transfer learning. We also presented famous pretrained models and popular applications of transfer learning using deep learning systems. Deep learning has proven to be very successful in recent years, and thus a lot of research is being done into using transfer learning in this space.

We briefly discussed different variants of deep transfer learning, such as *domain adaptation*, *domain confusion, multitask learning, one-shot learning*, and *zero-shot learning*. We concluded the chapter by presenting the challenges, such as negative transfer and transfer bounds, associated with transfer learning. Throughout this chapter, we have outlined various research publications and links associated with transfer learning, and we encourage readers to explore them for more information. This chapter acts as a pointer and an overview of the current transfer learning landscape. Stay tuned for more details in the next chapter where we will get to some hands-on exercises related to Transfer Learning.

5
Unleashing the Power of Transfer Learning

In the previous chapter, we covered the major concepts surrounding transfer learning. The key idea was that leveraging state-of-the-art, pretrained deep learning models in a wide variety of tasks yields superior results compared to building your own deep learning models and architectures from scratch. In this chapter, we will gain a more hands-on perspective of actually building deep learning models using transfer learning and applying them to a real-world problem. We will build various deep learning models with and without transfer learning. We will analyze their architecture and also compare and contrast their performance. We will be covering the following major aspects in this chapter:

- The need for transfer learning
- Building **Convolutional Neural Network** (**CNN**) models from scratch:
    - Building a basic CNN model
    - Improving our CNN model with regularization
    - Improving our CNN model with image augmentation
- Leveraging transfer learning with pretrained CNN models:
    - Using a pretrained model as a feature extractor
    - Improving our pretrained model with image augmentation
    - Improving our pretrained model with fine-tuning
- Model performance evaluation

We would like to thank Francois Chollet for not only creating the amazing deep learning framework, Keras, but also for talking about the real-world problem where transfer learning is effective in his book, *Deep Learning with Python*. We have taken that as an inspiration to portray the true power of transfer learning in this chapter. The code for this chapter will be available in the `Chapter 5` folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

The need for transfer learning

We have already briefly discussed the advantages of transfer learning, in `Chapter 4`, *Transfer Learning Fundamentals*. To recap, we get several benefits, such as improving the baseline performance, speeding up the overall model development and training time, and also getting an overall improved and superior model performance as compared to building a deep learning model from scratch. An important thing to remember here is that transfer learning as a domain existed long before deep learning and can also be applied to areas or problems that do not need deep learning.

Let's consider a real-world problem now, one which we will also be using throughout this chapter to illustrate our different deep learning models and leverage transfer learning on the same. One of the key requirements of deep learning, which you must have heard time and again, is that we need a lot of data and samples to build robust deep learning models. The idea behind this is that the model can then learn features automatically from a large number of samples. But what do we do if we don't have enough training samples and the problem to solve is still a relatively complex problem? For instance, a computer vision problem, such as image categorization, which might be difficult to solve using traditional statistical or **machine learning** (**ML**) techniques. Do we give up on deep learning?

Considering an **image categorization** problem, since we are dealing with images that are essentially high-dimensional tensors, having more data enables deep learning models to learn better underlying feature representations of the images. However, even if we have image samples ranging from a few hundred to thousands per category, a basic CNN model will still perform decently with the right architecture and regularization. The key point to remember here is the fact that CNNs learn patterns and features that are invariant to scaling, translation, and rotation, and hence we do not need custom feature engineering techniques here. However, we might still run into problems like model overfitting, which we will try to address later on in this chapter.

With regard to transfer learning, there are some excellent pretrained deep learning models that have been trained on the famous ImageNet dataset (`http://image-net.org/about-overview`). We have covered some of these models in detail in `Chapter 3`, *Understanding Deep Learning Architectures*, and we will be leveraging the famous `VGG-16` model in this chapter. The idea is to use a pretrained model that is usually an expert in image categorization to solve our problem with the constraint of fewer data samples.

Formulating our real-world problem

As we mentioned earlier, we will be working on an image categorization problem with the constraint of having a small number of training samples per category. The dataset for our problem is available on Kaggle and is one of the most popular computer vision based datasets out there. The dataset we will be using comes from the **Dogs vs. Cats** challenge (`https://www.kaggle.com/c/dogs-vs-cats/data`), whereas our primary objective is to build a model that can successfully recognize and categorize images into either a cat or a dog. In terms of ML, this is a binary classification problem based on images.

To start, download the `train.zip` file from the dataset page and store it in your local system. Once downloaded, unzip it into a folder. This folder will contain 25,000 images of dogs and cats; that is, 12,500 images per category.

Building our dataset

While we can use all 25,000 images and build some nice models on them, if you remember, our problem objective includes the added constraint of having a small number of images per category. Let's build our own dataset for this purpose. You can refer to the `Datasets Builder.ipynb` Jupyter Notebook in case you want to run the examples yourself.

To start with, we load up the following dependencies, including a utility module called `utils`, which is available in the `utils.py` file present in the code files for this chapter. This is mainly used to get a visual progress bar when we copy images into new folders:

```
import glob
import numpy as np
import os
import shutil
from utils import log_progress

np.random.seed(42)
```

Let's now load up all the images in our original training data folder as follows:

```
files = glob.glob('train/*')

cat_files = [fn for fn in files if 'cat' in fn]
dog_files = [fn for fn in files if 'dog' in fn]
len(cat_files), len(dog_files)

Out [3]: (12500, 12500)
```

We can verify with the preceding output that we have 12,500 images for each category. Let's now build our smaller dataset so that we have 3,000 images for training, 1,000 images for validation, and 1,000 images for our test dataset (with equal representation for the two animal categories):

```
cat_train = np.random.choice(cat_files, size=1500, replace=False)
dog_train = np.random.choice(dog_files, size=1500, replace=False)
cat_files = list(set(cat_files) - set(cat_train))
dog_files = list(set(dog_files) - set(dog_train))

cat_val = np.random.choice(cat_files, size=500, replace=False)
dog_val = np.random.choice(dog_files, size=500, replace=False)
cat_files = list(set(cat_files) - set(cat_val))
dog_files = list(set(dog_files) - set(dog_val))

cat_test = np.random.choice(cat_files, size=500, replace=False)
dog_test = np.random.choice(dog_files, size=500, replace=False)

print('Cat datasets:', cat_train.shape, cat_val.shape, cat_test.shape)
print('Dog datasets:', dog_train.shape, dog_val.shape, dog_test.shape)


Cat datasets: (1500,) (500,) (500,)
Dog datasets: (1500,) (500,) (500,)
```

Now that our datasets have been created, let's write them out to our disk in separate folders, so that we can come back to them anytime in the future without worrying if they are present in our main memory:

```
train_dir = 'training_data'
val_dir = 'validation_data'
test_dir = 'test_data'

train_files = np.concatenate([cat_train, dog_train])
validate_files = np.concatenate([cat_val, dog_val])
test_files = np.concatenate([cat_test, dog_test])

os.mkdir(train_dir) if not os.path.isdir(train_dir) else None
os.mkdir(val_dir) if not os.path.isdir(val_dir) else None
os.mkdir(test_dir) if not os.path.isdir(test_dir) else None

for fn in log_progress(train_files, name='Training Images'):
    shutil.copy(fn, train_dir)
for fn in log_progress(validate_files, name='Validation Images'):
    shutil.copy(fn, val_dir)
for fn in log_progress(test_files, name='Test Images'):
    shutil.copy(fn, test_dir)
```

The progress bars depicted in the following screenshot become green once all the images have been copied to their respective directory:
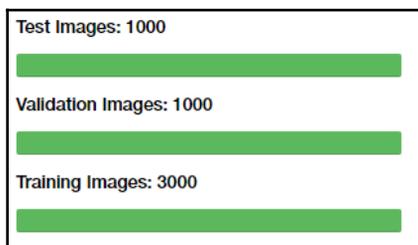
Formulating our approach

Since this is an image categorization problem, we will be leveraging CNN models or convNets to try and tackle this problem. We briefly talked about our approach at the beginning of this chapter. We will start by building simple CNN models from scratch, then try to improve using techniques such as regularization and image augmentation. Then, we will try and leverage pretrained models to unleash the true power of transfer learning!

Building CNN models from scratch

Let's start building our image categorization classifier. Our approach will be to build models on our training dataset and validate it on our validation dataset. In the end, we will test the performance of all our models on the test dataset. Before we jump into modeling, let's load and prepare our datasets. To start with, we load up some basic dependencies:

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

%matplotlib inline
```

Let's now load our datasets, using the following code snippet:

```
IMG_DIM = (150, 150)

train_files = glob.glob('training_data/*')
train_imgs = [img_to_array(load_img(img, target_size=IMG_DIM)) for img
              in train_files]
train_imgs = np.array(train_imgs)
train_labels = [fn.split('/')[1].split('.')[0].strip() for fn in
                train_files]

validation_files = glob.glob('validation_data/*')
validation_imgs = [img_to_array(load_img(img, target_size=IMG_DIM)) for
                   img in validation_files]
validation_imgs = np.array(validation_imgs)
validation_labels = [fn.split('/')[1].split('.')[0].strip() for fn in
                     validation_files]

print('Train dataset shape:', train_imgs.shape,
      'tValidation dataset shape:', validation_imgs.shape)


Train dataset shape: (3000, 150, 150, 3)
Validation dataset shape: (1000, 150, 150, 3)
```

We can clearly see that we have 3000 training images and 1000 validation images. Each image is of size 150 x 150 and has three channels for red, green, and blue (RGB), hence giving each image the (150, 150, 3) dimensions. We will now scale each image with pixel values between (0, 255) to values between (0, 1) because deep learning models work really well with small input values:

```
train_imgs_scaled = train_imgs.astype('float32')
validation_imgs_scaled = validation_imgs.astype('float32')
train_imgs_scaled /= 255
validation_imgs_scaled /= 255

# visualize a sample image
print(train_imgs[0].shape)
array_to_img(train_imgs[0])

(150, 150, 3)
```

The preceding code generates the following output:

The preceding output shows one of the sample images from our training dataset. Let's now set up some basic configuration parameters and also encode our text class labels into numeric values (otherwise, Keras will throw an error):

```
batch_size = 30
num_classes = 2
epochs = 30
input_shape = (150, 150, 3)

# encode text category labels
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
le.fit(train_labels)
train_labels_enc = le.transform(train_labels)
validation_labels_enc = le.transform(validation_labels)

print(train_labels[1495:1505], train_labels_enc[1495:1505])


['cat', 'cat', 'cat', 'cat', 'cat', 'dog', 'dog', 'dog', 'dog', 'dog']
[0 0 0 0 0 1 1 1 1 1]
```

We can see that our encoding scheme assigns the number 0 to the cat labels and 1 to the dog labels. We are now ready to build our first CNN-based deep learning model.

Basic CNN model

We will start by building a basic CNN model with three convolutional layers, coupled with max pooling for auto-extraction of features from our images and also downsampling the output convolution feature maps. To refresh your memory on how convolution and pooling layers work, check out the CNN section in `Chapter 3`, *Understanding Deep Learning Architectures*.

After extracting these feature maps, we will use one dense layer along with an output layer with a sigmoid function for classification. Since we are doing binary classification, a `binary_crossentropy` loss function will suffice. We will use the popular RMSprop optimizer, which helps us in optimizing the weights of the units in our network using backpropagation so that we minimize the loss in our network and, in turn, we end up with a decent classifier. Refer to the *Stochastic Gradient Descent* and *Improvements of SGD* sections in `Chapter 2`, *Deep Learning Essentials*, to get an in-depth idea around how optimizers work. In a nutshell, an optimizer (like RMSprop) specifies rules as to how the gradient of the loss will be used to update parameters in each batch of data passed to our network.

Let's leverage Keras and build our CNN model architecture now:

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.models import Sequential
from keras import optimizers

model = Sequential()

# convolution and pooling layers
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(1, activation='sigmoid'))


model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(),
              metrics=['accuracy'])

model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 16)      448
_____
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 16)        0
_____
conv2d_2 (Conv2D)            (None, 72, 72, 64)        9280
_____
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
_____
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_____
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
_____
flatten_1 (Flatten)          (None, 36992)             0
_____
dense_1 (Dense)              (None, 512)               18940416
_____
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 19,024,513
Trainable params: 19,024,513
Non-trainable params: 0
```

The preceding output shows us our basic CNN model summary. Just like we mentioned before, we are using three convolutional layers for feature extraction. The flatten layer is used to flatten out 128 of the 17 x 17 feature maps that we get as output from the third convolution layer. This is fed to our dense layers to get the final prediction of whether the image should be a dog (1) or a cat (0). All of this is part of the model training process, so let's train our model using the following snippet which leverages the `fit(...)` function. The following terminology is very important with regard to training our model:

- The `batch_size` indicates the total number of images passed to the model per iteration
- The weights of the units in layers are updated after each iteration
- The total number of iterations is always equal to the total number of training samples divided by the `batch_size`
- An epoch is when the complete dataset has passed through the network once, that is, all the iterations are completed based on data batches

We use a `batch_size` of `30` and our training data has a total of 3,000 samples, which indicates that there will be a total of 100 iterations per epoch. We train the model for a total of 30 epochs and validate it consequently on our validation set of 1,000 images:

```
history = model.fit(x=train_imgs_scaled, y=train_labels_enc,
validation_data=(validation_imgs_scaled,
                 validation_labels_enc),
                 batch_size=batch_size,
                 epochs=epochs,
                 verbose=1)

Train on 3000 samples, validate on 1000 samples
Epoch 1/30
3000/3000 - 10s - loss: 0.7583 - acc: 0.5627 - val_loss: 0.7182 - val_acc:
0.5520
Epoch 2/30
3000/3000 - 8s - loss: 0.6343 - acc: 0.6533 - val_loss: 0.5891 - val_acc:
0.7190
...
...
Epoch 29/30
3000/3000 - 8s - loss: 0.0314 - acc: 0.9950 - val_loss: 2.7014 - val_acc:
0.7140
Epoch 30/30
3000/3000 - 8s - loss: 0.0147 - acc: 0.9967 - val_loss: 2.4963 - val_acc:
0.7220
```

Looks like our model is kind of overfitting, based on the training and validation accuracy values. We can plot our model accuracy and errors using the following snippet to get a better perspective:

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
t = f.suptitle('Basic CNN Performance', fontsize=12)
f.subplots_adjust(top=0.85, wspace=0.3)

epoch_list = list(range(1,31))
ax1.plot(epoch_list, history.history['acc'], label='Train Accuracy')
ax1.plot(epoch_list, history.history['val_acc'], label='Validation
Accuracy')
ax1.set_xticks(np.arange(0, 31, 5))
ax1.set_ylabel('Accuracy Value')
ax1.set_xlabel('Epoch')
ax1.set_title('Accuracy')
l1 = ax1.legend(loc="best")
```

```
ax2.plot(epoch_list, history.history['loss'], label='Train Loss')
ax2.plot(epoch_list, history.history['val_loss'], label='Validation Loss')
ax2.set_xticks(np.arange(0, 31, 5))
ax2.set_ylabel('Loss Value')
ax2.set_xlabel('Epoch')
ax2.set_title('Loss')
l2 = ax2.legend(loc="best")
```

The following plots leverage the history object, which contains the accuracy and loss values per epoch:
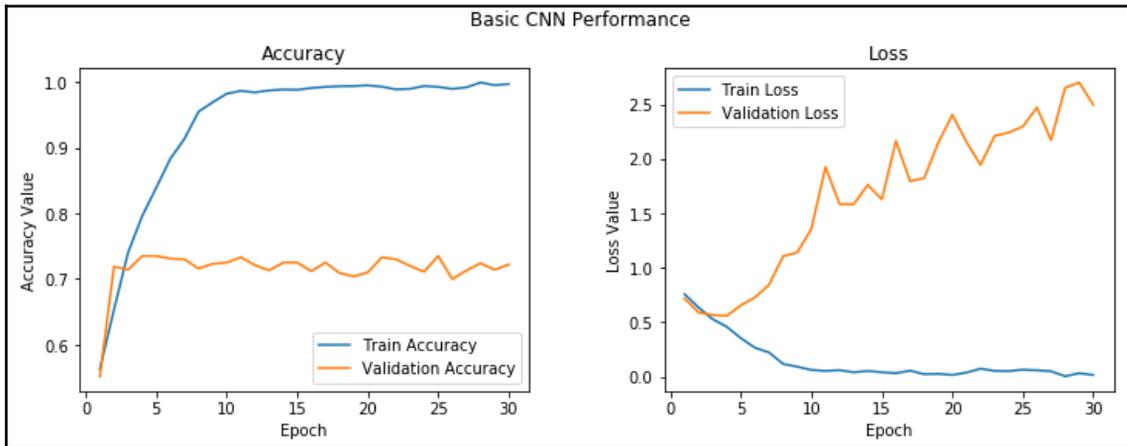
You can clearly see that after 2-3 epochs the model starts overfitting on the training data. The average accuracy we get in our validation set is around **72%**, which is not a bad start! Can we improve upon this model?

CNN model with regularization

Let's improve upon our base CNN model by adding in one more convolution layer, another dense hidden layer. Besides this, we will add dropout of 0.3 after each hidden dense layer to enable regularization. We covered dropout briefly in `Chapter 2`, *Deep Learning Essentials*, so feel free to quickly skim through it in case you need a refresher. Basically, dropout is a powerful method of regularizing in deep neural nets. It can be applied separately to both input layers and the hidden layers.

Dropout randomly masks the outputs of a fraction of units from a layer by setting their output to zero (in our case, it is 30% of the units in our dense layers):

```
model = Sequential()
# convolutional and pooling layers
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(),
              metrics=['accuracy'])
```

Let's now train our new model on the training data and validate its performance on the validation dataset:

```
history = model.fit(x=train_imgs_scaled, y=train_labels_enc,
                    validation_data=(validation_imgs_scaled,
                                     validation_labels_enc),
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1)

Train on 3000 samples, validate on 1000 samples
Epoch 1/30
3000/3000 - 7s - loss: 0.6945 - acc: 0.5487 - val_loss: 0.7341 - val_acc:
0.5210
Epoch 2/30
3000/3000 - 7s - loss: 0.6601 - acc: 0.6047 - val_loss: 0.6308 - val_acc:
0.6480
...
...
Epoch 29/30
3000/3000 - 7s - loss: 0.0927 - acc: 0.9797 - val_loss: 1.1696 - val_acc:
0.7380
```

```
Epoch 30/30
3000/3000 - 7s - loss: 0.0975 - acc: 0.9803 - val_loss: 1.6790 - val_acc:
0.7840
```

Let's also look at the accuracy and loss values across all the epochs during model training:
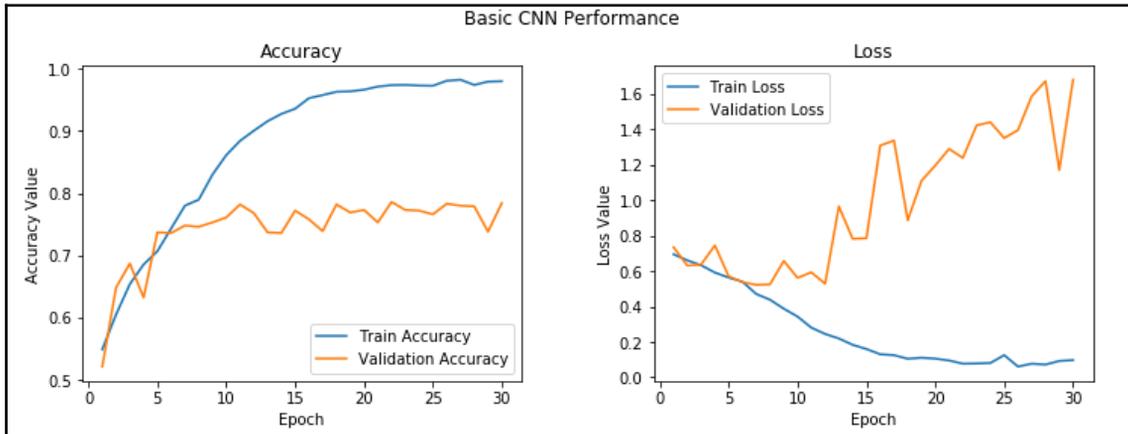
You can clearly see from the preceding outputs that we still end up overfitting the model, though it takes slightly longer and we also get a slightly better validation accuracy of around **78%**, which is decent but not amazing.

The reason for model overfitting is because we have much less training data and the model keeps seeing the same instances over time across each epoch. A way to combat this would be to leverage an image augmentation strategy to augment our existing training data with images that are slight variations of the existing images. We will cover this in detail in the following section. Let's save this model for the time being so we can use it later to evaluate its performance on the test data:

```
model.save('cats_dogs_basic_cnn.h5')
```

CNN model with image augmentation

Let's improve upon our regularized CNN model by adding in more data using a proper image augmentation strategy. Since our previous model was trained on the same small sample of data points each time, it wasn't able to generalize well and ended up overfitting after a few epochs.

The idea behind image augmentation is that we follow a set process of taking in existing images from our training dataset and applying some image transformation operations to them, such as rotation, shearing, translation, zooming, and so on, to produce new, altered versions of existing images. Due to these random transformations, we don't get the same images each time, and we will leverage Python generators to feed in these new images to our model during training.

The Keras framework has an excellent utility called `ImageDataGenerator` that can help us in doing all the preceding operations. Let's initialize two of the data generators for our training and validation datasets:

```
train_datagen = ImageDataGenerator(rescale=1./255, zoom_range=0.3,
                                   rotation_range=50,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')

val_datagen = ImageDataGenerator(rescale=1./255)
```

There are a lot of options available in `ImageDataGenerator` and we have just utilized a few of them. Feel free to check out the documentation at `https://keras.io/preprocessing/image/` to get a more detailed perspective. In our training data generator, we take in the raw images and then perform several transformations on them to generate new images. These include the following:

- Zooming the image randomly by a factor of `0.3` using the `zoom_range` parameter.
- Rotating the image randomly by `50` degrees using the `rotation_range` parameter.
- Translating the image randomly horizontally or vertically by a `0.2` factor of the image's width or height using the `width_shift_range` and the `height_shift_range` parameters.
- Applying shear-based transformations randomly using the `shear_range` parameter.
- Randomly flipping half of the images horizontally using the `horizontal_flip` parameter.
- Leveraging the `fill_mode` parameter to fill in new pixels for images after we apply any of the preceding operations (especially rotation or translation). In this case, we just fill in the new pixels with their nearest surrounding pixel values.

Let's see how some of these generated images might look so that you can understand them better. We will take two sample images from our training dataset to illustrate the same. The first image is an image of a cat:

```
img_id = 2595
cat_generator = train_datagen.flow(train_imgs[img_id:img_id+1],
                                    train_labels[img_id:img_id+1],
                                    batch_size=1)
cat = [next(cat_generator) for i in range(0,5)]
fig, ax = plt.subplots(1,5, figsize=(16, 6))
print('Labels:', [item[1][0] for item in cat])
l = [ax[i].imshow(cat[i][0][0]) for i in range(0,5)]
```

You can clearly see in the following output that we generate a new version of our training image each time (with translations, rotations, and zoom) and also we assign a label of cat to it so that the model can extract relevant features from these images and also remember that these are cats:
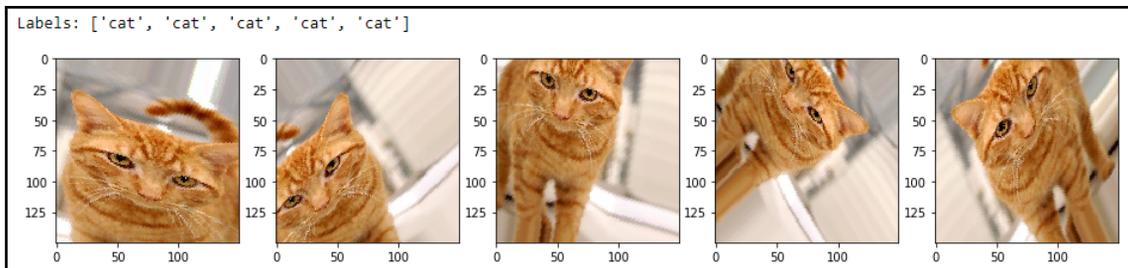
Let's look at an image that is a dog now:

```
img_id = 1991
dog_generator = train_datagen.flow(train_imgs[img_id:img_id+1],
                                    train_labels[img_id:img_id+1],
                                    batch_size=1)
dog = [next(dog_generator) for i in range(0,5)]
fig, ax = plt.subplots(1,5, figsize=(15, 6))
print('Labels:', [item[1][0] for item in dog])
l = [ax[i].imshow(dog[i][0][0]) for i in range(0,5)]
```

This shows us how image augmentation helps in creating new images, and how training a model on them should help in combating overfitting:
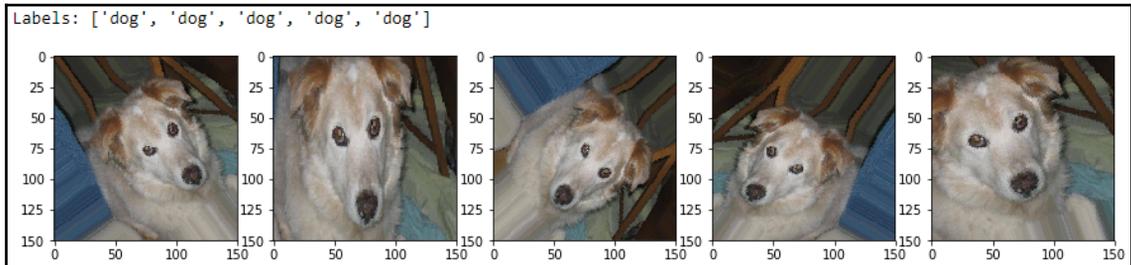
Remember for our validation generator, we just need to send the validation images (original ones) to the model for evaluation; hence, we just scale the image pixels (between 0-1) and do not apply any transformations. We just apply image augmentation transformations only on our training images:

```
train_generator = train_datagen.flow(train_imgs, train_labels_enc,
                                     batch_size=30)
val_generator = val_datagen.flow(validation_imgs,
                                 validation_labels_enc,
                                 batch_size=20)

input_shape = (150, 150, 3)
```

Let's now train a CNN model with regularization using the image augmentation data generators we created. We will use the same model architecture from before:

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.models import Sequential
from keras import optimizers

model = Sequential()
# convolution and pooling layers
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu',
                input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
```

```
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['accuracy'])
```

We reduce the default learning rate by a factor of 10 here for our optimizer to prevent the model from getting stuck in a local minima or overfit, as we will be sending a lot of images with random transformations. To train the model, we need to slightly modify our approach now, since we are using data generators. We will leverage the `fit_generator(...)` function from Keras to train this model. The `train_generator` generates 30 images each time, so we will use the `steps_per_epoch` parameter and set it to 100 to train the model on 3,000 randomly generated images from the training data for each epoch. Our `val_generator` generates 20 images each time so we will set the `validation_steps` parameter to 50 to validate our model accuracy on all the 1,000 validation images (remember we are not augmenting our validation dataset):

```
history = model.fit_generator(train_generator,
                              steps_per_epoch=100, epochs=100,
                              validation_data=val_generator,
                              validation_steps=50, verbose=1)

Epoch 1/100
100/100 - 12s - loss: 0.6924 - acc: 0.5113 - val_loss: 0.6943 - val_acc:
0.5000
Epoch 2/100
100/100 - 11s - loss: 0.6855 - acc: 0.5490 - val_loss: 0.6711 - val_acc:
0.5780
...
...
Epoch 99/100
100/100 - 11s - loss: 0.3735 - acc: 0.8367 - val_loss: 0.4425 - val_acc:
0.8340
Epoch 100/100
100/100 - 11s - loss: 0.3733 - acc: 0.8257 - val_loss: 0.4046 - val_acc:
0.8200
```

We get a validation accuracy jump to around **82%**, which is almost **4-5%** better than our previous model. Also, our training accuracy is very similar to our validation accuracy, indicating our model isn't overfitting anymore. The following depict the model accuracy and loss per epoch:
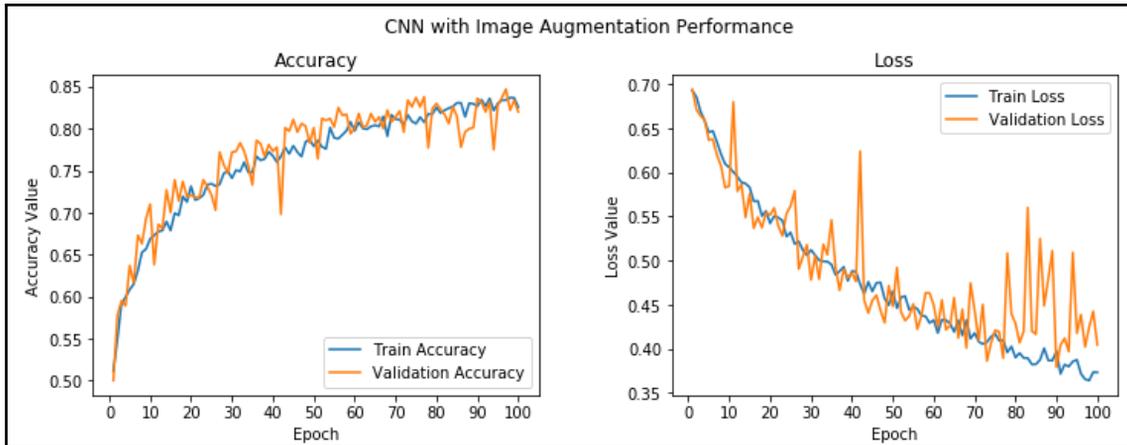
While there are some spikes in the validation accuracy and loss, overall, we see that it is much closer to the training accuracy, with the loss indicating that we obtained a model that generalizes much better as compared to our previous models. Let's save this model now so we can evaluate it later on our test dataset:

```
model.save('cats_dogs_cnn_img_aug.h5')
```

We will now try and leverage the power of transfer learning to see if we can build a better model.

Leveraging transfer learning with pretrained CNN models

So far, we have built our CNN deep learning models from scratch by specifying our own architecture. In this section, we will leverage a pretrained model that is basically an expert in the computer vision domain and renowned for image classification and categorization. We recommend you to check out `Chapter 4`, *Transfer Learning Fundamentals*, for a brief refresher around pretrained models and their applications in this domain.

Pretrained models are used in the following two popular ways when building new models or reusing them:

- Using a pretrained model as a feature extractor
- Fine-tuning the pretrained model

We will cover both of them in detail in this section. The pretrained model we will be using in this chapter is the popular VGG-16 model, created by the Visual Geometry Group at the University of Oxford, which specializes in building very deep convolutional networks for large-scale visual recognition. You can find out more about it at `http://www.robots.ox.ac.uk/~vgg/research/very_deep/`. The **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**) evaluates algorithms for object detection and image classification at large scale and their models have often secured the first place in this competition.

A pretrained model like the VGG-16 is an already trained model on a huge dataset (ImageNet) with a lot of diverse image categories. Considering this fact, the model should have learned a robust hierarchy of features, which are spatial, rotation, and translation invariant, as we have discussed before with regard to features learned by CNN models. Hence, the model, having learned a good representation of features for over a million images belonging to 1,000 different categories, can act as a good feature extractor for new images suitable for computer vision problems. These new images might never exist in the ImageNet dataset or might be of totally different categories, but the model should still be able to extract relevant features from these images, considering the principles of transfer learning that we discussed in `Chapter 4`, *Transfer Learning Fundamentals*.

This gives us an advantage of using pretrained models as effective feature extractors for new images, to solve diverse and complex computer vision tasks, such as solving our cat versus dog classifier with fewer images, or even building a dog breed classifier, a facial expression classifier, and much more! Let's briefly discuss the VGG-16 model architecture before unleashing the power of transfer learning on our problem.

Understanding the VGG-16 model

The VGG-16 model is a 16-layer (convolution and fully connected) network built on the ImageNet database, which is built for the purpose of image recognition and classification. This model was built by Karen Simonyan and Andrew Zisserman and is mentioned in their paper titled *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv (2014), available at `https://arxiv.org/pdf/1409.1556.pdf`.

I recommend all interested readers to go and read up on the excellent literature in this paper. The VGG-16 model was briefly mentioned in `Chapter 3`, *Understanding Deep Learning Architectures*, but we will discuss it in more detail and also leverage the same in our examples. The architecture of the VGG-16 model is depicted in the following diagram:
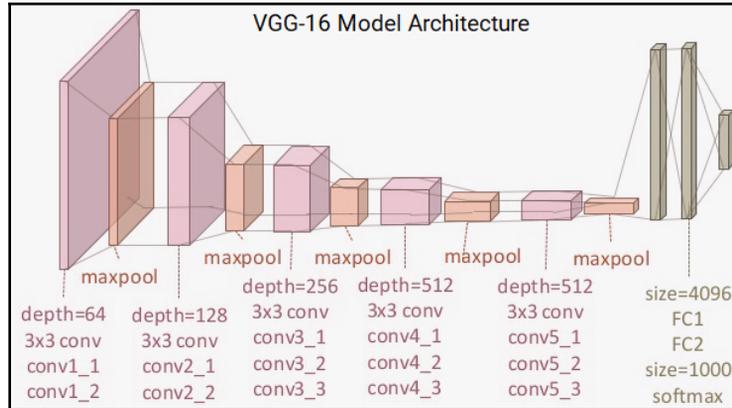
You can clearly see that we have a total of 13 convolution layers using 3 x 3 convolution filters along with max pooling layers for downsampling and a total of two fully connected hidden layers of 4,096 units in each layer followed by a dense layer of 1,000 units, where each unit represents one of the image categories in the ImageNet database.

We do not need the last three layers since we will be using our own fully connected dense layers to predict whether images will be a dog or a cat. We are more concerned with the first five blocks, so that we can leverage the VGG model as an effective feature extractor. For one of the models, we will use it as a simple feature extractor by freezing all the five convolution blocks to make sure their weights don't get updated after each epoch. For the last model, we will apply fine-tuning to the VGG model, where we will unfreeze the last two blocks (**Block 4** and **Block 5**) so that their weights get updated in each epoch (per batch of data) as we train our own model.

We represent the preceding architecture, along with the two variants (basic feature extractor and fine-tuning) that we will be using, in the following block diagram, so you can get a better visual perspective:
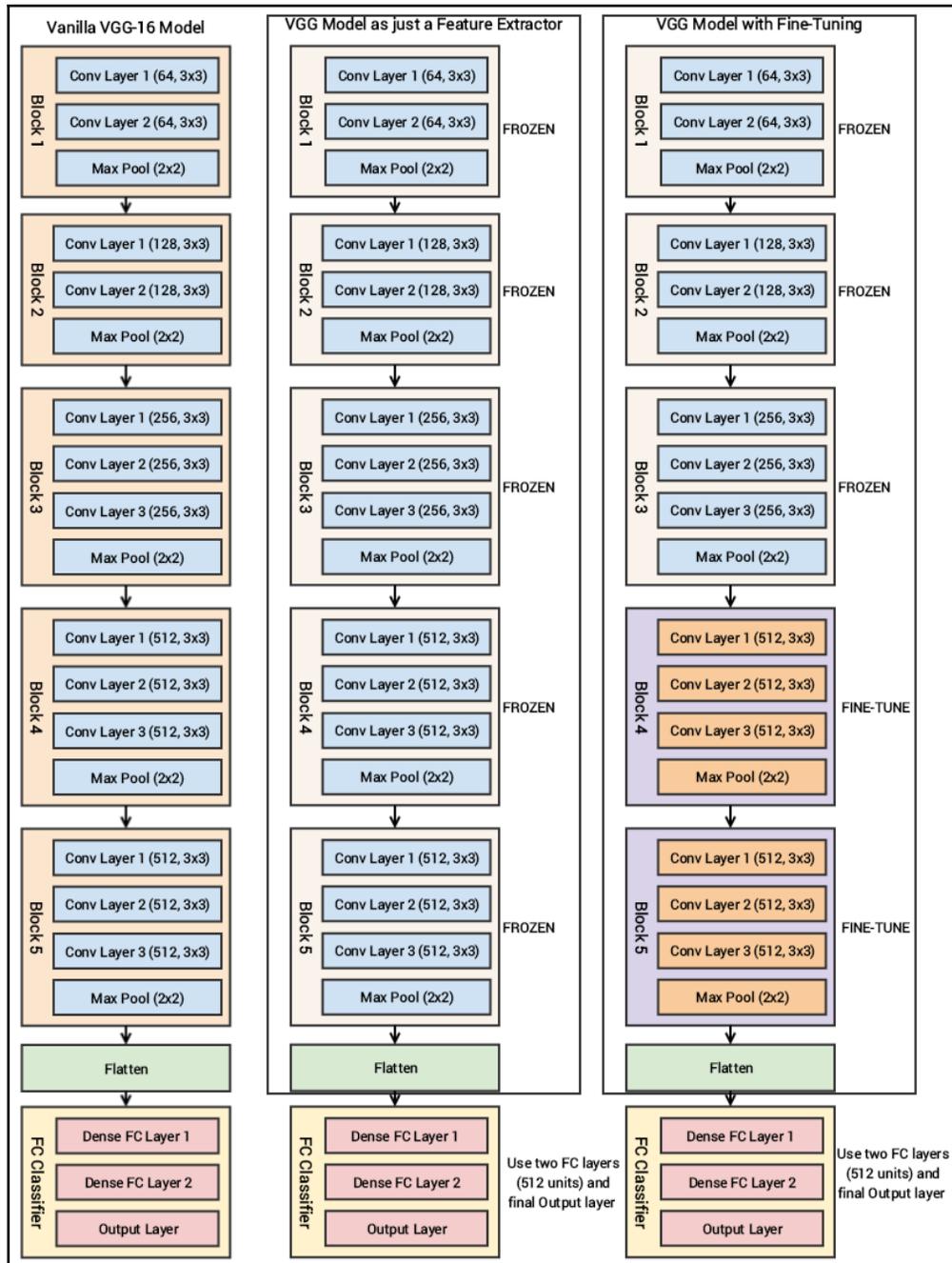
Thus, we are mostly concerned with leveraging the convolution blocks of the VGG-16 model and then flattening the final output (from the feature maps) so that we can feed it into our own dense layers for our classifier. All the code used in this section of the chapter is available in the CNN with the `Transfer Learning.ipynb` Jupyter Notebook.

Pretrained CNN model as a feature extractor

Let's leverage Keras, load up the VGG-16 model, and freeze the convolution blocks so that we can use it as just an image feature extractor:

```python
from keras.applications import vgg16
from keras.models import Model
import keras

vgg = vgg16.VGG16(include_top=False, weights='imagenet',
                                  input_shape=input_shape)

output = vgg.layers[-1].output
output = keras.layers.Flatten()(output)
vgg_model = Model(vgg.input, output)
vgg_model.trainable = False

for layer in vgg_model.layers:
    layer.trainable = False

vgg_model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0

```
block3_conv1 (Conv2D)          (None, 37, 37, 256)          295168
_____
block3_conv2 (Conv2D)          (None, 37, 37, 256)          590080
_____
block3_conv3 (Conv2D)          (None, 37, 37, 256)          590080
_____
block3_pool (MaxPooling2D)     (None, 18, 18, 256)          0
_____
block4_conv1 (Conv2D)          (None, 18, 18, 512)          1180160
_____
block4_conv2 (Conv2D)          (None, 18, 18, 512)          2359808
_____
block4_conv3 (Conv2D)          (None, 18, 18, 512)          2359808
_____
block4_pool (MaxPooling2D)     (None, 9, 9, 512)            0
_____
block5_conv1 (Conv2D)          (None, 9, 9, 512)            2359808
_____
block5_conv2 (Conv2D)          (None, 9, 9, 512)            2359808
_____
block5_conv3 (Conv2D)          (None, 9, 9, 512)            2359808
_____
block5_pool (MaxPooling2D)     (None, 4, 4, 512)            0
_____
flatten_1 (Flatten)            (None, 8192)                 0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_____
```

The model summary shows us each block and the layers present in each block, which match with the architecture diagrams we depicted earlier. You can see that we have removed the final part of the classifier pertaining to the VGG-16 model since we will be building our own classifier and leveraging VGG as a feature extractor.

To verify that the layers of the VGG-16 model are frozen, we can use the following code:

```
import pandas as pd
pd.set_option('max_colwidth', -1)

layers = [(layer, layer.name, layer.trainable) for layer in
          vgg_model.layers]
pd.DataFrame(layers, columns=['Layer Type', 'Layer Name', 'Layer
                             Trainable'])
```

The preceding code generates the following output:

	Layer Type	Layer Name	Layer Trainable
0	<keras.engine.topology.InputLayer object at 0x7f26c86b2518>	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0x7f277c9fc080>	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0x7f26c86b26d8>	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0x7f26c86e6c88>	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0x7f26c867dc18>	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0x7f26c8690f28>	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0x7f26c869e5c0>	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0x7f26c863f828>	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0x7f26c863f128>	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0x7f26c86607b8>	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83d7d68>	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0x7f26c83fd358>	block4_conv1	False
12	<keras.layers.convolutional.Conv2D object at 0x7f26c83fddd8>	block4_conv2	False
13	<keras.layers.convolutional.Conv2D object at 0x7f26c839da20>	block4_conv3	False
14	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83ac1d0>	block4_pool	False
15	<keras.layers.convolutional.Conv2D object at 0x7f26c834e978>	block5_conv1	False
16	<keras.layers.convolutional.Conv2D object at 0x7f271a15eb38>	block5_conv2	False
17	<keras.layers.convolutional.Conv2D object at 0x7f26c8371d68>	block5_conv3	False
18	<keras.layers.pooling.MaxPooling2D object at 0x7f26c8314b00>	block5_pool	False
19	<keras.layers.core.Flatten object at 0x7f26c828bda0>	flatten_1	False

```
print("Trainable layers:", vgg_model.trainable_weights)
Trainable layers: []
```

It is quite clear from the preceding output that all the layers of the VGG-16 model are frozen, which is good because we don't want their weights to change during model training. The last activation feature map in the VGG-16 model (output from `block5_pool`) gives us the bottleneck features, which can then be flattened and fed to a fully connected deep neural network classifier. The following snippet shows what the bottleneck features look like for a sample image from our training data:

```
bottleneck_feature_example = vgg.predict(train_imgs_scaled[0:1])
print(bottleneck_feature_example.shape)
plt.imshow(bottleneck_feature_example[0][:,:,0])

(1, 4, 4, 512)
```

The preceding code generates the following output:
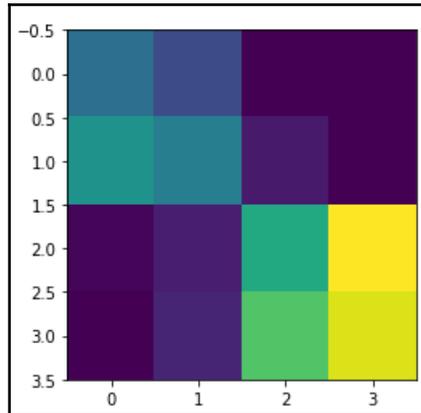
We flatten the bottleneck features in the `vgg_model` object to make them ready to be fed to our fully connected classifier. A way to save time in model training is to use this model and extract out all the features from our training and validation datasets and then feed them as inputs to our classifier. Let's extract out the bottleneck features from our training and validation sets now:

```
def get_bottleneck_features(model, input_imgs):
    features = model.predict(input_imgs, verbose=0)
    return features

train_features_vgg = get_bottleneck_features(vgg_model,
                                             train_imgs_scaled)
validation_features_vgg = get_bottleneck_features(vgg_model,
                                             validation_imgs_scaled)

print('Train Bottleneck Features:', train_features_vgg.shape,
      '\tValidation Bottleneck Features:',
       validation_features_vgg.shape)

Train Bottleneck Features: (3000, 8192) Validation Bottleneck Features:
    (1000, 8192)
```

The preceding output tells us that we have successfully extracted the flattened bottleneck features of dimension 1 x 8,192 for our 3,000 training images and our 1,000 validation images. Let's build the architecture of our deep neural network classifier now, which will take these features as input:

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout,
InputLayer
from keras.models import Sequential
from keras import optimizers

input_shape = vgg_model.output_shape[1]
model = Sequential()
model.add(InputLayer(input_shape=(input_shape,)))
model.add(Dense(512, activation='relu', input_dim=input_shape))
model.add(Dropout(0.3)) model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3)) model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['accuracy'])

model.summary()
_____
Layer (type) Output Shape Param #
=================================================================
input_2 (InputLayer) (None, 8192) 0
_____
dense_1 (Dense) (None, 512) 4194816
_____
dropout_1 (Dropout) (None, 512) 0
_____
dense_2 (Dense) (None, 512) 262656
_____
dropout_2 (Dropout) (None, 512) 0
_____
dense_3 (Dense) (None, 1) 513
=================================================================
```

Just like we mentioned previously, bottleneck feature vectors of size `8192` serve as input to our classification model. We use the same architecture as our previous models here with regard to the dense layers. Let's train this model now:

```
history = model.fit(x=train_features_vgg, y=train_labels_enc,
                    validation_data=(validation_features_vgg,
                                     validation_labels_enc),
                    batch_size=batch_size, epochs=epochs, verbose=1)

Train on 3000 samples, validate on 1000 samples
```

```
Epoch 1/30
3000/3000 – 1s 373us/step – loss: 0.4325 – acc: 0.7897 – val_loss: 0.2958 –
val_acc: 0.8730
Epoch 2/30
3000/3000 – 1s 286us/step – loss: 0.2857 – acc: 0.8783 – val_loss: 0.3294 –
val_acc: 0.8530
...
...
Epoch 29/30
3000/3000 – 1s 287us/step – loss: 0.0121 – acc: 0.9943 – val_loss: 0.7760 –
val_acc: 0.8930
Epoch 30/30
3000/3000 – 1s 287us/step – loss: 0.0102 – acc: 0.9987 – val_loss: 0.8344 –
val_acc: 0.8720
```

We get a model with a validation accuracy of close to **88%**, almost a **5-6%** improvement from our basic CNN model with image augmentation, which is excellent. The model does seem to be overfitting though, and we can check this using the accuracy and loss plots depicted in the following diagram:
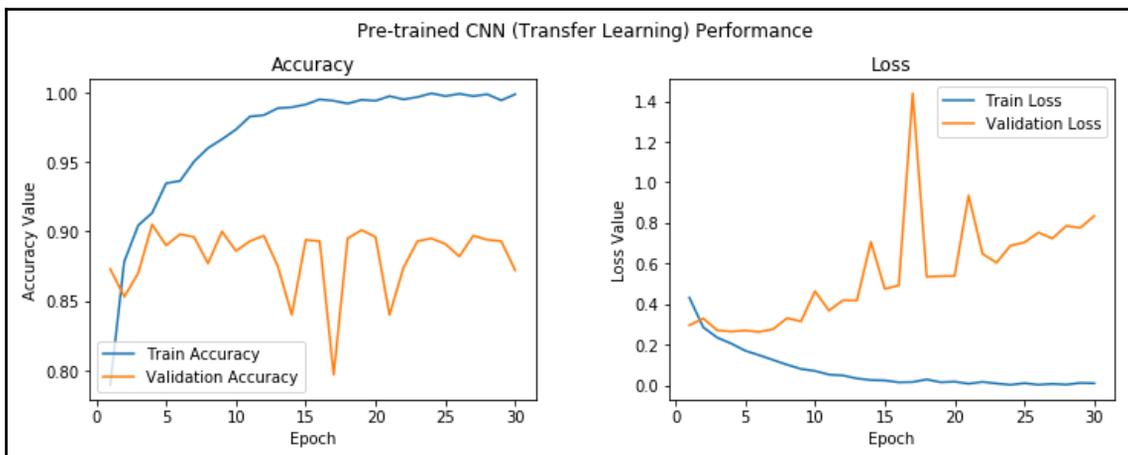
There is a decent gap between the model train and validation accuracy after the fifth epoch, which kind of makes it clear that the model is overfitting on the training data after that. But overall, this seems to be the best model so far, where by leveraging the VGG-16 model as a feature extractor, we seem to get close to **90%** validation accuracy without even using an image augmentation strategy. But we haven't tapped into the complete potential of transfer learning yet. Let's try using our image augmentation strategy on this model. Before that, we save this model to disk using the following code:

```
model.save('cats_dogs_tlearn_basic_cnn.h5')
```

Pretrained CNN model as a feature extractor with image augmentation

We will leverage the same data generators for our train and validation datasets that we used before. The code for building them is depicted as follows for ease of understanding:

```
train_datagen = ImageDataGenerator(rescale=1./255, zoom_range=0.3,
                                   rotation_range=50,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')

val_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow(train_imgs, train_labels_enc,
                                      batch_size=30)
val_generator = val_datagen.flow(validation_imgs,
                                 validation_labels_enc,
                                 batch_size=20)
```

Let's now build our deep learning model architecture. We won't extract the bottleneck features like last time since we will be training on data generators; hence, we will be passing the `vgg_model` object as an input to our own model:

```
model = Sequential()

model.add(vgg_model)
model.add(Dense(512, activation='relu', input_dim=input_shape))
model.add(Dropout(0.3)) model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3)) model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['accuracy'])
```

You can clearly see that everything is the same. We bring the learning rate slightly down since we will be training for 100 epochs and don't want to make any sudden abrupt weight adjustments to our model layers. Do remember that the VGG-16 model's layers are still frozen here and we are still using it as a basic feature extractor only:

```
history = model.fit_generator(train_generator, steps_per_epoch=100,
                              epochs=100,
                              validation_data=val_generator,
                              validation_steps=50,
                              verbose=1)
```

```
Epoch 1/100
100/100 - 45s 449ms/step - loss: 0.6511 - acc: 0.6153 - val_loss: 0.5147 -
val_acc: 0.7840
Epoch 2/100
100/100 - 41s 414ms/step - loss: 0.5651 - acc: 0.7110 - val_loss: 0.4249 -
val_acc: 0.8180
...
...
Epoch 99/100
100/100 - 42s 417ms/step - loss: 0.2656 - acc: 0.8907 - val_loss: 0.2757 -
val_acc: 0.9050
Epoch 100/100
100/100 - 42s 418ms/step - loss: 0.2876 - acc: 0.8833 - val_loss: 0.2665 -
val_acc: 0.9000
```

We can see that our model has an overall validation accuracy of **90%**, which is a slight improvement from our previous model, and also the train and validation accuracy are quite close to each other, indicating that the model is not overfitting. This can be reinforced by looking at the following plots for model accuracy and loss:
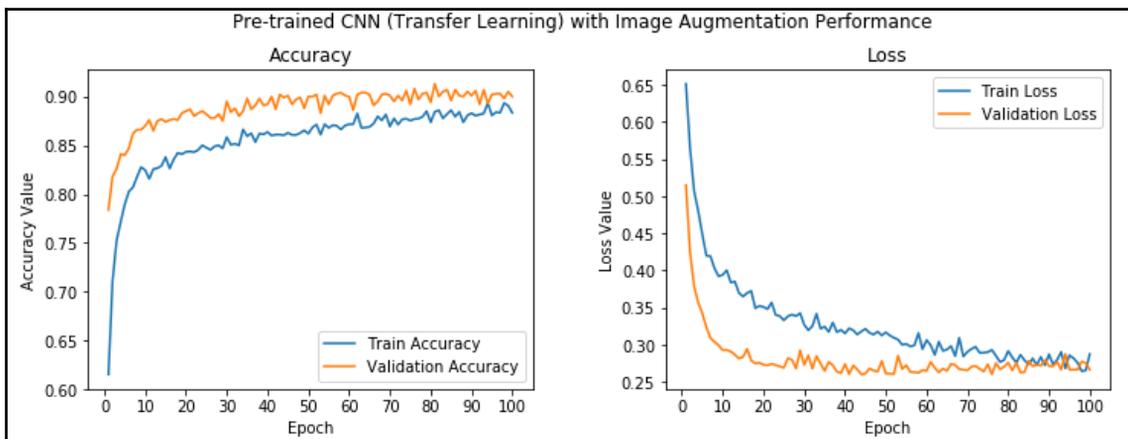
We can clearly see that the values of train and validation accuracy are quite close to each other and the model doesn't overfit. Also, we reach **90%** accuracy, which is neat! Let's save this model on the disk now for future evaluation on the test data:

```
model.save('cats_dogs_tlearn_img_aug_cnn.h5')
```

We will now fine-tune the VGG-16 model to build our last classifier, where we will unfreeze blocks 4 and 5, as we depicted at the beginning of this section.

Pretrained CNN model with fine-tuning and image augmentation

We will now leverage our VGG-16 model object stored in the `vgg_model` variable and unfreeze convolution blocks 4 and 5 while keeping the first three blocks frozen. The following code helps us achieve this:

```
vgg_model.trainable = True
set_trainable = False

for layer in vgg_model.layers:
    if layer.name in ['block5_conv1', 'block4_conv1']:
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False

print("Trainable layers:", vgg_model.trainable_weights)

Trainable layers:
[<tf.Variable 'block4_conv1/kernel:0' shape=(3, 3, 256, 512)
dtype=float32_ref>, <tf.Variable 'block4_conv1/bias:0' shape=(512,)
dtype=float32_ref>,
<tf.Variable 'block4_conv2/kernel:0' shape=(3, 3, 512, 512)
dtype=float32_ref>, <tf.Variable 'block4_conv2/bias:0' shape=(512,)
dtype=float32_ref>,
<tf.Variable 'block4_conv3/kernel:0' shape=(3, 3, 512, 512)
dtype=float32_ref>, <tf.Variable 'block4_conv3/bias:0' shape=(512,)
dtype=float32_ref>,
<tf.Variable 'block5_conv1/kernel:0' shape=(3, 3, 512, 512)
dtype=float32_ref>, <tf.Variable 'block5_conv1/bias:0' shape=(512,)
dtype=float32_ref>,
<tf.Variable 'block5_conv2/kernel:0' shape=(3, 3, 512, 512)
dtype=float32_ref>, <tf.Variable 'block5_conv2/bias:0' shape=(512,)
dtype=float32_ref>,
<tf.Variable 'block5_conv3/kernel:0' shape=(3, 3, 512, 512)
dtype=float32_ref>, <tf.Variable 'block5_conv3/bias:0' shape=(512,)
dtype=float32_ref>]
```

You can clearly see from the preceding output that the convolution and pooling layers pertaining to blocks 4 and 5 are now trainable, and you can also verify which layers are frozen and unfrozen using the following code:

```
layers = [(layer, layer.name, layer.trainable) for layer in
vgg_model.layers] pd.DataFrame(layers, columns=['Layer Type', 'Layer
                        Name', 'Layer Trainable'])
```

The preceding code generates the following output:

	Layer Type	Layer Name	Layer Trainable
0	<keras.engine.topology.InputLayer object at 0x7f26c86b2518>	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0x7f277c9fc080>	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0x7f26c86b26d8>	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0x7f26c86e6c88>	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0x7f26c867dc18>	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0x7f26c8690f28>	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0x7f26c869e5c0>	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0x7f26c863f828>	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0x7f26c863f128>	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0x7f26c86607b8>	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83d7d68>	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0x7f26c83fd358>	block4_conv1	True
12	<keras.layers.convolutional.Conv2D object at 0x7f26c83fddd8>	block4_conv2	True
13	<keras.layers.convolutional.Conv2D object at 0x7f26c839da20>	block4_conv3	True
14	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83ac1d0>	block4_pool	True
15	<keras.layers.convolutional.Conv2D object at 0x7f26c834e978>	block5_conv1	True
16	<keras.layers.convolutional.Conv2D object at 0x7f271a15eb38>	block5_conv2	True
17	<keras.layers.convolutional.Conv2D object at 0x7f26c8371d68>	block5_conv3	True
18	<keras.layers.pooling.MaxPooling2D object at 0x7f26c8314b00>	block5_pool	True
19	<keras.layers.core.Flatten object at 0x7f26c828bda0>	flatten_1	True

We can clearly see that the last two blocks are now trainable, which means the weights for these layers will also get updated with backpropagation in each epoch as we pass each batch of data. We will use the same data generators and model architecture as our previous model and train our model. We reduce the learning rate slightly since we don't want to get stuck at any local minimal, and we also do not want to suddenly update the weights of the trainable VGG-16 model layers by a big factor that might adversely affect the model:

```
# data generators
train_datagen = ImageDataGenerator(rescale=1./255, zoom_range=0.3,
                                   rotation_range=50,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')

val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow(train_imgs, train_labels_enc,
                                     batch_size=30)
val_generator = val_datagen.flow(validation_imgs,
                                 validation_labels_enc,
                                 batch_size=20)

# build model architecture
model = Sequential()

model.add(vgg_model)
model.add(Dense(512, activation='relu', input_dim=input_shape))
model.add(Dropout(0.3)) model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3)) model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['accuracy'])

# model training
history = model.fit_generator(train_generator, steps_per_epoch=100,
                              epochs=100,
                              validation_data=val_generator,
                              validation_steps=50,
                              verbose=1)


Epoch 1/100
100/100 - 64s 642ms/step - loss: 0.6070 - acc: 0.6547 - val_loss: 0.4029 -
val_acc: 0.8250
```

```
Epoch 2/100
100/100 - 63s 630ms/step - loss: 0.3976 - acc: 0.8103 - val_loss: 0.2273 -
val_acc: 0.9030
...
...
Epoch 99/100
100/100 - 63s 629ms/step - loss: 0.0243 - acc: 0.9913 - val_loss: 0.2861 -
val_acc: 0.9620
Epoch 100/100
100/100 - 63s 629ms/step - loss: 0.0226 - acc: 0.9930 - val_loss: 0.3002 -
val_acc: 0.9610
```

We can see from the preceding output that our model has obtained a validation accuracy of around **96%**, which is a **6%** improvement from our previous model. Overall, this model has gained a **24%** improvement in validation accuracy from our first basic CNN model. This really shows how useful transfer learning can be.

Let's observe the model accuracy and loss plots:
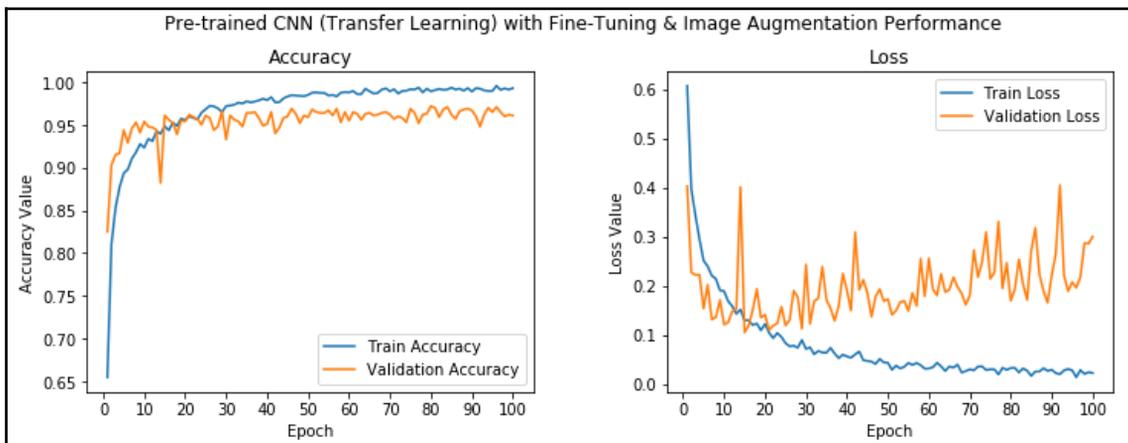
We can see that accuracy values are really excellent here, and although the model looks like it might be slightly overfitting on the training data, we still get great validation accuracy. Let's save this model to disk now using the following code:

```
model.save('cats_dogs_tlearn_finetune_img_aug_cnn.h5')
```

Let's now put all our models to the test by actually evaluating their performance on our test dataset.

Evaluating our deep learning models

We will now evaluate the five different models we built so far by first testing them on a sample test image, then visualizing how a CNN model actually tries to analyze and extract features from the image, and finally by testing each model's performance on our test dataset. The code for this section is available in the `Model Performance Evaluations.ipynb` Jupyter Notebook in case you want to execute the code and follow along with the chapter. We have also built a nifty utility module called `model_evaluation_utils`, which we will be using to evaluate the performance of our deep learning models. Let's load up the following dependencies before getting started:

```
import glob
import numpy as np
import matplotlib.pyplot as plt
from keras.preprocessing.image import load_img, img_to_array, array_to_img
from keras.models import load_model
import model_evaluation_utils as meu

%matplotlib inline
```

Once we have loaded up these dependencies, let's load up our models that we have saved so far:

```
basic_cnn = load_model('cats_dogs_basic_cnn.h5')
img_aug_cnn = load_model('cats_dogs_cnn_img_aug.h5')
tl_cnn = load_model('cats_dogs_tlearn_basic_cnn.h5')
tl_img_aug_cnn = load_model('cats_dogs_tlearn_img_aug_cnn.h5')
tl_img_aug_finetune_cnn =
            load_model('cats_dogs_tlearn_finetune_img_aug_cnn.h5')
```

This helps us in retrieving all the five models we created in this chapter using various techniques and architectures.

Model predictions on a sample test image

We will now load a sample image that is not a part of any of our datasets and try to see the predictions of our different models. I will use an image of my pet cat here, so this will be interesting! Let's load up the sample image and a few basic configurations:

```
# basic configurations
IMG_DIM = (150, 150)
input_shape = (150, 150, 3)
num2class_label_transformer = lambda l: ['cat' if x == 0 else 'dog' for
                                         x in l]
class2num_label_transformer = lambda l: [0 if x == 'cat' else 1 for x
                                         in l]

# load sample image
sample_img_path = 'my_cat.jpg'
sample_img = load_img(sample_img_path, target_size=IMG_DIM)
sample_img_tensor = img_to_array(sample_img)
sample_img_tensor = np.expand_dims(sample_img_tensor, axis=0)
sample_img_tensor /= 255.
print(sample_img_tensor.shape)
plt.imshow(sample_img_tensor[0]) (1, 150, 150, 3)
```

The preceding code generates the following output:
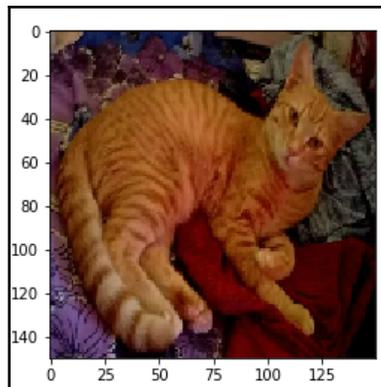
Now that we have our sample image loaded up, let's see what our models predict as the category for this image (my cat):

```
cnn_prediction = num2class_label_transformer(basic_cnn.predict_classes(
                                              sample_img_tensor,
                                              verbose=0))
cnn_img_aug_prediction =
num2class_label_transformer(img_aug_cnn.predict_classes(
                                                 sample_img_tensor,
                                                 verbose=0))
tlearn_cnn_prediction = num2class_label_transformer(tl_cnn.predict_classes(
                                       get_bottleneck_features(vgg_model,
                                       sample_img_tensor),
                                       verbose=0))
tlearn_cnn_img_aug_prediction =
num2class_label_transformer(
                     tl_img_aug_cnn.predict_classes(sample_img_tensor,
                                                 verbose=0))
tlearn_cnn_finetune_img_aug_prediction =
num2class_label_transformer(
             tl_img_aug_finetune_cnn.predict_classes(sample_img_tensor,
                                                 verbose=0))


print('Predictions for our sample image:\n',
      '\nBasic CNN:', cnn_prediction,
      '\nCNN with Img Augmentation:', cnn_img_aug_prediction,
      '\nPre-trained CNN (Transfer Learning):', tlearn_cnn_prediction,
      '\nPre-trained CNN with Img Augmentation (Transfer Learning):',
      tlearn_cnn_img_aug_prediction,
      '\nPre-trained CNN with Fine-tuning & Img Augmentation (Transfer
       Learning):', tlearn_cnn_finetune_img_aug_prediction)



Predictions for our sample image: Basic CNN: ['cat']
CNN with Img Augmentation: ['dog']
Pre-trained CNN (Transfer Learning): ['dog']
Pre-trained CNN with Img Augmentation (Transfer Learning): ['cat']
Pre-trained CNN with Fine-tuning & Img Augmentation (Transfer Learning):
['cat']
```

You can see from the preceding output that three of our models predicted the image correctly as a cat and two of them got it wrong. Interestingly, the basic CNN model also got it correct, along with the pretrained model with fine-tuning and image augmentation as expected.

Visualizing what a CNN model perceives

Deep learning models are often known as **black box models** because it is very difficult to actually interpret how the model is working internally compared to simpler ML models, such as decision trees. We know that CNN-based deep learning models use convolution layers, which use filters to extract activation feature maps representing a spatial hierarchy of features. Conceptually, the top convolutional layers learn small local patterns, and the layers that are lower in the network learn more complex and larger patterns, which are obtained from the top convolution layers. Let's try to visualize this with an example.

We will take our best model (transfer learning with fine-tuning and image augmentation) and try to extract the output activation feature maps from the first eight layers. Essentially, this would end up giving us convolution and pooling layers from the first three blocks of the VGG-16 model, since we used the same for feature extraction in our model.

To get a peek at the layers, you can use the following code:

```
tl_img_aug_finetune_cnn.layers[0].layers[1:9]

[<keras.layers.convolutional.Conv2D at 0x7f514841b0b8>,
<keras.layers.convolutional.Conv2D at 0x7f514841b0f0>,
<keras.layers.pooling.MaxPooling2D at 0x7f5117d4bb00>,
<keras.layers.convolutional.Conv2D at 0x7f5117d4bbe0>,
<keras.layers.convolutional.Conv2D at 0x7f5117d4bd30>,
<keras.layers.pooling.MaxPooling2D at 0x7f5117d4beb8>,
<keras.layers.convolutional.Conv2D at 0x7f5117d4bf98>,
<keras.layers.convolutional.Conv2D at 0x7f5117d00128>]
```

Let's now extract the feature maps from our model, based on what it tries to extract from our sample test image of my cat. For a simple illustration, we extract the output after the first convolution layer from block 1 and view some of the activation feature maps from the same in the following snippet:

```
from keras import models

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in
                    tl_img_aug_finetune_cnn.layers[0].layers[1:9]]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(
inputs=tl_img_aug_finetune_cnn.layers[0].layers[1].input,
                    outputs=layer_outputs)

# This will return a list of 8 Numpy arrays
# one array per layer activation
```

```
activations = activation_model.predict(sample_img_tensor)
print('Sample layer shape:', activations[0].shape)
print('Sample convolution (activation map) shape:',
                          activations[0][0, :, :, 1].shape)

fig, ax = plt.subplots(1,5, figsize=(16, 6))
ax[0].imshow(activations[0][0, :, :, 10], cmap='bone')
ax[1].imshow(activations[0][0, :, :, 25], cmap='bone')
ax[2].imshow(activations[0][0, :, :, 40], cmap='bone')
ax[3].imshow(activations[0][0, :, :, 55], cmap='bone')
ax[4].imshow(activations[0][0, :, :, 63], cmap='bone')

Sample layer shape: (1, 150, 150, 64)
Sample convolution (activation map) shape: (150, 150)
```

The preceding code generates the following output:
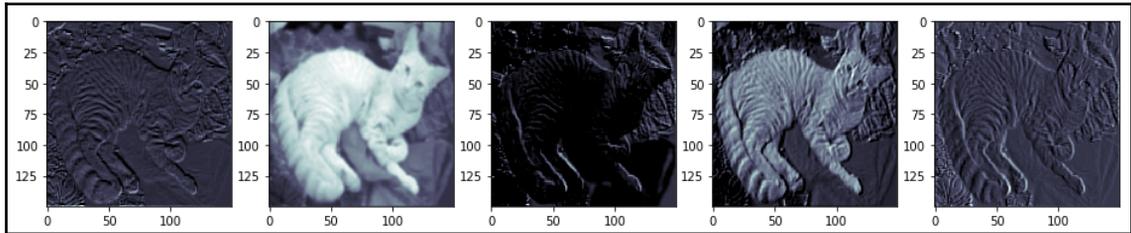
From the preceding output, we can clearly see that the output from the first convolution layer gives us a total of 64 activation feature maps of size 150 x 150 each. We visualize five of these feature maps in the preceding snippet. You can see how the model tries to extract relevant features pertaining to the image, such as hue, intensity, edges, corners, and so on. The following output depicts some more activation maps of the VGG-16 model from blocks 1 and 2:
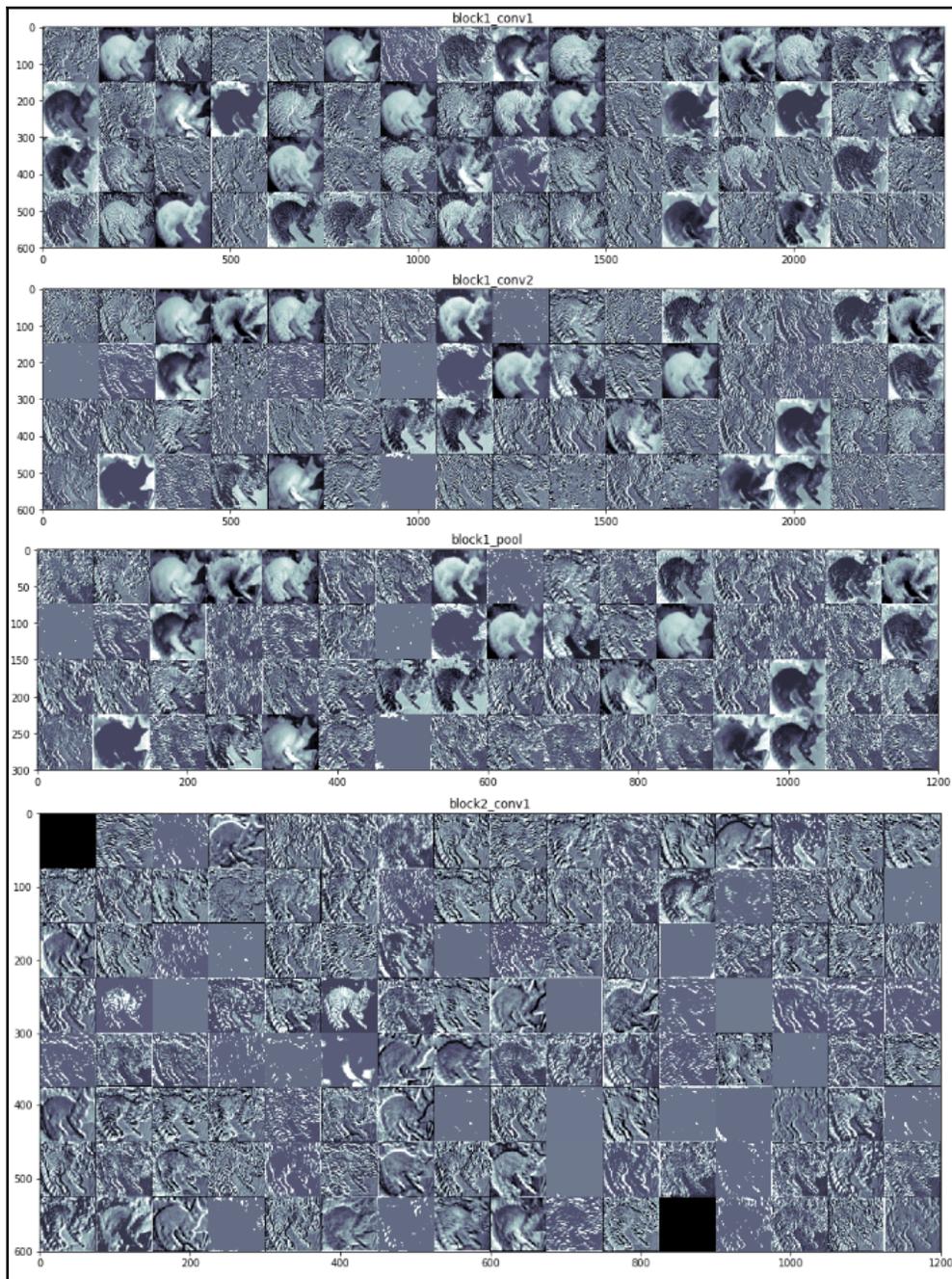
To get the aforementioned activation feature maps, we have leveraged a code snippet that is available in the `Model Performance Evaluations.ipynb` Jupyter Notebook, thanks to Francois Chollet and his book *Deep Learning with Python*, which can help visualize all the selected layers in our CNN model. We have visualized the top eight layers of our model, which we selected earlier in the notebook, but we have shown activation maps of the first two blocks here. Feel free to check out the notebook and reuse the same code for your own models. From the preceding screenshot, you might see that top layer feature maps usually retain a lot of the original image but as you go deeper in the model, the feature maps become more abstract, complex, and less interpretable.

Evaluation model performance on test data

It's time now for the final test, where we literally test the performance of our models by making predictions on our test dataset. Let's load up and prepare our test dataset first before we try making predictions:

```
IMG_DIM = (150, 150)
test_files = glob.glob('test_data/*')
test_imgs = [img_to_array(load_img(img, target_size=IMG_DIM))
                         for img in test_files]
test_imgs = np.array(test_imgs)

test_labels = [fn.split('/')[1].split('.')[0].strip() for fn in test_files]
test_labels_enc = class2num_label_transformer(test_labels)
test_imgs_scaled = test_imgs.astype('float32')
test_imgs_scaled /= 255

print('Test dataset shape:', test_imgs.shape)

Test dataset shape: (1000, 150, 150, 3)
```

Now that we have our scaled dataset ready, let's evaluate each model by making predictions for all the test images, and then evaluate the model performance by checking how accurate the predictions are:

```
# Model 1 - Basic CNN
predictions = basic_cnn.predict_classes(test_imgs_scaled, verbose=0)
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=predictions,
                                      classes=list(set(test_labels)))
```

The preceding code generates the following output:

```
Model Performance metrics: Model Classification report:                    Prediction Confusion Matrix:
-------------------------- ----------------------------                    -----------------------------
Accuracy: 0.776                         precision   recall  f1-score  support          Predicted:
Precision: 0.7769                                                                        cat  dog
Recall: 0.776                     cat      0.76      0.80     0.78       500 Actual: cat  402   98
F1 Score: 0.7758                  dog      0.79      0.75     0.77       500        dog  126  374

                          avg / total      0.78      0.78     0.78      1000
```

```
# Model 2 - Basic CNN with Image Augmentation
predictions = img_aug_cnn.predict_classes(test_imgs_scaled, verbose=0)
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=predictions,
                                      classes=list(set(test_labels)))
```

The preceding code generates the following output:

```
Model Performance metrics: Model Classification report:                    Prediction Confusion Matrix:
-------------------------- ----------------------------                    -----------------------------
Accuracy: 0.844                         precision   recall  f1-score  support          Predicted:
Precision: 0.844                                                                         cat  dog
Recall: 0.844                     cat      0.84      0.84     0.84       500 Actual: cat  422   78
F1 Score: 0.844                   dog      0.84      0.84     0.84       500        dog   78  422

                          avg / total      0.84      0.84     0.84      1000
```

```
# Model 3 - Transfer Learning (basic feature extraction)
test_bottleneck_features = get_bottleneck_features(vgg_model,
test_imgs_scaled) predictions =
tl_cnn.predict_classes(test_bottleneck_features, verbose=0) predictions =
num2class_label_transformer(predictions)

meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=predictions,
                                      classes=list(set(test_labels)))
```

The preceding code generates the following output:

```
Model Performance metrics:  Model Classification report:                    Prediction Confusion Matrix:
----------------------------------------------------------------            --------------------------------
Accuracy: 0.888                         precision    recall  f1-score   support           Predicted:
Precision: 0.8898                                                                            cat   dog
Recall: 0.888                     cat      0.92       0.85      0.88       500    Actual: cat    427    73
F1 Score: 0.8879                  dog      0.86       0.92      0.89       500            dog     39   461

                            avg / total    0.89       0.89      0.89      1000
```

```
# Model 4 – Transfer Learning with Image Augmentation
predictions = tl_img_aug_cnn.predict_classes(test_imgs_scaled, verbose=0)
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=predictions,
                                      classes=list(set(test_labels)))
```

The preceding code generates the following output:

```
Model Performance metrics:  Model Classification report:                    Prediction Confusion Matrix:
----------------------------------------------------------------            --------------------------------
Accuracy: 0.898                         precision    recall  f1-score   support           Predicted:
Precision: 0.8981                                                                            cat   dog
Recall: 0.898                     cat      0.89       0.91      0.90       500    Actual: cat    453    47
F1 Score: 0.898                   dog      0.90       0.89      0.90       500            dog     55   445

                            avg / total    0.90       0.90      0.90      1000
```

```
# Model 5 – Transfer Learning with Fine-tuning & Image Augmentation
predictions = tl_img_aug_finetune_cnn.predict_classes(test_imgs_scaled,
                                                      verbose=0)
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test_labels,
                                      predicted_labels=predictions,
                                      classes=list(set(test_labels)))
```

```
Model Performance metrics:  Model Classification report:                    Prediction Confusion Matrix:
----------------------------------------------------------------            --------------------------------
Accuracy: 0.961                         precision    recall  f1-score   support           Predicted:
Precision: 0.9611                                                                            cat   dog
Recall: 0.961                     cat      0.97       0.95      0.96       500    Actual: cat    476    24
F1 Score: 0.961                   dog      0.95       0.97      0.96       500            dog     15   485

                            avg / total    0.96       0.96      0.96      1000
```

We can see that we definitely have some interesting results. Each subsequent model performs better than the previous model, which is expected since we tried more advanced techniques with each new model. Our worst model is our basic CNN model, with a model accuracy and F1-score of around **78%**, and our best model is our fine-tuned model with transfer learning and image augmentation, which gives us a model accuracy and F1-score of **96%**, which is really amazing considering we trained our model from our 3,000 image training dataset. Let's plot the ROC curves of our worst and best models now:

```
# worst model - basic CNN
meu.plot_model_roc_curve(basic_cnn, test_imgs_scaled,
                         true_labels=test_labels_enc, class_names=[0,
                                                                    1])


# best model - transfer learning with fine-tuning & image augmentation
meu.plot_model_roc_curve(tl_img_aug_finetune_cnn, test_imgs_scaled,
                         true_labels=test_labels_enc, class_names=[0,
                                                                    1])
```

We get the plots as follows:
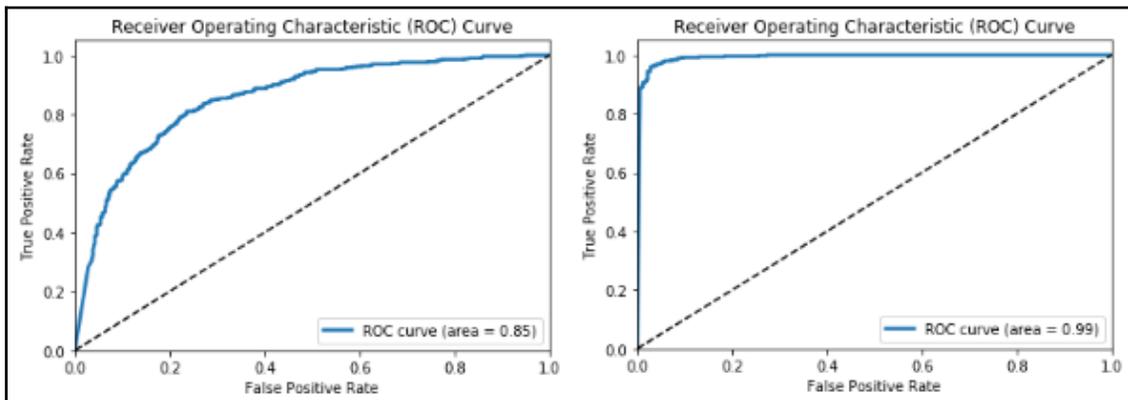
This should give you a good idea of how much of a difference pretrained models and transfer learning can make, especially in tackling complex problems when we have constraints like less data. We encourage you to try out similar strategies with your own data.

Summary

The intent of this chapter was to give you a more hands-on perspective of building deep learning models to solve a real-world problem, and also to understand the effectiveness of transfer learning. We covered aspects of the need for transfer learning, especially when solving problems with the constraints of limited data. We built several CNN models from scratch and also saw the benefits of a proper image augmentation strategy. We also looked at how to leverage pretrained models for transfer learning and covered the various ways to use them, including as feature extractors, as well as fine-tuning. We saw the detailed architecture of the VGG-16 model and how to leverage the model as an efficient image feature extractor. Strategies pertaining to transfer learning, including feature extraction and fine-tuning, along with image augmentation, were used to build effective deep learning image classifiers.

Last but not least, we evaluated all our models on our test dataset and also gained some perspective of how convolutional neural networks visualize images internally when building feature maps. In subsequent chapters, we will look at more complex real-world case studies that require transfer learning. Stay tuned!

6

Image Recognition and Classification

An investment in knowledge always pays the best interest.

– Benjamin Franklin

Image recognition is an active interdisciplinary field of study under the umbrella of computer vision. Image or object recognition, as the name suggests, is the task of identifying objects in an image or video sequence. Traditionally, this field has leveraged advancements in mathematical and computer-aided modeling, and the design of objects. Several hand-annotated datasets have been developed over the years to test and evaluate image recognition systems. Traditional techniques, as we now call them, were dominating the scene and iteratively improving upon the task until recently. In 2012, deep learning arrived at the ImageNet competition and opened the floodgates for rapid improvements and advancements in computer vision and deep learning techniques.

In this chapter, we will introduce the concept of image recognition and classification from the lenses of deep learning, and particularly transfer learning. This chapter will cover the following aspects:

- Introduction to image classification using deep learning
- Benchmarking datasets
- State-of-the-art deep image classification models
- Image classification and transfer learning use cases

This chapter begins part three of this book. In this part of the book, we will cover case studies involving concepts and techniques discussed in the previous two parts. The use cases will present real-world topics/research areas and help you learn how transfer learning can be utilized in different settings. The code for this chapter is available for quick reference in the `Chapter 6` folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

Deep learning-based image classification

Convolutional Neural Networks (**CNNs**) are at the heart of this deep learning revolution for improving the task of image classification. CNNs are specialized neural networks to handle image data. As a quick brush-up, CNNs help us infer shift and space invariant features through their shared weight architectures, and are basically a variant of feed forward networks. We have already covered the basics of CNNs in detail in `Chapter 3`, *Understanding Deep Learning Architectures*, and `Chapter 5`, *Unleashing the Power of Transfer Learning*. Before we move on, readers are encouraged to have a quick refresher for a better understanding. The following image showcases a typical CNN in action:
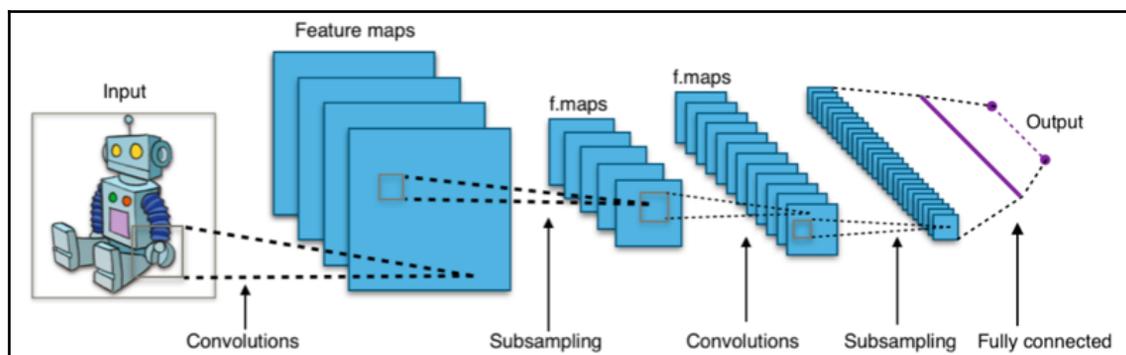
A typical CNN [Source: https://en.wikipedia.org/wiki/File:Typical_cnn.png]

Neural networks arrived on the scene of image classification competitions as early as 2011. GPU-trained networks were starting to win competitions. It was in 2012, for the first time, when a deep CNN improved upon the previous best to 83% on the ImageNet image classification task that the world took notice. The results were amazing enough to catch global attention and help with the proliferation of use cases being solved using deep learning.

Benchmarking datasets

Image classification, or, for that matter, any classification task, is inherently a supervised learning task. Supervised tasks learn about the different classes through the underlying training sets available.

Even though CNNs are optimized feed forward networks that share weights, the number of parameters to train in a deep ConvNet might be huge. This is one of the reasons why huge training sets are required to achieve better performing networks. Luckily, research groups across the globe have been working towards collecting, hand-annotating, and crowdsourcing different datasets. These datasets are utilized to benchmark performance of different algorithms, as well as to identify winners in different competitions.

The following is a brief listing of widely accepted benchmarking datasets in the field of image classification:

- **ImageNet**: With over 14 million hand-annotated high-resolution colored images spanning 20,000 categories, this is a gold-standard visual dataset. It was designed for use in visual object identification tasks by the computer science department at Princeton University in 2009. Since then, this dataset (in its trimmed version of 1,000 non-overlapping classes) has been used as the basis of the *ImageNet Large Scale Visual Recognition Challenge* (`https://arxiv.org/abs/1409.0575`).
- **80 Million Tiny Images dataset**: As the name suggests, this MIT dataset contains 80 million images collected from the internet and tagged to more than 75,000 different non-abstract English nouns. This dataset also forms the basis for multiple other widely used datasets, including the CIFAR datasets.
- **CIFAR-10**: Developed by the Canadian Institute for Advanced Research, CIFAR-10 is one of the most widely used datasets for **machine learning** (**ML**) research. This dataset contains 60,000 low-resolution images spanning across 10 non-overlapping classes.
- **CIFAR-100**: From the same research group, this dataset contains 60,000 images evenly spread across 100 different classes.
- **Common Objects in Context**: **Common Object in Context** (**COCO**) is a large-scale visual database for object identification, segmentation, and captioning. This dataset contains more than 200,000 labeled images spanning different classes.
- **Open Images**: This is possibly one of biggest annotated datasets available for use. Version 4 of this dataset contains more than 9 million annotated images.
- **Caltech 101 and Caltech 256**: These datasets contain annotated images spanning across 101 and 256 categories respectively. Caltech 101 contains around 9,000 images, while Caltech 256 contains close to 30,000.

- **Stanford Dog dataset**: This is an interesting dataset specific to different dog breeds. It contains 20,000 colored images spanning across 120 different dog breeds.
- **MNIST**: One of the most famous visual datasets of all time, MNIST has become the defacto, *Hello, World* dataset for ML enthusiasts. It contains over 60,000 hand-labeled digits (zero to nine).

The preceding list is just the tip of the iceberg. There are numerous other datasets capturing different aspects of the world. Preparing these datasets is a painful and time-consuming process, but these datasets are what make deep learning so successful in its current form. Readers are encouraged to explore these and other such datasets in detail to understand the nuances behind them and the challenges each of these datasets pose for us to solve. We will be utilizing some of these datasets in this and coming chapters to understand transfer learning concepts.

State-of-the-art deep image classification models

Deep learning has garnered much attention and hype over the years. It is no surprise that a ton of research work is being shared in reputed competitions, conferences, and journals worldwide centered around deep learning. It is particularly the image classification architectures that have been enjoying the spotlight for some years now, with iterative improvements being shared on a regular basis. Let us have a quick look at some of the best-performing and popular state-of-the-art deep image classification architectures:

- **AlexNet**: This is the network that can be credited for *opening the floodgates*. Designed by one of the pioneers of deep learning, Geoffrey Hinton and team, this network reduced the top-five error rate to just 15.3%. It was also one of the first architectures to leverage GPUs for speeding up the learning process.
- **VGG-16**: The network from Oxford's Visual Geometry Group is one of the best-performing architectures, widely used for benchmarking other designs. VGG-16 utilizes a simple architecture based on 3 x 3 convolutional layers stacked one over the other (16 times), followed by a max pooling layer to achieve formidable performance. This model was succeeded by a slightly more complex model named **VGG19**.

- **Inception**: Also known as **GoogleNet**, this network was introduced in the **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**) in 2014, and achieved a top-five error rate of 6.67%. It was one of the first architectures to achieve near human performance. The novelty behind this network was the use of an inception layer, which involves concatenation of different-sized kernels at the same level.
- **ResNet**: Introduced by Microsoft Research Asia, the **residual network** (**ResNet**) was a novel architecture utilizing batch normalization and skipping connections to achieve a top-five error rate of just 3.57%. It is many times deeper (152 layers) and more complex than simpler architectures like VGG.
- **MobileNet**: While most architectures are in the arms race to outperform others, each new complex network requires even more compute power and data resources. MobileNet deviates from such architectures and is designed to be suitable for mobile and embedded systems. This network utilizes a novel idea of using depth-wise separable convolutions to reduce the overall number of parameters required to train a network.

We have provided a quick overview and recap of some of the state-of-the-art architectures in the deep learning-based image classification space. For a detailed discussion, readers can check out the *Convolutional Neural Networks* section in `Chapter 3`, *Understanding Deep Learning Architectures*.

Image classification and transfer learning

So far, we have discussed what image classification is all about. In this section, we will get our hands dirty by building our own classifiers. In one of the earlier sections of the chapter, we briefly mentioned famous benchmarking datasets, including CIFAR-10 and Stanford Dogs datasets, which we will be concentrating on in the coming sections. We will also utilize pretrained models to understand how we can leverage transfer learning to improve upon our models.

CIFAR-10

CIFAR-10 is one of the most widely used image datasets in the world of deep learning. Prepared by the Canadian Institute for Advanced Research, this is a fairly decent-sized dataset. The major advantage of this dataset is that it contains a balanced distribution of 10 non-overlapping categories. The images are of low resolution and size, and hence the dataset can be used on for training on fairly small memory footprint systems.

Building an image classifier

CIFAR-10 is one of the few balanced datasets available. It has an overall size of 60,000 images. The following snippet loads the CIFAR-10 dataset, and sets the train and test variables:

```
# load CIFAR dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

The images in the dataset are of low resolution, and sometimes even difficult for a human to label. The code shared in this section is available in the IPython Notebook `CIFAR-10_CNN_Classifier.ipynb`.

We have discussed CNNs and how they are optimized for visual datasets. CNNs work upon the principles of weight sharing to reduce the number of parameters; developing them from scratch requires not just strong deep learning skills but huge infrastructure requirements as well. Keeping this in mind, it would be interesting to develop a CNN from scratch and test our skills.

The following snippet showcases a very simple CNN with just five layers (two convolutional, one max pooling, one dense, and one final softmax layer) built using Keras:

```
model = Sequential()
model.add(Conv2D(16, kernel_size=(3, 3),
                activation='relu',
input_shape=INPUT_SHAPE))

model.add(Conv2D(32, (3,3), padding='same',
kernel_regularizer=regularizers.l2(WEIGHT_DECAY),
                                activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.2))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(NUM_CLASSES, activation='softmax'))
```

To improve overall generalization performance, the model also contains a `BatchNormalization` layer along with a `DropOut` layer. These layers help us keep overfit in check, and also prevent the network from memorizing the dataset itself.

We ran the model with just 25 epochs to achieve approximately 65% accuracy on the validation set. The following screenshot shows the output predictions from the trained model:
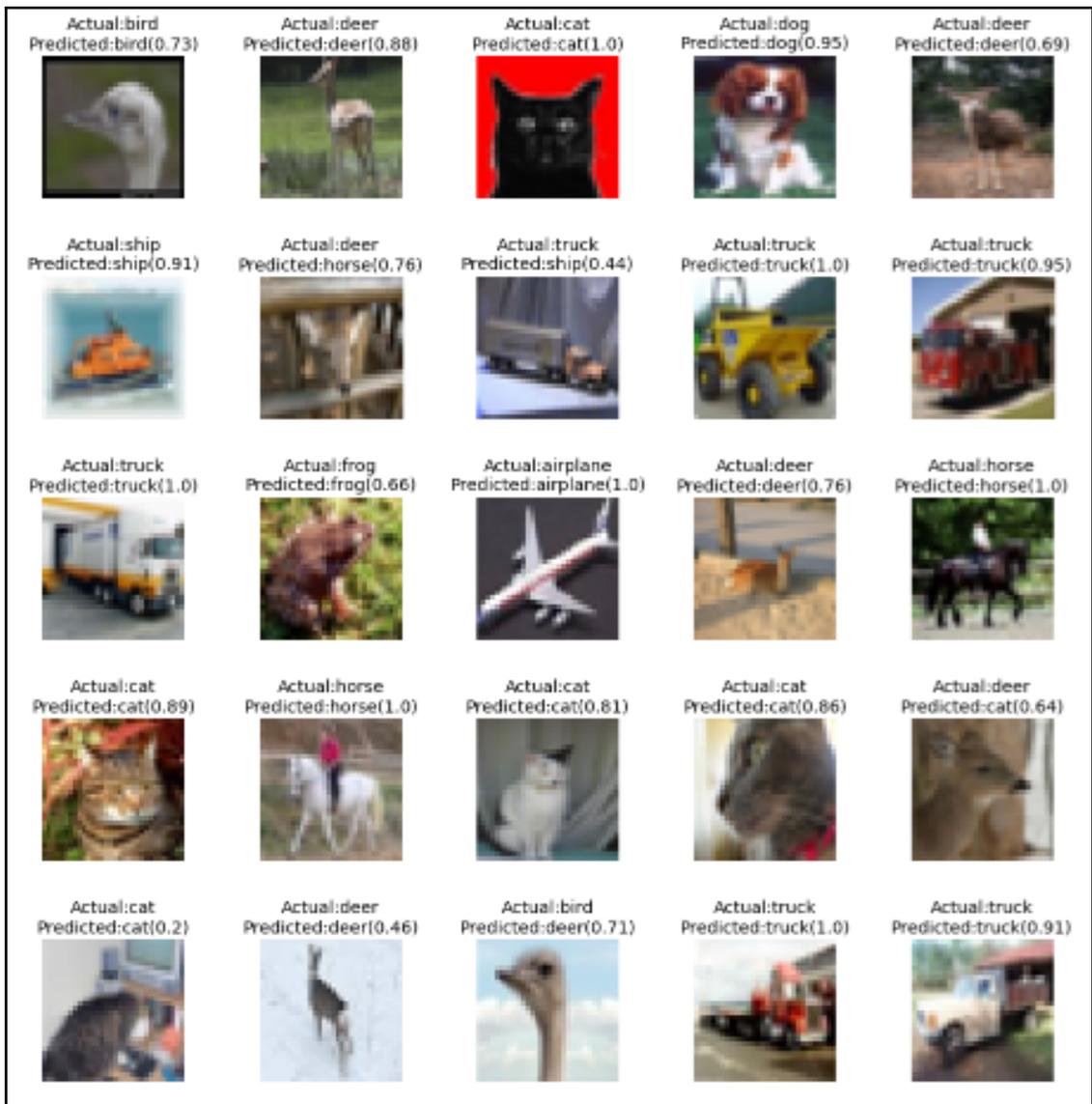
Predictions from CNN based CIFAR-10 classifier

The results are decent enough, though by no means close to state-of-the-art results. Readers should keep in mind that this CNN was just to showcase the immense potential at hand, and we encourage you to try experimenting on the same lines.

Transferring knowledge

Since this chapter and book is focused upon transfer learning, let's quickly get on with the actual task of leveraging and transferring learned information. We have discussed different state-of-the-art CNN architectures in the previous section. Let's now leverage the VGG-16 model, trained on ImageNet, to classify images from the CIFAR-10 dataset. The code for this section is available in the IPython Notebook
`CIFAR10_VGG16_Transfer_Learning_Classifier.ipynb`.

ImageNet is a huge visual dataset with over 20,000 different categories. CIFAR-10, on the other hand, is restricted to only 10 non-overlapping categories. A powerful network like VGG-16 requires immense computational power and time to train to perform better than humans. This brings transfer learning into the picture. Since most of us do not have access to unlimited compute, we can leverage these networks under two distinct settings:

- Use pretrained state-of-the-art networks as feature extractors. This is done by removing the top classification layer and using the output from the penultimate layer.
- Fine-tuning the state-of-the-art network on the new dataset.

We will utilize VGG-16 as a feature extractor and build a custom classifier on top of it. The following snippet loads and prepares the CIFAR-10 dataset for use:

```
# extract data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

#split train into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train,
                                                  y_train,
                                                  test_size=0.15,
                                                  stratify=np.array
                                                          (y_train),
                                                  random_state=42)


# perform one hot encoding
Y_train = np_utils.to_categorical(y_train, NUM_CLASSES)
Y_val = np_utils.to_categorical(y_val, NUM_CLASSES)
Y_test = np_utils.to_categorical(y_test, NUM_CLASSES)

# Scale up images to 48x48
X_train = np.array([sp.misc.imresize(x,
                                      (48, 48)) for x in X_train])
X_val = np.array([sp.misc.imresize(x,
                                   (48, 48)) for x in X_val])
X_test = np.array([sp.misc.imresize(x,
                                    (48, 48)) for x in X_test])
```

The preceding code snippet not only splits the training dataset into train and validation sets, but also transforms the target variable into one-hot encoded form. We also resize the images to 48 x 48 from 32 x 32, to comply with VGG-16 input requirements. Once the train, validation, and test datasets are ready, we can work toward preparing our classifier.

The following code snippet shows how easily we can attach a new layer(s) on top of an existing model. As our objective is to train the classification layer only, we freeze the rest of the layers by setting the parameter trainable to `False`. This allows us to leverage the existing architectures even on less powerful infrastructures, and transfer the learned weights from one domain to another:

```
base_model = vgg.VGG16(weights='imagenet',
include_top=False,
input_shape=(48, 48, 3))

# Extract the last layer from third block of vgg16 model
last = base_model.get_layer('block3_pool').output

# Add classification layers on top of it
x = GlobalAveragePooling2D()(last)
x= BatchNormalization()(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.6)(x)
pred = Dense(NUM_CLASSES, activation='softmax')(x)
model = Model(base_model.input, pred)

for layer in base_model.layers:
layer.trainable = False
```

We have the basic ingredients in place. One final block remaining in this overall pipeline is data augmentation. The overall dataset contains just 60,000 images; data augmentation comes in handy to add certain variations to the sample set at hand. These variations enable the network to learn more generalized features than otherwise. The following code snippet utilizes the `ImageDataGenerator()` utility to prepare train and validation augmentation objects:

```
# prepare data augmentation configuration
train_datagen = ImageDataGenerator(rescale=1. / 255,
                                   horizontal_flip=False)

train_datagen.fit(X_train)
train_generator = train_datagen.flow(X_train,
                                     Y_train,
                                     batch_size=BATCH_SIZE)
```

```
val_datagen = ImageDataGenerator(rescale=1. / 255,
                                 horizontal_flip=False)

val_datagen.fit(X_val)
val_generator = val_datagen.flow(X_val,
                                 Y_val,
                                 batch_size=BATCH_SIZE)
```

Let's now train the model for a few epochs and measure its performance. The following code snippet calls the `fit_generator()` function to train the newly added layer to the model:

```
train_steps_per_epoch = X_train.shape[0] // BATCH_SIZE
val_steps_per_epoch = X_val.shape[0] // BATCH_SIZE

history = model.fit_generator(train_generator,
                              steps_per_epoch=train_steps_per_epoch,
                              validation_data=val_generator,
                              validation_steps=val_steps_per_epoch,
                              epochs=EPOCHS,
                              verbose=1)
```

The history object returned by `fit_generator()` contains details about each epoch. We utilize these to plot the overall model performance in terms of accuracy and loss. The results are shown as follows:
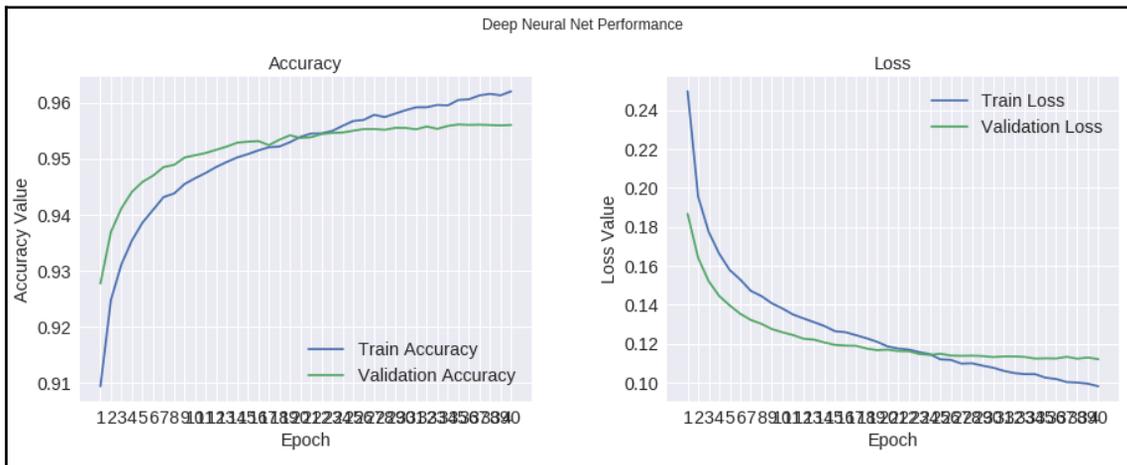
Train-validation performance

As we can see, transfer learning has helped us achieve an amazing boost in the overall performance, compared to the model developed from scratch. This improvement utilized trained weights of VGG-16 to transfer the learned features to this domain. Readers can utilize the same utility `plot_predictions()` to visualize classification results on a random sample, as shown in the following screenshot:
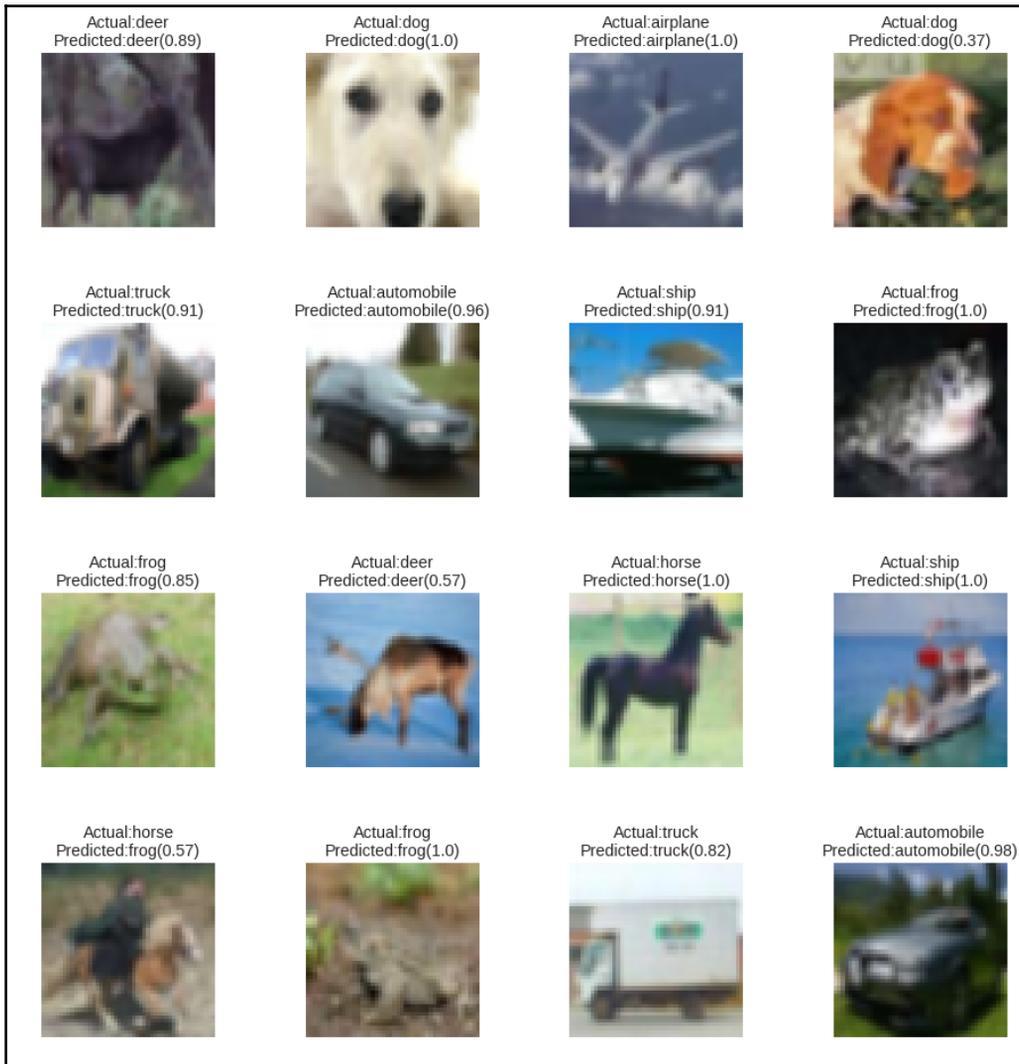
Prediction results from classifier built using VGG-16 based transfer learning

> Neural networks are fairly complex learning machines, which are very difficult to debug and optimize. Though there are a number of techniques available, it requires experience to fine-tune a network. In the current scenario, using a deep CNN like VGG-16 might have been an overkill on images of such small size, yet it shows the immense potential. Use it wisely!

This was a quick and simple application of transfer learning, where we utilized an amazingly complex deep CNN like VGG-16 to prepare a CIFAR-10 classifier. Readers are encouraged to try out not just different configurations of the custom classifier, but rather to even try out different pretrained networks to understand the complexities involved.

Dog Breed Identification dataset

In the previous section, we utilized a low-resolution image dataset to classify images amongst 10 non-overlapping categories. It was by no means a simple task, yet we achieved decent performance with minimal effort.

Now let us level up the game and make this task of image classification even more exciting. In this section, we will be concentrating toward the task of fine-grained image classification. Unlike usual image classification tasks, fine-grained image classification refers to the task of recognizing different subclasses within a higher-level class.

To help understand this task better, we will be focusing our discussion around the **Stanford Dogs** dataset (`http://vision.stanford.edu/aditya86/ImageNetDogs/`). This dataset, as the name suggests, contains images of different dog breeds. In this case, the task is to identify each of those dog breeds. Hence, the high-level concept is the dog itself, while the task is to categorize different subconcepts or subclasses—in this case, breeds—correctly. This dataset contains 20,000 labeled images from the ImageNet dataset consisting of 120 different dog breeds. For the purposes of this discussion, we will be leveraging the dataset available through **Kaggle**. The dataset is available at the following link: `https://www.kaggle.com/c/dog-breed-identification/data`.

Let's get started with our task of building a dog classifier. But before the actual model, let's perform a quick exploratory analysis on the dataset itself for a better understanding.

Exploratory analysis

We cannot emphasize enough how important it is to understand the underlying dataset. In the current scenario, we are dealing with a visual dataset consisting of over 10,000 samples spread across 120 classes (dog breeds). Readers can refer to all the steps related to exploratory analysis in the IPython Notebook titled `dog_breed_eda.ipynb`.

Since this is a visual dataset, let's first visualize a few samples from the dataset. There are multiple ways to ingest and visualize image data in Python; we will be relying on SciPy and matplotlib-related utilities to do so. The following snippet imports the required libraries:

```
In [1]: import os
   ...: import scipy as sp
   ...: import numpy as np
   ...: import pandas as pd
   ...:
   ...: import PIL
   ...: import scipy.ndimage as spi
   ...:
   ...: import matplotlib.pyplot as plt
   ...: import seaborn as sns
   ...:
   ...:np.random.seed(42)
```

Since the dataset is huge, we prepare a couple of utilities to load a random batch of images and to display the selected batch. The utility functions are titled `load_batch()` and `plot_batch()`; details of these are available in the IPython Notebook. The following snippet plots the random batch for reference:

```
In [7]:batch_df = load_batch(dataset_df,
   ...:                       batch_size=36)

In [8]:plot_batch(batch_df, grid_width=6, grid_height=6
   ...:           ,im_scale_x=64, im_scale_y=64)
```

The generated output is as shown in the grid as follows:
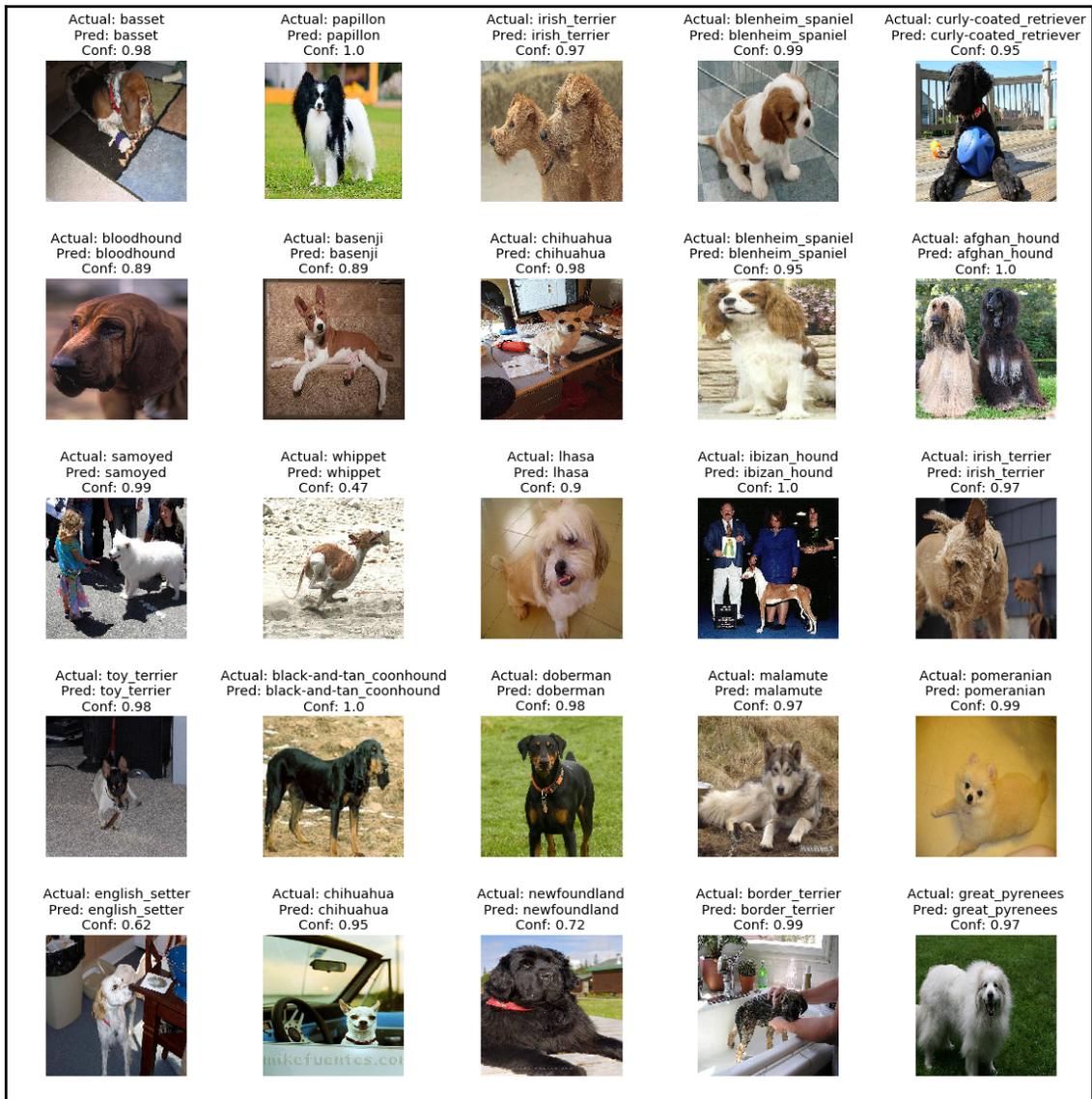
Sample images from Dog Breed Identification dataset

From the preceding grid, we can see that there is a lot of variation, in terms of resolution, lighting, zoom levels, and so on, available along with the fact that images do not just contain just a single dog but other dogs and surrounding items as well. We also need to understand how different the image dimensions are. Using the following snippet, we generate a scatter plot to understand them:

```
In [12]: plt.plot(file_dimension_list[:, 0],
                   file_dimension_list[:, 1], "ro")
   ...: plt.title("Image sizes")
   ...: plt.xlabel("width")
   ...: plt.ylabel("height")
```

The generated scatter plot is as shown as follows. We can clearly see that the maximum number of images fall within 500 x 500 dimension, but we do have variations in shapes:
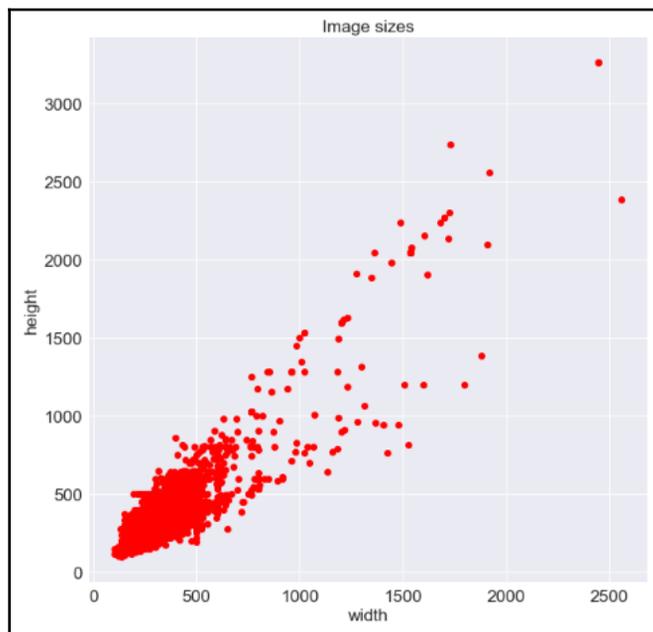
Scatter plot for image size

The dog breed distribution also needs to be checked to understand what we are dealing with. Since we have a labeled dataset, we can easily check. The following snippet uses `pandas` to plot the breed distribution:

```
In [13]: fig = plt.figure(figsize = (12,5))
    ...:
    ...: ax1 = fig.add_subplot(1,2, 1)
    ...:dataset_df.breed.value_counts().tail().plot('bar',
    ...:                  ax=ax1,color='gray',
    ...:                    title="Breeds with Lowest Counts")
    ...:
    ...: ax2 = fig.add_subplot(1,2, 2)
    ...:dataset_df.breed.value_counts().head().plot('bar',
    ...:                ax=ax2,color='black',
    ...:                  title="Breeds with Highest Counts")
```

The dataset is not evenly split; certain breeds have more samples compared to others. This is evident from the plot in the following diagram:
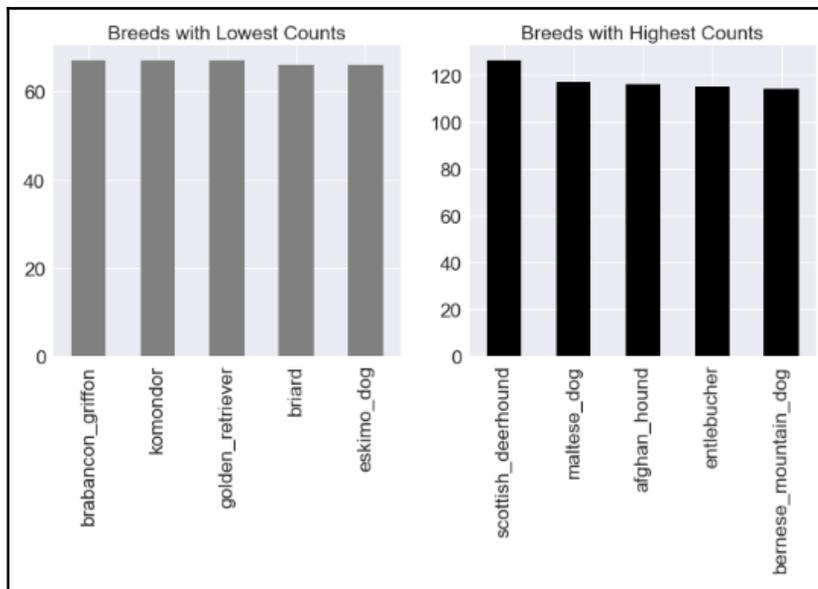
Highest and lowest dog breed sample count

Such a dataset requires a thorough exploration. We have covered a few of the exploratory steps in this section. Further steps are listed/performed in the referred IPython Notebook. Readers are encouraged to go through the steps to understand the impact of image resizing, how different layers detect different features, grayscaling, and so on.

Data preparation

Exploratory analysis helps us get a better understanding of the dataset at hand. The next task is to build an actual classifier for the dataset. As we know, for any classification problem the first and foremost step is to split the dataset into train and validation sets. Since we are utilizing Keras, we would take help from its utilities to help prepare our datasets. The following code snippet showcases the process of organizing the original dataset into train and validation sets:

```
# Prepare column to store image path
data_labels['image_path'] = data_labels.apply(
                                            lambda row: (train_folder +
                                            row["id"] +  ".jpg" ),
                                            axis=1)
# load image data as arrays of defined size
train_data = np.array([img_to_array(load_img(img, target_size=(299,
                                                    299)))
                          for img in
data_labels['image_path'].values.tolist()
                      ]).astype('float32')

# split data into train and test
x_train, x_test, y_train, y_test = train_test_split(train_data,
                                            target_labels,
                                            test_size=0.3,
stratify=np.array(target_labels),
                                            random_state=42)

# split train dataset into train and validation sets
x_train, x_val, y_train, y_val = train_test_split(x_train,
                                            y_train,
                                            test_size=0.15,
                                            stratify=
                                              np.array(y_train),
                                            random_state=42)
```

The first step, shown precedingly, is to prepare a derived column in the labels `dataframe` to hold actual image paths. Then, we proceed to simply splitting the dataset into train, validation, and test datasets. The next step is to quickly convert our labels into their one-hot encoded form before we feed them to our models. The following code snippet prepares one-hot encoded form of target variables:

```
y_train_ohe = pd.get_dummies(y_train.reset_index(
                                        drop=True)
                                          ).as_matrix()
y_val_ohe = pd.get_dummies(y_val.reset_index(
                                        drop=True)
                                        ).as_matrix()
y_test_ohe = pd.get_dummies(y_test.reset_index(
                                        drop=True)
                                        ).as_matrix()
```

As we know, deep learning algorithms are data hungry. In this case, even though we have a total of 10,000 images, the count per category is not huge. To improve upon this, we perform augmentation. **Data augmentation**, in simple words, is the process of utilizing a curated dataset to enlarge itself, by generating variations of existing data points. In this case, we utilize `ImageDataGenerator` from `keras` to augment our train and validation datasets, as shown in the following snippet:

```
# Create train generator.
train_datagen = ImageDataGenerator(rescale=1./255,
                                   rotation_range=30,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   horizontal_flip = 'true')

train_generator = train_datagen.flow(x_train,
                                     y_train_ohe,
                                     shuffle=False,
                                     batch_size=BATCH_SIZE,
                                     seed=1)

# Prepare Validation data augmentation
val_datagen = ImageDataGenerator(rescale = 1./255)
val_generator = train_datagen.flow(x_val,
                                   y_val_ohe,
                                   shuffle=False,
                                   batch_size=BATCH_SIZE,
                                   seed=1)
```

Now that we have our data ready, the next step is to prepare the actual classifier.

Dog classifier using transfer learning

Now that our datasets are ready, let's get started with the modeling process. We already know how to build a deep convolutional network from scratch. We also understand the amount of fine-tuning required to achieve good performance. For this task, we will be utilizing the concepts of transfer learning.

A pretrained model is the basic ingredient required to begin with the task of transfer learning. As discussed in earlier chapters, transfer learning can be leveraged by either fine-tuning weights of a pretrained network on a current task, or utilizing the pretrained model as a feature extractor.

In this use case, we will concentrate on utilizing a pretrained model as a feature extractor. As we know, a deep learning model is basically a stacking of interconnected layers of neurons, with the final one acting as a classifier. This architecture enables deep neural networks to capture different features at different levels in the network. Thus, we can utilize this property to use them as feature extractors. This is made possible by removing the final layer or using the output from the penultimate layer. This output from the penultimate layer is then fed into an additional set of layers, followed by a classification layer. The following code snippet showcases feature extraction based on the **InceptionV3** pretrained model and stacking additional layers to prepare the classifier:

```
# Get the InceptionV3 model so we can do transfer learning
base_inception = InceptionV3(weights='imagenet',
                             include_top = False,
                             input_shape=(299, 299, 3))


# Add a global spatial average pooling layer
out = base_inception.output
out = GlobalAveragePooling2D()(out)
out = Dense(512, activation='relu')(out)
out = Dense(512, activation='relu')(out)
total_classes = y_train_ohe.shape[1]

predictions = Dense(total_classes,
                    activation='softmax')(out)
```

As shown in the preceding code snippet, **Keras** provides simple utilities to work with many pretrained models, and using them as feature extractors is as simple as setting the flag `include_top` to `False`. In the following code snippet, we prepare the final model by stacking the two sets of layers one on top of the other, and then freezing the layers from InceptionV3:

```
model = Model(inputs=base_inception.input,
              outputs=predictions)

# only if we want to freeze layers
for layer in base_inception.layers:
    layer.trainable = False
```

We now have the model in place, all set to be trained on our Dog Breed Identification dataset. We train the model using the `fit_generator()` method to leverage the data augmentation prepared in the previous step. We set the batch size to 32, and train the model for 13 epochs. The following snippet sets the ball rolling:

```
batch_size = BATCH_SIZE
train_steps_per_epoch = x_train.shape[0] // batch_size
val_steps_per_epoch = x_val.shape[0] // batch_size

history = model.fit_generator(train_generator,
                              steps_per_epoch=train_steps_per_epoch,
                              validation_data=val_generator,
                              validation_steps=val_steps_per_epoch,
                              epochs=15,
                              verbose=1)
```

Since we are saving the output of model parameters and performance after every epoch (the `history` object), we will now utilize it to understand the model performance. The following figure plots the model's train and test accuracies along with its loss performance:
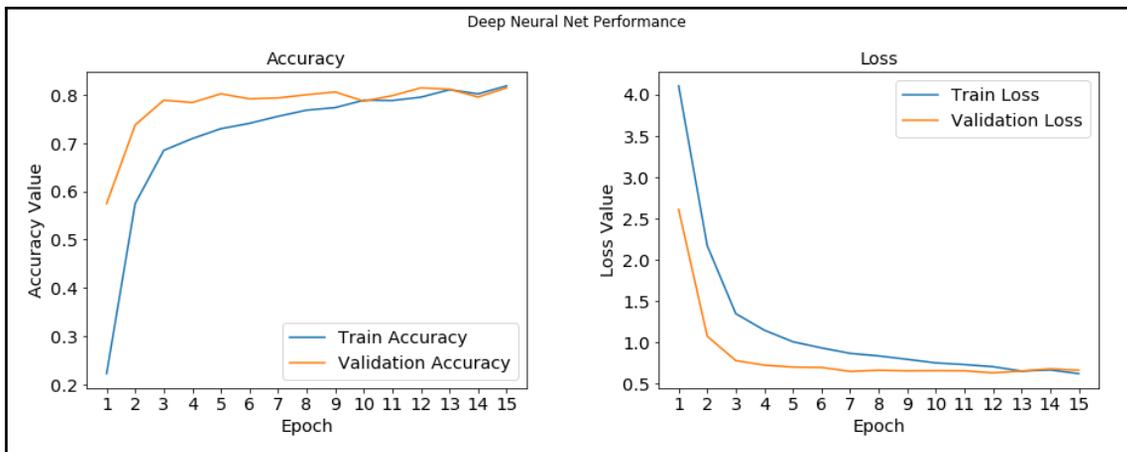
Dog breed classifier performance

The model achieves a commendable performance of more than **80%** accuracy on both train and validation sets within just 15 epochs. The plot on the right-hand side shows how quickly the loss drops and converges to around **0.5**. This is a clear example of how powerful, yet simple, transfer learning can be.

Training and validation performance is pretty good, but how about performance on unseen data? Since we already divided our original dataset into three separate portions. The important thing to remember here is that the test dataset has to undergo similar preprocessing as the training dataset. To account for this, we scale the test dataset as well, before feeding it into the function.

The model achieves an amazing **85% accuracy** as well as a **0.85 F1 score** on the test dataset. Given that we just trained for 15 epochs with minimal inputs from our side, transfer learning helped us achieve a pretty decent classifier:
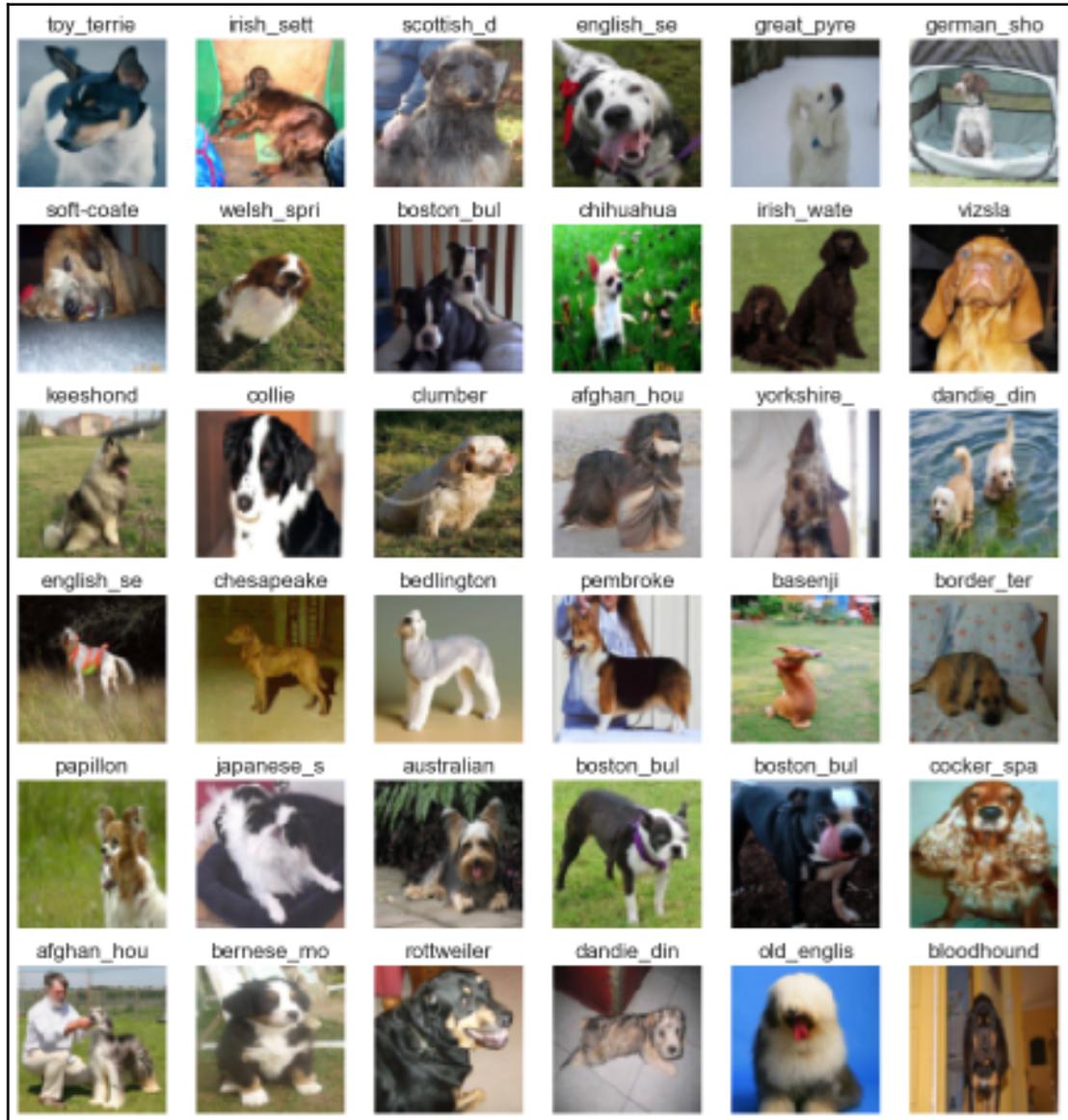
Dog breed classifier predictions

The preceding image presents a visual proof of the model's performance. As we can see, in most of the cases the model is not only predicting the correct dog breed, it also does so with very high confidence.

Summary

We covered a whole lot of theory in the first two parts of the book. Having built a strong foundation of concepts and techniques, we started the use case-driven journey in this chapter. This chapter is the first in a series of upcoming chapters to showcase actual use cases of transfer learning in different scenarios and domains. In this chapter, we applied transfer learning to the domain of visual object identification, or, as it is popularly termed, **image classification**.

We started off with a quick refresher around CNNs and how the whole stage of computer-aided object identification changed once and for all with the arrival of deep learning models in 2012. We briefly touched upon various state-of-the-art image classification models, which have surpassed human performance. We also had a quick look into different benchmarking datasets utilized by academic and industry experts to train and fine-tune their models. Once the context was set, we started off with the CIFAR-10 dataset. We built a classifier from scratch using Keras with TensorFlow as the backend. We leveraged transfer learning to improve upon this performance by using VGG-16 as a pretrained model for feature extraction.

In the final section of the chapter, we leveraged transfer learning to work on a slightly more complex problem. Instead of a dataset where the categories are non-overlapping (CIFAR-10), we prepared a dog breed classifier to identify 120 different dog breeds based on the Stanford Dogs dataset. With a few lines of code, we were able to achieve near state-of-the-art performance. The second use case is also termed as a *fine-grained image classification* task, and is more complex than the usual task of image classification. In this chapter, we showcased both the power and ease of achieving amazing results with transfer learning. Stay tuned for some amazing use cases in the upcoming chapters from domains like computer vision, audio analysis, and so on.

7
Text Document Categorization

In this chapter, we discuss the application of transfer learning to text document categorization. Text categorization is a very popular natural language processing task. The key objective is to assign a document to one or more classes or categories based on its textual content. This has widespread applications in the industry including email classification to spam/non-spam, review and ratings classification, sentiment analysis, email or incident routing where we categorize emails\incidents so that it can be automatically assigned to respective person. The following are the major topics that will be covered in this chapter:

- Text categorization in general, industry applications, and challenges
- Benchmark text categorization datasets and performance of traditional models
- Word representation by dense vectors—deep learning models
- CNN document model—word-to-sentence embedding and then to document embeddings
- Application of transfer learning where source and target domain distributions are different; that is, the source domain consists of classes with less overlap and the target domain has many mixing classes
- Application of transfer learning where source and target domains themselves are different (for example, the source is news and the target is movie reviews, and so on)
- Application of the trained model to do other text analysis tasks such as document summarization—explaining why a review is categorized as negative/positive

We will focus on both conceptual and practical implementations with hands-on examples. The code for this chapter is available for quick reference in the `Chapter 7` folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

Text categorization

Given a set of text documents and a set of predefined categories, the objective of text categorization is to assign each document to a category. The output can be a soft assignment or a hard assignment, depending on the problem. Soft assignment means that the category assignment is defined as a probability distribution over all categories.

There are a wide range of applications of text categorization in industry. The following are a few examples:

- **Spam filtering**: Given an email, classify it as spam or legitimate email.
- **Sentiment classification**: Given a review text (movie review, product review), identify the user polarity—whether its a positive or negative or neural review.
- **Problem ticket assignment**: Typically, in any industry, whenever a user faces an issue regarding any IT application or a software/hardware product, the fist step is to create a problem ticket. These tickets are text documents that describe the problem the user is facing. The next logical step is, someone has to read the description and assign it to the team with the right expertise to solve the issue. Now, given some historical ticket and resolution team categories, it's possible to build a text classifier to automatically classify the problem ticket.
- **Auto resolution of problem tickets**: In some cases the resolution to a problem is also predefined; that is, the expert team knows which steps to follow to solve the issue. So, in such cases, if a text classifier can be built with good accuracy for classifying the tickets, then, once the ticket category is predicted, an automated script can be run to directly resolve the issue. This is one of the goals of future **Artificial Intelligence for IT Operations** (**AIOps**).
- **Targeted marketing**: Marketers can monitor users in social media and classify them as promoters or detractors, and, based on that, what they are saying about the product online.
- **Genre classification**: Automatic text genre classification is very important for classification and retrieval purposes. Even if a set of documents belongs to the same class as the documents share a common topic, they often serve different purposes, falling into diverse genre classes. If the genre of every document in a search database can be detected, information retrieval results could be better presented to the user, depending on the user preference.
- **Fraud detection in claims**: Analyzing insurance claim text documents and detecting whether the claim is fraudulent.

Traditional text categorization

Building text categorization algorithms/models involves a set of preprocessing steps and proper representation of textual data as numerical vectors. Following are the general preprocessing steps:

1. **Sentence splitting**: Split a document into a set of sentences.
2. **Tokenization**: Split sentences into constituent words.
3. **Stemming or lemmatization**: The word tokens are reduced to their base form. For example, words such as playing, played, and plays have one base: *play*. The base word output of stemming need not be a word in the dictionary. Whereas the root word from lemmatization, also known as the **lemma**, will always be present in the dictionary.
4. **Text cleanup**: Case conversion, correcting spellings, and removing stopwords and other unnecessary terms.

Given a corpus of text documents, we can apply the preceding steps and then get a cleaned vocabulary of words constituting the corpus. The next step is text representation. The **bag of words** (**BoW**) model is one of the simplest yet most powerful techniques for extracting features from text documents and creating a vector representation of text. If we have N words in the extracted vocabulary, then any document can be represented as $D= \{w_1, w_2, \ldots, w_N\}$, where w_i represents the frequency of occurrence of the word in the document. This representation of text as a sparse vector is called the BoW model. Here, we don't consider the sequential nature of text data. One way to partially capture the sequential information is to consider word phrases or n-grams along with single word features while building the vocabulary. However, one challenge with that is the size of our representation; that is, our vocabulary size explodes.

The document vector can also be represented as a binary vector, where each $w_i \in \{0, 1\}$ denotes the presence or absence of the word in the document. The most popular representation is a normalized representation of the word frequency called **Term Frequency-Inverse Document Frequency** (**TF-IDF**) representation. Inverse document frequency, denoted by IDF, is computed by dividing the total number of documents in our corpus by the document frequency for each term and then applying logarithmic scaling on the result. The TF-IDF value is the product of *term frequency* and the *inverted document frequency*. It increases proportionally to the number of times a word appears in the document and is scaled down by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Now, we are all set to build a classification model. We need a labeled set of documents or training data. The following are some popular text classification algorithms:

- Multinomial Naive Bayes
- Support Vector Machine
- k-Nearest Neighbor

The **Support Vector Machine** (**SVM**) with linear kernel has generally shown much higher accuracy compared to the others in the benchmark datasets for text classification.

Shortcomings of BoW models

With the word count-based BoW model, we lose additional information such as the semantics, structure, sequence, and context around nearby words in each text document. Words with similar meaning are treated differently in BoW. Other models of text are **latent semantic indexing** (**LSI**), where the documents are represented in a low dimensional (k << vocabulary size)—hidden topic space. In LSI, the constituent words in the document can also be represented as a *k* dimensional dense vector. It's observed that, in LSI models, words with similar semantics have close representations. Also, this dense representation of words is the first step for applying deep learning models to text and is called **word embedding**. Neural network-based language models try to predict words from their neighboring words by looking at word sequences in the corpus, and in the process learn distributed representations, giving us dense word embedding.

Benchmark datasets

Following is a list of benchmark datasets used in the majority of text categorization research:

- **IMDB Movie Review dataset**: This is a dataset for binary sentiment classification. It contains a set of 25,000 movie reviews for training, and 25,000 for testing. There is additional unlabeled data for use as well. This dataset is downloadable from `http://ai.stanford.edu/~amaas/data/sentiment/`.

- **Reuters dataset**: This dataset has 90 classes, 9,584 training documents and 3,744 testing documents. It's available as a part of the package `nltk.corpus`. The class distribution for the documents in this dataset is very skewed, with the two most frequent classes containing approximately 70% of all the documents. Even if we consider only the 10 most frequent classes, the two most frequent classes in this dataset have approximately 80% of all the documents. So most of the classification results are evaluated on subsets of these top frequent classes, and are named R8, R10, and R52 for the top 8, 10, and 52 frequent classes in the train set.
- **20 Newsgroup dataset**: This data is organized into 20 different newsgroups, each corresponding to a different topic. Some of the newsgroups are very closely related to each other (for example: `comp.sys.ibm.pc.hardware/comp.sys.mac.hardware`), while others are highly unrelated (for example: `misc.forsale/soc.religion.christian`). Here is a list of the 20 newsgroups, partitioned into six major categories according to subject matter. This dataset is available in `sklearn.datasets`:

`comp.graphics`
`comp.os.ms-windows.misc`
`comp.sys.ibm.pc.hardware`
`comp.sys.mac.hardware`
`comp.windows.x`
	`rec.autos`
	`rec.motorcycles`
	`rec.sport.baseball`
	`rec.sport.hockey`
		`sci.crypt`
		`sci.electronics`
		`sci.med`
		`sci.space`

`misc.forsale`	`talk.politics.misc`
	`talk.politics.guns`
	`talk.politics.mideast`
		`talk.religion.misc`
		`alt.atheism`
		`soc.religion.christian`

We will discuss later how to load up this dataset for further analysis.

Word representations

Let's look at some of these advanced strategies for handling text data and extracting meaningful features or word embeddings from it, which can be used in other **machine learning** (**ML**) systems for more advanced tasks such as classification, summarization, and translation. We can transfer the learned representation of words to another model. If we have huge training data then word embeddings can be jointly learned along with the final task.

Word2vec model

This model was created by Google in 2013 and is a predictive, deep learning-based model that computes and generates high quality, distributed, and continuous dense vector representations of words, which capture contextual and semantic similarity. Essentially, these are unsupervised models that can take in massive textual corpora, create a vocabulary of possible words, and generate dense word embeddings for each word in the vector space representing that vocabulary. Usually, you can specify the size of the word embedding vectors, and the total number of vectors is essentially the size of the vocabulary. This makes the dimensionality of this dense vector space much lower than the high-dimensional sparse vector space built using traditional BoW models.

There are two different model architectures that can be leveraged by Word2vec to create these word embedding representations. These are:
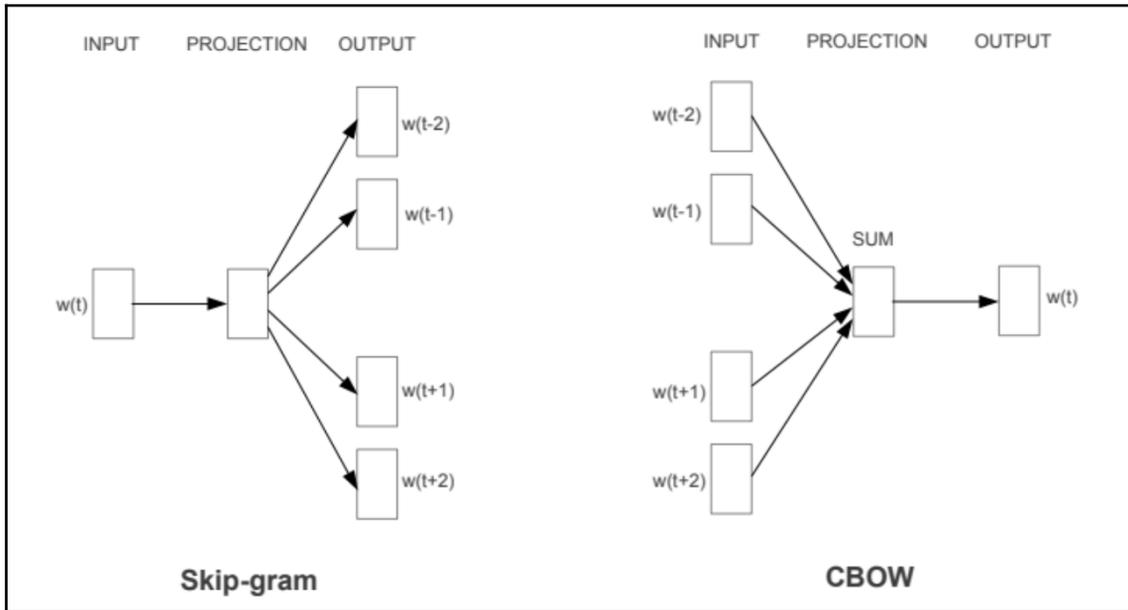
- The **Continuous Bag of Words** (**CBOW**) model
- The **Skip-gram** model

The CBOW model architecture tries to predict the current target word (the center word) based on the source context words (surrounding words). Considering a simple sentence, *the quick brown fox jumps over the lazy dog*, this can be pairs of (`context_window`, `target_word`), where if we consider a context window of size 2, we have examples like *([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy)*, and so on. Thus, the model tries to predict the target word based on context window words. The Word2vec family of models are unsupervised; this means that you can just give it a corpus without additional labels or information and it can construct dense word embeddings from the corpus. But you will still need to leverage a supervised classification methodology once you have this corpus to get to the embeddings. But we will do that from within the corpus itself, without any auxiliary information. We can model this CBOW architecture as a deep learning classification model, such that we take in the context words as our input, *X*, and try to predict the target word, *Y*. In fact, building this architecture is simpler than the skip-gram model, where we try to predict a whole bunch of context words from a source target word.

The skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the center word). Consider our simple sentence from earlier, *the quick brown fox jumps over the lazy dog*. If we used the CBOW model, we get pairs of (`context_window`, `target_word`), where if we consider a context window of size 2, we have examples like *([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy)*, and so on. Now, considering that the skip-gram model's aim is to predict the context from the target word, the model typically inverts the contexts and targets, and tries to predict each context word from its target word.

Hence, the task becomes to predict the context [*quick, fox*] given the target word *brown* or [*the, brown*] given the target word *quick*, and so on. Thus, the model tries to predict the `context_window` words based on the `target_word`.

Following is the architecture diagram of the preceding two models:

An implementation of these models in Keras can be found in the following blog post by one of

us: `https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa`.

Word2vec using gensim

The gensim framework, created by Radim Rehurek, consists of a robust, efficient, and scalable implementation of the Word2vec model (`https://radimrehurek.com/gensim/models/word2vec.html`). It allows us to chose either one of the skip-gram or CBOW models. Let's try to learn and visualize the word embedding for the IMDB corpora. As discussed before, this has 50,000 labeled documents and 50,000 unlabeled documents. For learning word representations, we don't need any labels and hence we can use all of the available 100,000 documents.

Let's first load the full corpora. The downloaded documents are divided into `train`, `test`, and `unsup` folders:

```python
def load_imdb_data(directory = 'train', datafile = None):
        '''
        Parse IMDB review data sets from Dataset from
        http://ai.stanford.edu/~amaas/data/sentiment/
        and save to csv.
        '''
        labels = {'pos': 1, 'neg': 0}
        df = pd.DataFrame()
        for sentiment in ('pos', 'neg'):
            path =r'{}/{}/{}'.format(config.IMDB_DATA, directory,
                                    sentiment)
            for review_file in os.listdir(path):
                with open(os.path.join(path, review_file), 'r',
                        encoding= 'utf-8') as input_file:
                    review = input_file.read()
                df = df.append([[utils.strip_html_tags(review),
                                labels[sentiment]]],
                                ignore_index=True)
        df.columns = ['review', 'sentiment']
        indices = df.index.tolist()
        np.random.shuffle(indices)
        indices = np.array(indices)
        df = df.reindex(index=indices)
        if datafile is not None:
            df.to_csv(os.path.join(config.IMDB_DATA_CSV, datafile),
                    index=False)
        return df
```

We can combine all the three data sources and get a list of 100,000 documents as follows:

```python
corpus = unsupervised['review'].tolist() + train_df['review'].tolist()
        + test_df['review'].tolist()
```

We can preprocess this corpus and convert each document to a sequence of word tokens. We use `nltk` for this. Then, we can start the training as follows. We are using a huge number of iterations and thus will require 6-8 hours of time to train on a CPU:

```python
# tokenize sentences in corpus
wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document.lower()) for document in corpus]

w2v_model = word2vec.Word2Vec(tokenized_corpus, size=50,
                                window=10, min_count=5,
                                sample=1e-3, iter=1000)
```

Let's now see what this model has learned. Let's choose some opinion words from this corpus. A good list of opinion words typically used in movie reviews can be found here: `http://member.tokoha-u.ac.jp/~dixonfdm/Writing%20Topics%20htm/ Movie%20Review%20Folder/movie_descrip_vocab.htm`. We will first find the top five words with similar embeddings as these given words. The following is the code for this:

```
similar_words = {search_term: [item[0] for item in
w2v_model.wv.most_similar([search_term], topn=5)]
                for search_term in ['good','superior','violent',
                                    'romantic','nasty','unfortunate',
                                    'predictable', 'hilarious',
                                    'fascinating', 'boring','confused',
                                    'sensitive',
                                    'imaginative','senseless',
                                    'bland','disappointing']}

pd.DataFrame(similar_words).transpose()
```

The output for the preceding code is as follows:

Out[12]:

	0	1	2	3	4
bland	dull	lifeless	forgettable	uninspired	unconvincing
boring	dull	pointless	tedious	predictable	uninteresting
confused	irritated	puzzled	disturbed	frustrated	annoyed
disappointing	unsatisfying	disappointed	enjoyable	surprising	satisfying
fascinating	compelling	enthralling	captivating	unique	vivid
good	decent	great	nice	bad	fine
hilarious	funny	hysterical	priceless	comical	humorous
imaginative	inventive	innovative	ingenious	intricate	creative
nasty	sadistic	sleazy	gory	icky	vicious
predictable	clichéd	formulaic	contrived	implausible	dull
romantic	romance	screwball	bittersweet	sentimental	delightful
senseless	pointless	meaningless	disgusting	sickening	boring
sensitive	sincere	passionate	mature	delicate	confident
superior	inferior	weaker	truer	classier	maligned
unfortunate	unacceptable	disastrous	dubious	inadequate	important
violent	brutal	graphic	gruesome	sadistic	violence

We can see that the learned embeddings represent the words used in a similar context with similar embedding vectors. These words need not be synonyms all the time, they can be opposite as well. But, they are used in a similar context.

GloVe model

The GloVe model stands for Global Vectors, which is an unsupervised learning model that can be used to obtain dense word vectors similar to Word2Vec. However, the technique is different and training is performed on an aggregated global word-word co-occurrence matrix, giving us a vector space with meaningful substructures. This method was published in the paper, *GloVe: Global Vectors for Word Representation* by Pennington and their co-authors (`https://www.aclweb.org/anthology/D14-1162`). We have talked about count-based matrix factorization methods, such as **latent semantic analysis** (**LSA**) and predictive methods like Word2vec. This paper claims that currently both families suffer significant drawbacks. Methods like LSA efficiently leverage statistical information, but they do relatively poorly on the word analogy task—how we found out semantically similar words. Methods like skip-gram may do better on the analogy task, but they poorly utilize the statistics of the corpus on a global level.

The basic methodology of the GloVe model is to first create a huge word-context co-occurrence matrix consisting of (word, context) pairs, such that each element in this matrix represents how often a word occurs within the context (which can be a sequence of words). This word-context matrix *WC* is very similar to the term-document matrix popularly used in text analysis for various tasks. Matrix factorization is used to represent the matrix *WC* as a product of two matrices; the **Word-Feature** (**WF**) matrix and the **Feature-Context** (**FC**) matrix. *WC = WF x FC*. *WF* and *FC* are initialized with some random weights, and we multiply them to get *WC'* (an approximation of *WC*) and measure how close it is to *WC*. We do this multiple times, using **Stochastic Gradient Descent** (**SGD**) to minimize the error. Finally, the *WF* matrix gives us the word embeddings for each word, where *F* can be preset to a specific number of dimensions. A very important point to remember is that both the Word2vec and GloVe models are very similar in how they work. Both of them aim to build a vector space where the position of each word is influenced by its neighboring words, based on their context and semantics. Word2vec starts with local individual examples of word co-occurrence pairs, and GloVe starts with global aggregated co-occurrence statistics across all words in the corpus.

In the following sections, we will be using both Word2vec and GloVe for various classification problems. We have developed some utility code to read and load GloVe and Word2vec vectors from a file and return an embedding matrix. The file format expected is standard GloVe file format. Following is an example in five-dimensional embedding format for a few words—word followed by vectors, all space separated:

- Flick 7.068106 -5.410074 1.430083 -4.482612 -1.079401
- Heart -1.584336 4.421625 -12.552878 4.940779 -5.281123
- Side 0.461367 4.773087 -0.176744 8.251079 -11.168787
- Horrible 7.324110 -9.026680 -0.616853 -4.993752 -4.057131

Following is the main function to read the GloVe vectors, given a vocabulary as a Python dictionary with the dictionary keys as words in vocabulary. This is needed to load only the required embedding for the words that occur in our training vocabulary. Also, the words in the vocabulary that are not present in GloVe embeddings are initialized with a mean vector of all embeddings and some white noise. The rows 0 and 1 are dedicated for the space and **out-of-vocabulary** (**OOV**) words. These are words that are not in the vocabulary but are there in the corpus, such as very infrequent words or some filtered out noise. The embedding for space is a zero vector. The embedding for OOV is the mean vector of all the remaining embeddings:

```python
def _init_embedding_matrix(self, word_index_dict,
                           oov_words_file='OOV-Words.txt'):
    # reserve 0, 1 index for empty and OOV
    self.embedding_matrix = np.zeros((len(word_index_dict)+2 ,
                                      self.EMBEDDING_DIM))
    not_found_words=0
    missing_word_index = []
    with open(oov_words_file, 'w') as f:
        for word, i in word_index_dict.items():
            embedding_vector = self.embeddings_index.get(word)
            if embedding_vector is not None:
                # words not found in embedding index will be all-
                  zeros.
                self.embedding_matrix[i] = embedding_vector
            else:
                not_found_words+=1
                f.write(word + ','+str(i)+'\n')
                missing_word_index.append(i)

        #oov by average vector:
        self.embedding_matrix[1] = np.mean(self.embedding_matrix,
                                           axis=0)
```

```
            for indx in missing_word_index:
                self.embedding_matrix[indx] =
                                    np.random.rand(self.EMBEDDING_DIM)+
                                            self.embedding_matrix[1]
        print("words not found in embeddings:
            {}".format(not_found_words))
```

One more utility function is `update_embeddings`. This is required for transfer learning. We may want to update the embeddings learned by one model with the embeddings learned by another model:

```
def update_embeddings(self, word_index_dict, other_embedding,
other_word_index):
        num_updated = 0
        for word, i in other_word_index.items():
            if word_index_dict.get(word) is not None:
                embedding_vector = other_embedding[i]
                this_vocab_word_indx = word_index_dict.get(word)
                self.embedding_matrix[this_vocab_word_indx] =
                                                embedding_vector
                num_updated+=1
        print('{} words are updated out of {}'.format(num_updated,
            len(word_index_dict)))
```

CNN document model

We previously saw how word embeddings are capable of capturing many semantic relationships between the concepts they represent. We will now introduce a ConvNet document model that builds hierarchical distributed representations of documents. This was published in the paper `https://arxiv.org/pdf/1406.3830.pdf` by Misha Denil et al. The model is divided into two levels, a sentence level and a document level, both of which are implemented using ConvNets. At the sentence level, a ConvNet is used to transform embeddings for the words in each sentence into an embedding for the entire sentence. At the document level, another ConvNet is used to transform sentence embeddings to a document embedding.

In any ConvNet architecture a convolution layer is followed by a sub-sampling/pooling layer. Here, we use k-max pooling. A k-max pooling operation is slightly different from normal max pooling, which takes the max from a sliding window of neurons. In a k-max pooling operation, the maximum k neurons are taken from all the neurons in the layer below. For example, applying 2-max pooling to [3, 1, 5, 2] yields [3, 5]. Here, normal max pooling with kernel size 3 and stride 1 will give the same result. Let's take another case. If we apply max pooling on [1, 2, 3, 4, 5] we will get [3, 5], but 2-max pool will give [4, 5]. K-max pooling can be applied on variable size inputs and we can still get the same number of output units.

The following diagram depicts the **Convolutional Neural Network** (**CNN**) architecture. We have fine-tuned this architecture a bit for various use cases that will be discussed here:
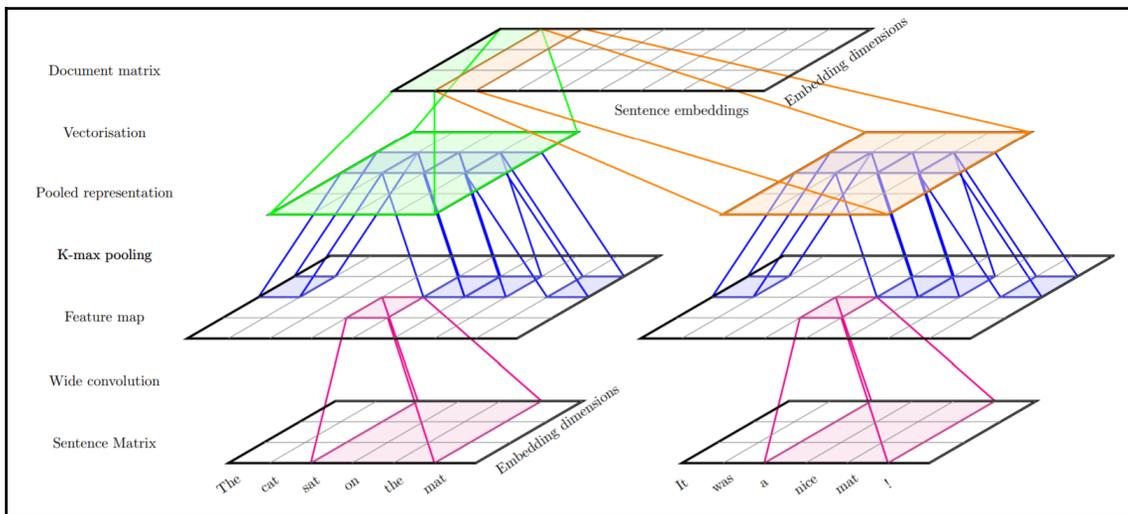
The input layer to this network is not shown here. The input layer is a sequence of sentences in the document, in order, where each sentence is represented by a sequence of word indices. The following code snippet describes how the word indices are defined, given a training corpus. The indices 0 and 1 are reserved for empty and OOV words. First, the documents in the corpus are tokenized to words. The non-English words are filtered out. Also, the frequency of each word in the entire corpus is computed. For a large corpus we can filter out infrequent words from vocabulary. Then, an integer index is assigned to each word in the vocabulary:

```
from nltk.tokenize import sent_tokenize, wordpunct_tokenize
import re

corpus = ['The cat sat on the mat . It was a nice mat !',
```

```
                    'The rat sat on the mat . The mat was damaged found at 2 places.']

vocab ={}
word_index = {}
for doc in corpus:
    for sentence in sent_tokenize(doc):
        tokens = wordpunct_tokenize(sentence)
        tokens = [token.lower().strip() for token in tokens]
        tokens = [token for token in tokens
                        if re.match('^[a-z,.;!?]+$',token) is not None ]
        for token in tokens:
            vocab[token] = vocab.get(token, 0)+1
# i= 0 for empty, 1 for OOV
i = 2
for word, count in vocab.items():
    word_index[word] = i
    i +=1
print(word_index.items())

#Here is the output:
dict_items([('the', 2), ('cat', 3), ('sat', 4), ('on', 5), ('mat', 6),
('.', 7), ('it', 8), ('was', #9), ('a', 10), ('nice', 11), ('!', 12),
('rat', 13), ('damaged', 14), ('found', 15), ('at', 16), ('places', 17)])
```

Now, the corpus can be converted to an array of word indices. In the corpus, different sentences and documents have different lengths. Although convolutions can handle inputs of arbitrary width, for simplicity of implementation we can define a fixed size input to the network. We can zero pad short sentences and truncate longer sentences to fit the fixed sentence length and do the same at document level. In the following code snippet, we show how to use the keras.preprocessing module to zero pad the sentences and document and prepare the data:

```
from keras.preprocessing.sequence import pad_sequences

SENTENCE_LEN = 10; NUM_SENTENCES=3;
for doc in corpus:
    doc2wordseq = []
    sent_num =0
    for sentence in sent_tokenize(doc):
        words = wordpunct_tokenize(sentence)
        words = [token.lower().strip() for token in words]
        word_id_seq = [word_index[word] if word_index.get(word) is not
         None \
                                        else 1 for word in words]
        padded_word_id_seq = pad_sequences([word_id_seq],
                                            maxlen=SENTENCE_LEN,
                                            padding='post',
```

```
                                       truncating='post')

        if sent_num < NUM_SENTENCES:
            doc2wordseq = doc2wordseq + list(padded_word_id_seq[0])
    doc2wordseq = pad_sequences([doc2wordseq],
                                maxlen=SENTENCE_LEN*NUM_SENTENCES,
                                padding='post',
                                truncating='post')
    print(doc2wordseq)

# sample output
[ 2  3  4  5  2  6  7  0  0  0  8  9 10 11  6 12  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 2 13  4  5  2  6  7  0  0  0  2  6  9 14 15 16  1 17  7  0  0  0  0  0  0  0  0  0  0  0]
```

So, you can see that each document input is a one-dimensional tensor of size *doc_length* = SENTENCE_LEN * NUM_SENTENCES. These tensors are passed through the first layer, the embedding layer in the network, to convert word indices to a dense word representation, and we get a two-dimensional tensor of shape *doc_length* × *embedding_dimension*. All preceding preprocessing code is bundled in the class Preprocess and has fit and transform methods, like scikit modules. The fit method takes the train corpus as input, builds the vocabulary, and assigns word indices to each word in the vocabulary. Then, the transform method can be used to convert the test or hold out set to padded word index sequences as shown precedingly. The transform method will use the word indices calculated by fit.

The embedding matrix can be initialized with GloVe or Word2vec. Here, we have used 50 dimensional GloVe embedding to initialize the embedding matrix. The words not found in GloVe and OOV words are initialized as follows:

- OOV words—words excluded in training data vocabulary (index 1) are initialized by a mean of all GloVe vectors
- The words not found in GloVe are initialized by a mean of all Glove vectors and a random vector of the same dimension

The following code snippet does the same in the _init_embedding_matrix method from the GloVe class discussed before:

```
#oov by average vector:
self.embedding_matrix[1] = np.mean(self.embedding_matrix, axis=0)
for indx in missing_word_index:
    self.embedding_matrix[indx] = np.random.rand(self.EMBEDDING_DIM)+
                                  self.embedding_matrix[1]
```

Having initialized the embedding matrix, we are now ready to build the first layer, the embedding layer, as follows:

```
from keras.layers import Embedding
embedding_layer = Embedding(vocab_size,
                            embedding_dim,
                            weights=[embedding_weights],
                            input_length=max_seq_length,
                            trainable=True,
                            name='embedding')
```

Next, we have to build the word convolution layer. We want the same one-dimensional convolution filters to be applied across all sentences—that is, the same convolution filter weights to be shared across all sentences. First, we use a `Lambda` layer to split the input into sentences. Then, if we use *C* convolution filters, each sentence two-dimensional-tensor of shape (*SENTENCE_LEN× EMBEDDING _DIM*) will be converted to the ((*SENTENCE_LEN-filter+1) × C)* tensor. The following code does the same:

```
#Let's take sentence_len=30, embedding_dim=50, num_sentences = 10
#following convolution filters to be used for all sentences.
word_conv_model = Conv1D(filters= 6,
                         kernel_size= 5,
                         padding="valid",
                         activation="relu",
                         trainable = True,
                         name = "word_conv",
                         strides=1)

for sent in range(num_sentences):
    ##get one sentence from the input document
    sentence = Lambda(lambda x : x[:, sent*sentence_len:
                                      (sent+1)*sentence_len, :])(z)
    ##sentence shape : (None, 30, 50)
    conv = word_conv_model(sentence)
    ## convolution shape : (None, 26, 6)
```

The k-max pooling layer is not available in `keras`. We can implement k-max pool as a custom layer. To implement a custom layer, we need to implement three methods:

- `call(x)`: This is where the layer's logic is implemented
- `compute_output_shape(input_shape)`: In case the custom layer modifies the shape of its input
- `build(input_shape)`: Define layer weights (we don't need this as our layer has no weights)

Here, is the full code of the k-max pooling layer:

```
import tensorflow as tf
from keras.layers import Layer, InputSpec

class KMaxPooling(Layer):
    def __init__(self, k=1, **kwargs):
        super().__init__(**kwargs)
        self.input_spec = InputSpec(ndim=3)
        self.k = k

    def compute_output_shape(self, input_shape):
        return (input_shape[0], (input_shape[2] * self.k))

    def call(self, inputs):
        # swap last two dimensions since top_k will be
        # applied along the last dimension
        shifted_input = tf.transpose(inputs, [0, 2, 1])
        # extract top_k, returns two tensors [values, indices]
        top_k = tf.nn.top_k(shifted_input, k=self.k, sorted=True,
                            name=None)[0]
        # return flattened output
        return top_k
```

Applying the preceding k-max pooling layer on the word convolutions, we get the sentence embedding layer:

```
for sent in range(num_sentences):
    ##get one sentence from the input document
    sentence = Lambda(lambda x : x[:,sent*sentence_len:
                                    (sent+1)*sentence_len, :])(z)
    ##sentence shape : (None, 30, 50)
    conv = word_conv_model(sentence)
    ## convolution shape : (None, 26, 6)
    conv = KMaxPooling(k=3)(conv)
    #transpose pooled values per sentence
    conv = Reshape([word_filters*sent_k_maxpool,1])(conv)
    ## shape post k-max pooling and reshape (None, 18=6*3, 1)
```

So, we convert each sentence of shape 30 ×50 to 18 × 1 and then we concatenate these tensors to get the sentence embedding. We use the `Concatenate` layer in Keras to implement the same:

```
z = Concatenate()(conv_blocks) if len(conv_blocks) > 1 else conv_blocks[0]
z = Permute([2,1], name='sentence_embeddings')(z)
## output shape of sentence embedding is : (None, 10, 18)
```

As before, one-dimensional convolution followed by k-max pooling is applied on the preceding sentence embedding to get the *document* embedding. This completes the document model for text. Based on the learning task in hand, the next layers can be defined. For a classification task, the document embedding can be connected to a dense layer, followed by a final softmax layer with *K* units for k-class classification problems. We can have more than one dense layer before the final layer. The following code snippet implements the same:

```
sent_conv = Conv1D(filters=16,
                   kernel_size=3,
                   padding="valid",
                   activation="relu",
                   trainable = True,
                   name = 'sentence_conv',
                   strides=1)(z)

z = KMaxPooling(k=5)(sent_conv)
z = Flatten(name='document_embedding')(z)
for i in range(num_hidden_layers):
    layer_name = 'hidden_{}'.format(i)
    z = Dense(hidden_dims, activation=hidden_activation,
              name=layer_name)(z)
model_output = Dense(K, activation='sigmoid',name='final')(z)
```

The whole code is included in the `cnn_document_model` module.

Building a review sentiment classifier

Let's now build a sentiment classifier by training the preceding CNN document model. We will be using the Amazon Reviews for Sentiment Analysis dataset from `https://www.kaggle.com/bittlingmayer/amazonreviews` to train this model. This dataset consists of a few million Amazon customer reviews (input text) and star ratings (output labels). Here is the the data format: label followed by space, review titles followed by : and a space, are prepended to the review text. This dataset is much bigger than the popular IMDB Movie Review dataset. Also, this dataset contains quite a diverse set of reviews of various products and also movies:

```
__label__<X> <summary/title>: <Review Text>

Example:
__label__2 Good Movie: Awesome.... simply awesome. I couldn't put this down
and laughed, smiled, and even got tears! A brand new favorite author.
```

Here, `__label__1` corresponds to 1 and 2-star reviews, and `__label__2` corresponds to 4 and 5-star reviews. However, 3-star reviews, that is, reviews with a neutral sentiment, were not included in this dataset. We have 3.6 million training examples and 400,000 test examples in total in this dataset. We will take a random sample of size 200,000 from training examples to start with so that we can guess a good hyperparameter for the training to proceed:

```
train_df = Loader.load_amazon_reviews('train')
print(train_df.shape)

test_df = Loader.load_amazon_reviews('test')
print(test_df.shape)

dataset = train_df.sample(n=200000, random_state=42)
dataset.sentiment.value_counts()
```

Next, we use the `Preprocess` class to convert the corpus to padded word index sequences as follows:

```
preprocessor = Preprocess()
corpus_to_seq = preprocessor.fit(corpus=corpus)

holdout_corpus = test_df['review'].values
holdout_target = test_df['sentiment'].values
holdout_corpus_to_seq = preprocessor.transform(holdout_corpus)
```

Let's initialize the embedding with GloVe using the `GloVe` class, and build the document model. We also need to define the document model parameters, such as number of convolution filters, activation functions, hidden units, and so on. To avoid overfitting of the network, we can interleave dropout layers in between the input layers, convolution layers, and even the final layer or the dense layer. Also, as we observed for the dense layer, putting a Gaussian noise layer acts as a better regularizer. The `DocumentModel` class can be initialized with all these parameters defined as follows. To come up with a good initialization for the model parameters, we start with a small number of epochs and small sampled training examples. Initially, we started with six word convolution filters, as mentioned in the paper for IMDB data, and found that the model was underfitting—the accuracy of training was not going beyond 80%, and then we kept on slowly increasing the number of word filters. Similarly, we found a good number of sentence convolution filters. We tried both ReLU and tanh activations for the convolution layers. As mentioned in the paper `https://arxiv.org/pdf/1406.3830.pdf`, they used tanh activations for their model:

```
    glove=GloVe(50)
    initial_embeddings = glove.get_embedding(preprocessor.word_index)

    amazon_review_model =
```

```
DocumentModel(vocab_size=preprocessor.get_vocab_size(),
                        word_index = preprocessor.word_index,
                        num_sentences = Preprocess.NUM_SENTENCES,
                        embedding_weights = initial_embeddings,
                        conv_activation = 'tanh',
                        hidden_dims=64,
                        input_dropout=0.40,
                        hidden_gaussian_noise_sd=0.5)
```

Following is the full list of parameters for this model, which we have used for training on the full 3.6 million training examples:

```
{
    "embedding_dim":50,
    "train_embedding":true,
    "sentence_len":30,
    "num_sentences":10,
    "word_kernel_size":5,
    "word_filters":30,
    "sent_kernel_size":5,
    "sent_filters":16,
    "sent_k_maxpool":3,
    "input_dropout":0.4,
    "doc_k_maxpool":4,
    "sent_dropout":0,
    "hidden_dims":64,
    "conv_activation":"relu",
    "hidden_activation":"relu",
    "hidden_dropout":0,
    "num_hidden_layers":1,
    "hidden_gaussian_noise_sd":0.5,
    "final_layer_kernel_regularizer":0.0,
    "learn_word_conv":true,
    "learn_sent_conv":true
}
```

Finally, before we start full training, we need to figure out a good batch size. With large batch sizes like 256 the training was very slow, and hence we used a batch size of 64. We used the `rmsprop` optimizer to train our model and started with the default learning rate that `keras` uses. Here is the complete list of training parameters, which are stored in the class `TrainingParameters`:

```
{"seed":55,
    "batch_size":64,
    "num_epochs":35,
    "validation_split":0.05,
    "optimizer":"rmsprop",
    "learning_rate":0.001}
```

The following is the code to start the training:

```
train_params = TrainingParameters('model_with_tanh_activation')

amazon_review_model.get_classification_model().compile(
                                            loss="binary_crossentropy",
                                            optimizer=
                                              train_params.optimizer,
                                            metrics=["accuracy"])
checkpointer = ModelCheckpoint(filepath=train_params.model_file_path,
                               verbose=1,
                               save_best_only=True,
                               save_weights_only=True)

x_train = np.array(corpus_to_seq)
y_train = np.array(target)

x_test = np.array(holdout_corpus_to_seq)
y_test = np.array(holdout_target)

amazon_review_model.get_classification_model().fit(x_train, y_train,
                       batch_size=train_params.batch_size,
                       epochs=train_params.num_epochs,
                       verbose=2,
                       validation_split=train_params.validation_split,
                       callbacks=[checkpointer])
```

We have trained this model on CPU, and following is the result after five epochs. With 190k samples only one epoch is quite slow and takes about 10 minutes to run. However, you can see in the following that the training and validation accuracy after five epochs reaches **92%**, which is quite good:

```
Train on 190000 samples, validate on 10000 samples
 Epoch 1/35
 - 577s - loss: 0.3891 - acc: 0.8171 - val_loss: 0.2533 - val_acc: 0.8369
 Epoch 2/35
 - 614s - loss: 0.2618 - acc: 0.8928 - val_loss: 0.2198 - val_acc: 0.9137
 Epoch 3/35
 - 581s - loss: 0.2332 - acc: 0.9067 - val_loss: 0.2105 - val_acc: 0.9191
 Epoch 4/35
 - 640s - loss: 0.2197 - acc: 0.9128 - val_loss: 0.1998 - val_acc: 0.9206
 Epoch 5/35
 ...
 ...
```

We evaluated this model on the hold out set of 400k reviews, and we get **92%** accuracy there as well. This clearly shows that the model is fitting well for this review data, and with more data there is some more scope for improvement. In this whole training process till now, the major use of transfer learning is the GloVe embedding vectors being used for initializing the word embedding. Here, as we have huge data, we could have learned the weights from scratch. However, let's see which word embeddings are updated the most in the whole training process.

What has embedding changed most?

We can take the initial GloVe embedding and the final learned embedding and compare them by taking the norm of their difference for each word. Then, we can sort the norm values to see which words changed the most. Here is the code to do this:

```
learned_embeddings = amazon_review_model.get_classification_model()
                         .get_layer('embedding').get_weights()[0]

embd_change = {}
for word, i in preprocessor.word_index.items():
    embd_change[word] = np.linalg.norm(initial_embeddings[i]-
                                            learned_embeddings[i])
embd_change = sorted(embd_change.items(), key=lambda x: x[1],
                    reverse=True)
embd_change[0:20]
```

You can check that the most updated embedding are for the opinion words.

Transfer learning – application to the IMDB dataset

One of the scenarios where we should use transfer learning is when we have much less labeled data for the task in hand but lot of training data for a similar but a different domain. The IMDB dataset (`http://ai.stanford.edu/~amaas/data/sentiment/`) is a binary sentiment classification dataset. It has a set of 25,000 movie reviews for training and 25,000 for testing. There are a lot of published papers on this dataset, and probably the best result on this dataset is achieved by the paragraph vector of Le and Mikolov (`https://arxiv.org/pdf/1405.4053.pdf`) from Google. They achieved **92.58%** accuracy on this dataset. SVM achieved 89% on this. This dataset is of decent size, and we can train the CNN model from scratch on this data. This gives us a result on a par with an SVM. It's discussed in next section.

Now, let's try to build a model with a small sample of IMDB data—say, **5%** of the data. In many practical scenarios, we face the problem of insufficient training data. We cannot train a CNN with this small dataset. So, we will use transfer learning to build a model for this dataset.

We follow the same steps first to preprocess and prepare the data as we did for the other dataset:

```
train_df = Loader.load_imdb_data(directory = 'train')
train_df = train_df.sample(frac=0.05, random_state = train_params.seed)
#take only 5%
print(train_df.shape)

test_df = Loader.load_imdb_data(directory = 'test')
print(test_df.shape)

corpus = train_df['review'].tolist()
target = train_df['sentiment'].tolist()
corpus, target = remove_empty_docs(corpus, target)
print(len(corpus))

preprocessor = Preprocess(corpus=corpus)
corpus_to_seq = preprocessor.fit()

test_corpus = test_df['review'].tolist()
test_target = test_df['sentiment'].tolist()
test_corpus, test_target = remove_empty_docs(test_corpus, test_target)
```

```
print(len(test_corpus))

test_corpus_to_seq = preprocessor.transform(test_corpus)

x_train = np.array(corpus_to_seq)
x_test = np.array(test_corpus_to_seq)

y_train = np.array(target)
y_test = np.array(test_target)

print(x_train.shape, y_train.shape)

glove=GloVe(50)
initial_embeddings = glove.get_embedding(preprocessor.word_index)

#IMDB MODEL
```

Now, let's load the trained model first. We have two methods to load: the hyperparameters of the model and the learned model weights in the `DocumentModel` class:

```
def load_model(file_name):
        with open(file_name, "r", encoding= "utf-8") as hp_file:
            model_params = json.load(hp_file)
            doc_model = DocumentModel( **model_params)
            print(model_params)
        return doc_model
def load_model_weights(self, model_weights_filename):
     self._model.load_weights(model_weights_filename, by_name=True)
```

Then, we use the preceding methods to load the pretrained model, and then transfer the learned weights to the new model as follows. The embedding matrix for the pretrained model is bigger and has more words than the corpus. So, we cannot directly use the embedding matrix from the pretrained model. We will use the `update_embedding` method from the `GloVe` class to update the GloVe initialized embedding for the IMDB model with the embedding from the trained model:

```
amazon_review_model = DocumentModel.load_model("model_file.json")
amazon_review_model.load_model_weights("model_weights.hdf5")
learned_embeddings = amazon_review_model.get_classification_model()\
                     .get_layer('embedding').get_weights()[0]

#update the GloVe embeddings.
glove.update_embeddings(preprocessor.word_index,
                        np.array(learned_embeddings),
                        amazon_review_model.word_index)
```

Now, we are all set to build the transfer learning model. Let's build the IMDB model first and initialize the weights from the other pretrained model. We will not train the lower layers of this network with this small amount of data. So, we will set `trainable=False` for them. We will be training only the final layers with large dropouts:

```
initial_embeddings = glove.get_embedding(preprocessor.word_index)#get
                                             updated embeddings

imdb_model = DocumentModel(vocab_size=preprocessor.get_vocab_size(),
                           word_index = preprocessor.word_index,
                           num_sentences=Preprocess.NUM_SENTENCES,
                           embedding_weights=initial_embeddings,
                           conv_activation = 'tanh',
                           train_embedding = False,
                           learn_word_conv = False,
                           learn_sent_conv = False,
                           hidden_dims=64,
                           input_dropout=0.0,
                                   hidden_layer_kernel_regularizer=0.001,
                                   final_layer_kernel_regularizer=0.01)

#transfer word & sentence conv filters
for l_name in ['word_conv','sentence_conv','hidden_0', 'final']:
    imdb_model.get_classification_model()\
              .get_layer(l_name).set_weights(weights=amazon_review_model
                                .get_classification_model()
                                .get_layer(l_name).get_weights())
```

After training for a few epochs, only fine-tuning the hidden layer and the final sigmoid layer, we get **86%** test accuracy on the 25k test set we have. If we try training an SVM model on this small dataset and predict for the entire 25k test set, we could get only 82% accuracy. So, clearly transfer learning is helping in building a better model even when we have less data.

Training on the full IMDB dataset with Word2vec embeddings

Now, let's try to train the document CNN model on the full IMDB dataset by transferring the learned Word2vec embeddings.

Note, we are not using the learned weights from the Amazon Review model. We will be training the model from scratch. In fact, this is what was done in the paper.

This code for this is very similar to the preceding IMDB training code. You just need to exclude the weight loading part from the Amazon model. The code is available in the repository, in the module named `imdb_model.py`. Also, here are the model parameters:

```
{
    "embedding_dim":50,
    "train_embedding":true,
    "embedding_regularizer_l2":0.0,
    "sentence_len":30,
    "num_sentences":20,
    "word_kernel_size":5,
    "word_filters":30,
    "sent_kernel_size":5,
    "sent_filters":16,
    "sent_k_maxpool":3,
    "input_dropout":0.4,
    "doc_k_maxpool":5,
    "sent_dropout":0.2,
    "hidden_dims":64,
    "conv_activation":"relu",
    "hidden_activation":"relu",
    "hidden_dropout":0,
    "num_hidden_layers":1,
    "hidden_gaussian_noise_sd":0.3,
    "final_layer_kernel_regularizer":0.04,
    "hidden_layer_kernel_regularizer":0.0,
    "learn_word_conv":true,
    "learn_sent_conv":true,
    "num_units_final_layer":1
}
```

While training, we have used another trick to avoid overfitting. We freeze the embedding layers (that is, `train_embedding=False`) after the first 10 epochs and only train the remaining layers. After 50 epochs, we achieved **89%** accuracy on the IMDB dataset, which is the result claimed in this paper. We observed that if we don't initialize the embedding weights before training, the model starts overfitting and is not able to achieve accuracy validation beyond **80%**.

Creating document summaries with CNN model

A review has many sentences. Some of these sentences are neutral and some are redundant to determine the polarity of the overall document. Summarizing the review or highlighting the sentences in the review where the user actually expresses the opinion is very useful. In fact, it also gives an explanation for the prediction we make and thus makes the model interpretable.

As explained in the paper, the first step for text summarization is to create a saliency map for the document by assigning an importance score to each sentence. To generate the saliency map for a given document we can apply the following technique:

1. We first perform a forward pass through the network to generate a class prediction for the document.
2. We then construct a pseudo-label by inverting the network predictions.
3. Feed the pseudo-label to the training loss function as the true label. This choice of pseudo-label allows us to induce the greatest loss. This in turn will make backpropagation modify the weights of the sentence embeddings that contributed most to decide the class labels. So, in the case of an actually positive label, if we are passing a 0 as a pseudo-label then the strong positive sentence embeddings should see a maximal change, that is, a high gradient norm.
4. Calculate the derivative of the loss function with respect to the sentence embedding layer.
5. Sort sentences by gradient norm in descending order to get the most important sentences at the top.

Let's implement this in Keras. We have to do the preprocessing as before and get the `x_train` and `y_train` NumPy arrays. We will first load our trained IMDB model and the learned weights. Then, we also need to compile the model with the optimizer we used to get the derivative and the loss functions out of this model:

```
imdb_model = DocumentModel.load_model(config.MODEL_DIR+
                                '/imdb/model_02.json')
imdb_model.load_model_weights(config.MODEL_DIR+ '/imdb/model_02.hdf5')

model = imdb_model.get_classification_model()
model.compile(loss="binary_crossentropy", optimizer='rmsprop',
            metrics=["accuracy"])
```

Now, let's do the *step 1* mentioned beforehand, that is, the forward pass followed by generating the pseudo-labels:

```
preds = model.predict(x_train)
#invert predicted label
pseudo_label = np.subtract(1,preds)
```

To calculate the gradient, we will use the Keras function `model.optimizer.get_gradients()`:

```
#Get the learned sentence embeddings
sentence_ebd = imdb_model.get_sentence_model().predict(x_train)

input_tensors = [model.inputs[0], # input data
# how much to weight each sample by
                 model.sample_weights[0],
                 model.targets[0], # labels
                ]
#variable tensor at the sentence embedding layer
weights = imdb_model.get_sentence_model().outputs

#calculate gradient of the total model loss w.r.t
#the variables at sentence embd layer
gradients = model.optimizer.get_gradients(model.total_loss, weights)
get_gradients = K.function(inputs=input_tensors, outputs=gradients)
```

Now, we can calculate the gradient for one document (say, document number `10`) as follows:

```
document_number = 10
K.set_learning_phase(0)
inputs = [[x_train[document_number]], # X
          [1], # sample weights
          [[pseudo_label[document_number][0]]], # y
]
grad = get_gradients(inputs)
```

Now, we can sort the sentences by the gradient norm. We will use the same `nltk` `sent_tokenize` function that we used in the preprocessing to get to the text sentence:

```
sent_score = []
for i in range(Preprocess.NUM_SENTENCES):
    sent_score.append((i, -np.linalg.norm(grad[0][0][i])))

sent_score.sort(key=lambda tup: tup[1])
summary_sentences = [ i for i, s in sent_score[:4]]

doc = corpus[document_number]
```

```
label = y_train[document_number]
prediction = preds[document_number]
print(doc, label , prediction)

sentences = sent_tokenize(doc)
for i in summary_sentences:
    print(i, sentences[i])
```

Following is a **negative** review:

> Wow, what a great cast! Julia Roberts, John Cusack, Christopher Walken,
> Catherine Zeta-Jones, Hank Azaria...what's that? A script, you say? Now
> you're just being greedy! Surely such a charismatic bunch of thespians will
> weave such fetching tapestries of cinematic wonder that a script will be
> unnecessary? You'd think so, but no. America's Sweethearts is one missed
> opportunity after another. It's like everyone involved woke up before each
> day's writing/shooting/editing and though "You know what? I've been working
> pretty hard lately, and this is guaranteed to be a hit with all these big
> names, right? I'm just gonna cruise along and let somebody else carry the
> can." So much potential, yet so painful to sit through. There isn't a
> single aspect of this thing that doesn't suck. Even Julia's fat suit is
> lame.

From the first two sentences it seems very positive. Our predicted score for this document is 0.15, which is proper. Let's see what summary we get:

> 4 Surely such a charismatic bunch of thespians will weave such
>   fetching tapestries of cinematic wonder that a script will be
>   unnecessary?
> 2 A script, you say?
> 6 America's Sweethearts is one missed opportunity after another.

Lets take another positive example, here our model predicted 0.98:

> This is what I was expecting when star trek DS9 premiered. Not to slight
> DS9. That was a wonderful show in it's own right, however it never really
> gave the fans more of what they wanted. Enterprise is that show. While
> having a similarity to the original trek it differs enough to be original
> in it's own ways. It makes the ideas of exploration exciting to us again.
> And that was one of the primary ingredients that made the original so
> loved. Another ingredient to success was the relationships that evolved
> between the crew members. Viewers really cared deeply for the crew.
> Enterprise has much promise in this area as well. The chemistry between
> Bakula and Blalock seems very promising. While sexual tension in a show can
> often become a crutch, I feel the tensions on enterprise can lead to much
> more and say alot more than is typical. I think when we deal with such
> grand scale characters of different races or species even, we get some very
> interesting ideas and television. Also, we should note the performances,

```
Blalock is very convincing as Vulcan T'pol and Bacula really has a whimsy
and strength of character that delivers a great performance. The rest of
the cast delivered good performances also. My only gripes are as follows.
The theme. It's good it's different, but a little to light hearted for my
liking. We need something a little more grand. Doesn't have to be
orchestral. Maybe something with a little more electronic sound would
suffice. And my one other complaint. They sell too many adds. They could
fix this by selling less ads, or making all shows two parters. Otherwise
we'll end up seeing the shows final act getting wrapped up way too quickly
as was one of my complaints of Voyager.
```

Here is the summary:

```
2 That was a wonderful show in it's own right, however it never really
  gave the fans more of what they wanted.
5 It makes the ideas of exploration exciting to us again.
6 And that was one of the primary ingredients that made the original
  so loved.
8 Viewers really cared deeply for the crew.
```

You can see how nicely it picks up the summary sentences. You really don't need to go through the whole review to understand it. Therefore, this text CNN model is comparable to the state-of-the art models for the IMDB dataset and also, once learned, it can perform other advanced text analysis tasks, such as text summarization.

Multiclass classification with the CNN model

Let's now apply the same model for multiclass classification. We will be using the 20 NewsGroup dataset for this. To train a CNN model, this dataset is small. We will still try to do it with a simpler problem. As discussed before, the 20 classes in this dataset have quite a lot of mixing and with SVM we get a maximum of 70% accuracy. Here, we will take the six broad categories of this dataset and try to build a CNN classifier. So, first we will map the 20 categories to the six broad categories. Following is the code to first load the dataset from scikit learn:

```
def load_20newsgroup_data(categories = None, subset='all'):
        data = fetch_20newsgroups(subset=subset,
                                 shuffle=True,
                                 remove=('headers', 'footers', 'quotes'),
                                 categories = categories)
        return data

dataset = Loader.load_20newsgroup_data(subset='train')
corpus, labels = dataset.data, dataset.target
```

```
test_dataset = Loader.load_20newsgroup_data(subset='test')
test_corpus, test_labels = test_dataset.data, test_dataset.target
```

Next, we will map the 20 classes to the six categories as follows:

```
six_groups = {
    'comp.graphics':0,'comp.os.ms-
        windows.misc':0,'comp.sys.ibm.pc.hardware':0,
    'comp.sys.mac.hardware':0, 'comp.windows.x':0,
    'rec.autos':1, 'rec.motorcycles':1, 'rec.sport.baseball':1,
    'rec.sport.hockey':1,
    'sci.crypt':2, 'sci.electronics':2,'sci.med':2, 'sci.space':2,
    'misc.forsale':3,
    'talk.politics.misc':4, 'talk.politics.guns':4,
    'talk.politics.mideast':4,
    'talk.religion.misc':5, 'alt.atheism':5, 'soc.religion.christian':5
    }

map_20_2_6 = [six_groups[dataset.target_names[i]] for i in range(20)]
labels = [six_groups[dataset.target_names[i]] for i in labels]
test_labels = [six_groups[dataset.target_names[i]] for i in
                test_labels]
```

We will do the same preprocessing steps followed by model initialization. Here, also, we have used GloVe embeddings to initialize the word embedding vectors. The detailed code is available in the repository, in the `20newsgrp_model` module. Here are the model's hyperparameters:

```
{
    "embedding_dim":50,
    "train_embedding":false,
    "embedding_regularizer_l2":0.0,
    "sentence_len":30,
    "num_sentences":10,
    "word_kernel_size":5,
    "word_filters":30,
    "sent_kernel_size":5,
    "sent_filters":20,
    "sent_k_maxpool":3,
    "input_dropout":0.2,
    "doc_k_maxpool":4,
    "sent_dropout":0.3,
    "hidden_dims":64,
    "conv_activation":"relu",
    "hidden_activation":"relu",
    "hidden_dropout":0,
    "num_hidden_layers":2,
    "hidden_gaussian_noise_sd":0.3,
```

```
    "final_layer_kernel_regularizer":0.01,
    "hidden_layer_kernel_regularizer":0.0,
    "learn_word_conv":true,
    "learn_sent_conv":true,
    "num_units_final_layer":6
}
```

Here are the detailed results of the model on the test set:

```
                precision recall f1-score support
0                   0.80    0.91      0.85    1912
1                   0.86    0.85      0.86    1534
2                   0.75    0.79      0.77    1523
3                   0.88    0.34      0.49     382
4                   0.78    0.76      0.77    1027
5                   0.84    0.79      0.82     940

avg / total         0.81    0.80      0.80    7318

[[1733   41  114    1   14    9]
 [  49 1302  110   11   47   15]
 [ 159   63 1196    5   75   25]
 [ 198   21   23  130    9    1]
 [  10   53   94    0  782   88]
 [  22   30   61    0   81  746]]
0.8047280677780815
```

Let's try an SVM on this dataset and see what is the best accuracy we get:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
tv = TfidfVectorizer(use_idf=True, min_df=0.00005, max_df=1.0,
                     ngram_range=(1, 1), stop_words = 'english',
                     sublinear_tf=True)
tv_train_features = tv.fit_transform(corpus)
tv_test_features = tv.transform(test_corpus)

clf = SVC(C=1,kernel='linear', random_state=1, gamma=0.01)
svm=clf.fit(tv_train_features, labels)
preds_test = svm.predict(tv_test_features)


from sklearn.metrics import
      classification_report,accuracy_score,confusion_matrix

print(classification_report(test_labels, preds_test))
print(confusion_matrix(test_labels, preds_test))
print(accuracy_score(test_labels, preds_test))
```

The following are the results from an SVM model. We have tuned the parameter *C* such that we get the best cross-validation accuracy:

```
                precision  recall  f1-score  support

    0               0.86    0.89      0.87     1912
    1               0.83    0.89      0.86     1534
    2               0.75    0.78      0.76     1523
    3               0.87    0.73      0.80      382
    4               0.82    0.75      0.79     1027
    5               0.85    0.76      0.80      940

    avg / total     0.82    0.82      0.82     7318

  0.82344902978956
```

Thus, we see that this text CNN model can give comparable results in case of multi-class classification results as well. Again, as before, now one can use the trained model to perform text summarization as well.

Visualizing document embeddings

In our document CNN model, we have the document embedding layer. Let's try to visualize what features the model has learned at this layer. We will first take the test set and calculate the document embeddings as follows:

```
doc_embeddings = newsgrp_model.get_document_model().predict(x_test)
print(doc_embeddings.shape)

(7318, 80)
```

We get 80 dimensional embedding vectors for all the test documents. To visualize these vectors, we will use the popular t-SNE dimentionality reduction technique to project the vectors in two-dimensionality space and graph a scatter plot as follows:

```
from utils import scatter_plot

doc_proj = TSNE(n_components=2, random_state=42,
                ).fit_transform(doc_embeddings)
f, ax, sc, txts = scatter_plot(doc_proj, np.array(test_labels))
```

The output of the preceding code is as follows:
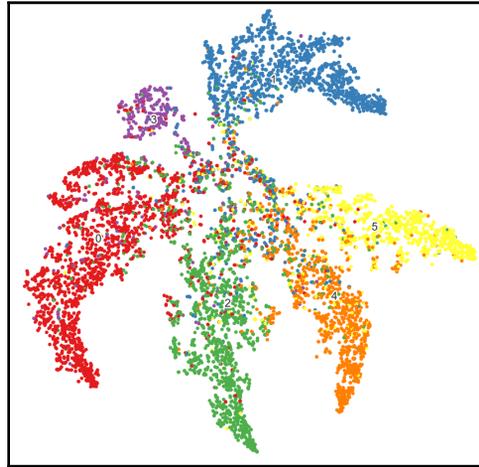
The labels (0-5) on the scatter represent the six classes. As you can see, the model has learned decent embeddings and is able to separate the six classes well in the 80-dimensional space. We can use these embeddings for other text analytics tasks, such as *information retrieval* or *text search*. Given a query document, we can compute its dense embedding and then compare that with similar embeddings in the entire corpus. This can help us boost the keyword based query results and improve retrieval performance.

Summary

We have learned about concepts from natural language processing, text classification, text summarization, and the application of deep learning CNN models in text domain. We have seen that transfer learning in terms of word embedding is the default first step in most of the use cases, especially if we have less training data. We have seen how to apply transfer learning for the text CNN model, learned on a huge Amazon product review dataset, to make predictions on a small movie review dataset, a related but not the same domain.

Also, we have learned here how we can use the learned CNN model for other text processing tasks, such as the summarization and representation of documents as dense vectors, which can be used in an information retrieval system to improve the retrieval performance.

8
Audio Event Identification and Classification

We have looked at some really interesting case studies on applying transfer learning to real-world problems in the previous chapters. Image and text data are two forms of unstructured data that we have tackled previously. We have demonstrated various ways to apply transfer learning to get more robust and superior models, and also to tackle constraints such as having less training data. In this chapter, we will tackle the new real-world problem of identifying and classifying audio events.

Creating pretrained deep learning models for audio data is a huge challenge because we do not have the advantage of efficient pretrained visual models such as the VGG or Inception (available for image data) or word-embedding based models such as Word2vec or GloVe (available for text data). The question then might arise as to what might be our strategy for audio data. We will explore some innovative approaches in this chapter, so stay tuned! We will be covering the following major aspects in this chapter:

- Understanding audio event classification
- Formulating our real-world problem
- Exploratory analysis of audio events
- Feature engineering and representation of audio events
- Audio event classification with transfer learning
- Building a deep learning audio event identifier

We will look at a real-world case study of identifying and classifying audio events in this chapter. Concepts such as audio feature engineering, transfer learning, deep learning, and object oriented programming will be used to build a robust, automated audio event identifier for classification. The code for this chapter is available for quick reference in the `Chapter 8` folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

Understanding audio event classification

By now, you should understand the basic task of classification or categorization, where we have structured or unstructured data that is usually labeled or annotated with specific groups or categories. The main task in automated classification is to build a model such that, with future data points, we can classify each data point or record into one of the specific categories based on various data attributes or features.

We have already looked at text and image classification in the previous chapters. In this chapter, we will look at classifying audio events. An audio event is basically an event or occurrence of an activity that is usually captured by an audio signal. Usually, short audio clips are used to represent audio events since, even if they are recurring, the sounds are usually similar. Sometimes, though, longer audio clips might be used to represent more complex audio events. Examples of audio events could be children playing in a playground, a siren alarm, dogs barking, and so on. In fact, Google has built a massive dataset called **AudioSet** (`https://research.google.com/audioset/index.html`), which is a large-scale dataset of manually annotated audio events, and they have also published several papers around audio event identification and classification. We will be using a smaller dataset for our problem, but interested readers should definitely check out this massive dataset of over 632 audio event classes consisting of a collection of 2,084,320 human-labeled 10-second sound clips drawn from YouTube videos.

Formulating our real-world problem

The main objective of our real-world case study here is audio event identification and classification. This is a supervised learning problem where we will be working on an audio event dataset with samples of audio data that belong to specific categories (which are the sources of the sounds).

We will be leveraging concepts from transfer learning and deep learning to build a robust classifier whereby, with any given audio sample belonging to one of our pre-determined categories, we should be able to correctly predict the source of this sound. The dataset we will be using is popularly known as the UrbanSound8K dataset (`https://urbansounddataset.weebly.com/`), and has 8,732 labeled audio sound files (the duration of which are usually equal to or greater than 4 seconds) that contain excerpts from common urban sounds. The ten categories of sounds in this dataset are as follows:

- `air_conditioner`
- `car_horn`
- `children_playing`

- `dog_bark`
- `drilling`
- `engine_idling`
- `gun_shot`
- `jackhammer`
- `siren`
- `streen_music`

For a detailed description of this dataset and other potential datasets and initiatives, we recommend readers to visit the UrbanSound website and also check out this amazing paper by the creators, J. Salamon, C. Jacoby, and J. P. Bello, *A Dataset and Taxonomy for Urban Sound Research* (`http://www.justinsalamon.com/uploads/4/3/9/4/4394963/salamon_urbansound_acmmm14.pdf`), 22nd ACM International Conference on Multimedia, Orlando USA, November 2014. We would like to thank them, as well as NYU's **Center for Urban Science and Progress** (**CUSP**), for making this a reality.

To get the data, you will need to fill out a form on their website, after which you should get a download link over email. Once you unzip the file, you should be able to see all the audio files in ten folders (ten folds) and a `readme` file containing more details about the dataset.

Exploratory analysis of audio events

We will follow a standard workflow of analyzing, visualizing, modeling, and evaluating our models on our audio data. Once all the data is downloaded, you will notice that there are a total of ten folders containing audio data samples in `WAV` format. We also have a metadata folder, which contains metadata information for each audio file in the `UrbanSound8K.csv` file. You can use this file to assign the class labels for each file or you can understand the file naming nomenclature to do the same.

Each audio file is named in a specific format. The name takes the `[fsID]-[classID]-[occurrenceID]-[sliceID].wav` format, which is populated as follows:

- `[fsID]`: The freesound ID of the recording from which this excerpt (slice) is taken
- `[classID]`: A numeric identifier of the sound class

- [occurrenceID]: A numeric identifier to distinguish different occurrences of the sound within the original recording
- [sliceID]: A numeric identifier to distinguish different slices taken from the same occurrence

Each class identifier is a numeric number that can be mapped to a specific class label. We will expand more on this shortly. Let's start with some basic exploratory analysis of our audio data. You can refer to the Exploratory Analysis Sound Data.ipynb Jupyter Notebook from our GitHub repository in case you want to run the examples yourself.

To start with, we load up the following dependencies, including the librosa module, which you might need to install if you don't have it:

```
import glob
import os
import librosa
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import specgram
import pandas as pd
import librosa.display
import IPython.display
import soundfile as sf

%matplotlib inline
```

The librosa module is an excellent open source Python framework for audio and music analysis. We recommend that our readers check out this framework in more detail. We will be using this for analyzing as well as extracting features from audio data in subsequent sections. Let's load one folder of data now for analysis:

```
files = glob.glob('UrbanSound8K/audio/fold1/*')
len(files)

873
```

We can see that each folder roughly contains 870+ audio samples. Based on information from the metadata and readme files, we can now create a class ID for name mapping the audio sample categories:

```
class_map = {'0' : 'air_conditioner', '1' : 'car_horn',
             '2' : 'children_playing', '3' : 'dog_bark',
             '4' : 'drilling', '5' : 'engine_idling',
             '6' : 'gun_shot', '7' : 'jackhammer',
             '8' : 'siren', '9' : 'street_music'}
pd.DataFrame(sorted(list(class_map.items())))
```

Let's take ten different audio samples belonging to each of these classes now, for further analysis:

```
samples = [(class_map[label],
            [f for f in files if f.split('-')[1] == label][0])
                                        for label in class_map.keys()]
samples

[('street_music', 'UrbanSound8K/audio/fold1\108041-9-0-11.wav'),
 ('engine_idling', 'UrbanSound8K/audio/fold1\103258-5-0-0.wav'),
 ('jackhammer', 'UrbanSound8K/audio/fold1\103074-7-0-0.wav'),
 ('air_conditioner', 'UrbanSound8K/audio/fold1\127873-0-0-0.wav'),
 ('drilling', 'UrbanSound8K/audio/fold1\14113-4-0-0.wav'),
 ('children_playing', 'UrbanSound8K/audio/fold1\105415-2-0-1.wav'),
 ('gun_shot', 'UrbanSound8K/audio/fold1\102305-6-0-0.wav'),
 ('siren', 'UrbanSound8K/audio/fold1\106905-8-0-0.wav'),
 ('car_horn', 'UrbanSound8K/audio/fold1\156194-1-0-0.wav'),
 ('dog_bark', 'UrbanSound8K/audio/fold1\101415-3-0-2.wav')]
```

Now that we have our sample data files, we still need to read in the audio data into memory before we can do any analysis. We noticed that librosa was throwing an error for some of the audio files (because they were of very short length or sample rates). Hence, we leverage the soundfile Python framework to read in the audio files to get their raw data and their original sample rate. You can get more information about the soundfile framework here: https://pypi.org/project/SoundFile/ if needed.

Audio sample rate is defined as the number of samples of audio carried per second, which is usually measured in Hz or kHz (one kHz being 1,000 Hz). The default sampling rate for librosa is 22,050 Hz, which is what we will be re-sampling all our audio data into to maintain consistency. The following code helps us read in the data and also shows the total length of the raw audio data:

```
def get_sound_data(path, sr=22050):
data, fsr = sf.read(path)
data_22k = librosa.resample(data.T, fsr, sr)
if len(data_22k.shape) > 1:
    data_22k = np.average(data_22k, axis=0)
    return data_22k, sr

sample_data = [(sample[0], get_sound_data(sample[1])) for sample in
               samples]
[(sample[0], sample[1][0].shape) for sample in sample_data]
```

```
[('street_music', (88200,)), ('engine_idling', (88200,)),
 ('jackhammer', (88200,)), ('air_conditioner', (44982,)),
 ('drilling', (88200,)), ('children_playing', (88200,)),
 ('gun_shot', (57551,)), ('siren', (88200,)),
 ('car_horn', (5513,)), ('dog_bark', (88200,))]
```

It is quite evident that most of the audio samples are around four seconds in duration but some have a really short duration. The beauty of Jupyter Notebooks is that you can even embed the audio in the notebook itself and play it using the following snippet.

For data in `sample_data`:

```
print(data[0], ':')
IPython.display.display(IPython.display.Audio(data=data[1][0],rate=data[
1][1]))
```

This creates the following:

Let's now visualize what these different audio sources look like by plotting their waveforms. Typically, this will be a waveform amplitude plot for each audio sample:

```
i = 1
fig = plt.figure(figsize=(15, 6))
for item in sample_data:
    plt.subplot(2, 5, i)
    librosa.display.waveplot(item[1][0], sr=item[1][1], color='r',
                             alpha=0.7)
    plt.title(item[0])
    i += 1
plt.tight_layout()
```

The created plots will appear as follows:
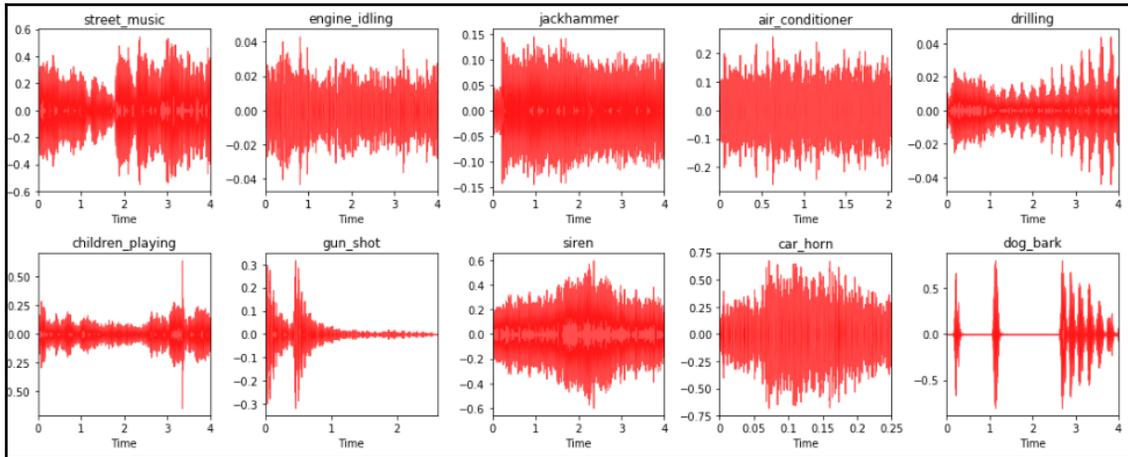
You can clearly see in the preceding diagram the different audio data samples with their source labels and their corresponding audio waveform plot. This depicts some interesting insights. Sources like **engine_idling**, **jackhammer**, and **air_conditioner** typically have a constant sound, which doesn't change over time. Hence, you can notice the constant amplitude in the waveforms. The **siren** and **car_horn** also usually have a constant audio waveform with intermittent increases of amplitude. The **gun_shot** usually have a huge noise in the beginning, followed by silence. And the **dog_bark** comes in intermittent intervals. Hence, sounds have short intervals of high amplitude besides silence. Can you find any more interesting patterns?

Another interesting visualization technique for audio data is spectrograms. Typically, a spectrogram is a visual representation technique for representing spectrums of frequencies from audio data. They are also popularly known as **sonographs** and **voicegrams**. Let's visualize our audio samples as spectrograms:

```
i = 1
fig = plt.figure(figsize=(15, 6))

for item in sample_data:
    plt.subplot(2, 5, i)
    specgram(item[1][0], Fs=item[1][1])
    plt.title(item[0])
    i += 1
plt.tight_layout()
```

The spectrograms appear as follows:
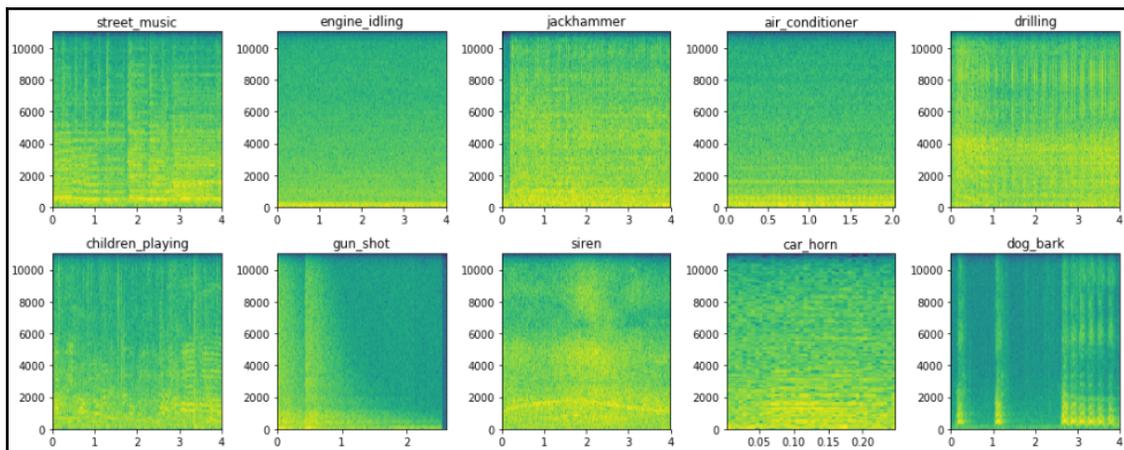
We can see how audio data can be represented as a nice image representation with a spectrogram, which could be useful for models like **Convolutional Neural Networks** (**CNNs**) to extract features from, as we can definitely see some marked differences in the spectrograms across the different audio sources. However, we will be using a mel spectrogram, which is usually better than a basic spectrogram because it represents the spectrogram on a mel-scale. The name **mel** comes from the word *melody*. This indicates that the scale is based on pitch comparisons. The mel-scale is thus a perceptual scale of pitches that have been judged by listeners to be equal in distance from one another. This is extremely useful if we are using CNNs to extract features from these spectrograms. The following snippet depicts mel spectrograms:

```
i = 1
fig = plt.figure(figsize=(15, 6))
for item in sample_data:
    plt.subplot(2, 5, i)
    S = librosa.feature.melspectrogram(item[1][0], sr=item[1]
    [1],n_mels=128)
    log_S = librosa.logamplitude(S)
    librosa.display.specshow(log_S, sr=item[1][1],
    x_axis='time',y_axis='mel')
    plt.title(item[0])
    plt.colorbar(format='%+02.0f dB')
    i += 1
plt.tight_layout()
```

The mel spectrograms appear as follows:
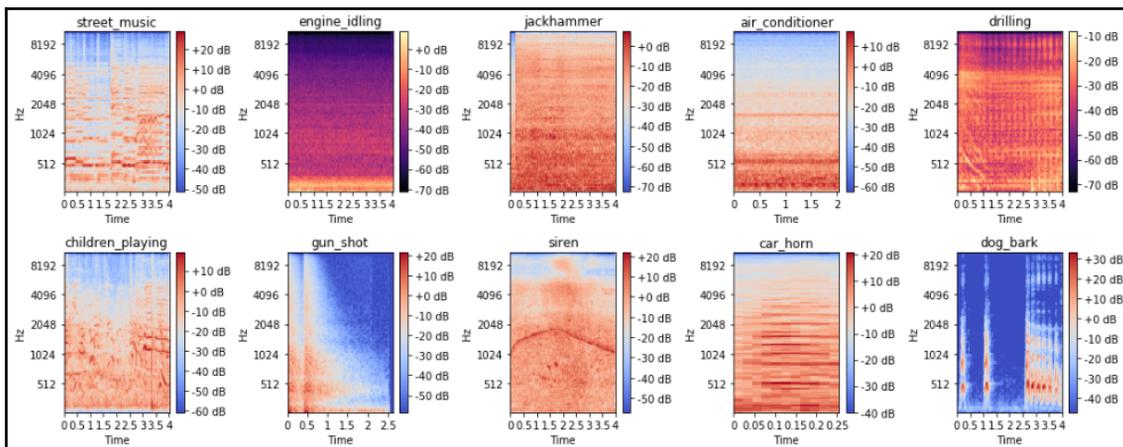
We can see that with the mel-scale, the spectrograms are much easier to distinguish based on the audio sources. Let's now focus on a few specific visual techniques that we will be using as our base sources for feature engineering in the next section. To start with, let's look at what our `gun_shot` audio sample looks like as a mel spectrogram:

```
y = sample_data[6][1][0]
S = librosa.feature.melspectrogram(y, sr=22050, n_mels=128)
log_S = librosa.logamplitude(S)
plt.figure(figsize=(12,4))
librosa.display.specshow(log_S, sr=22050, x_axis='time', y_axis='mel')
plt.colorbar(format='%+02.0f dB')
```

The spectrogram appears as follows:
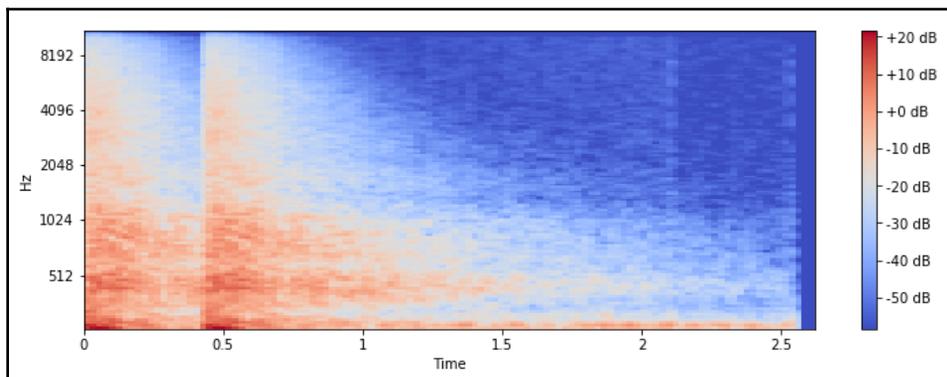
The spectrogram is consistent with how the audio waveform plot is for this audio source. Another interesting aspect of audio is that usually any audio time series data can be decomposed into harmonic and percussive components. These can present completely new and interesting representations of any audio sample. Let's get these components and plot them as spectrograms:

```
y_harmonic, y_percussive = librosa.effects.hpss(y)
S_harmonic   = librosa.feature.melspectrogram(y_harmonic,sr=22050,
                                               n_mels=128)
S_percussive = librosa.feature.melspectrogram(y_percussive,sr=22050)
log_Sh = librosa.power_to_db(S_harmonic)
log_Sp = librosa.power_to_db(S_percussive)

# Make a new figure
plt.figure(figsize=(12,6))
plt.subplot(2,1,1)
librosa.display.specshow(log_Sh, sr=sr, y_axis='mel')
plt.title('mel power spectrogram (Harmonic)')
plt.colorbar(format='%+02.0f dB')
plt.subplot(2,1,2)
librosa.display.specshow(log_Sp, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram (Percussive)')
plt.colorbar(format='%+02.0f dB')
plt.tight_layout()
```

The spectrograms will appear as follows:
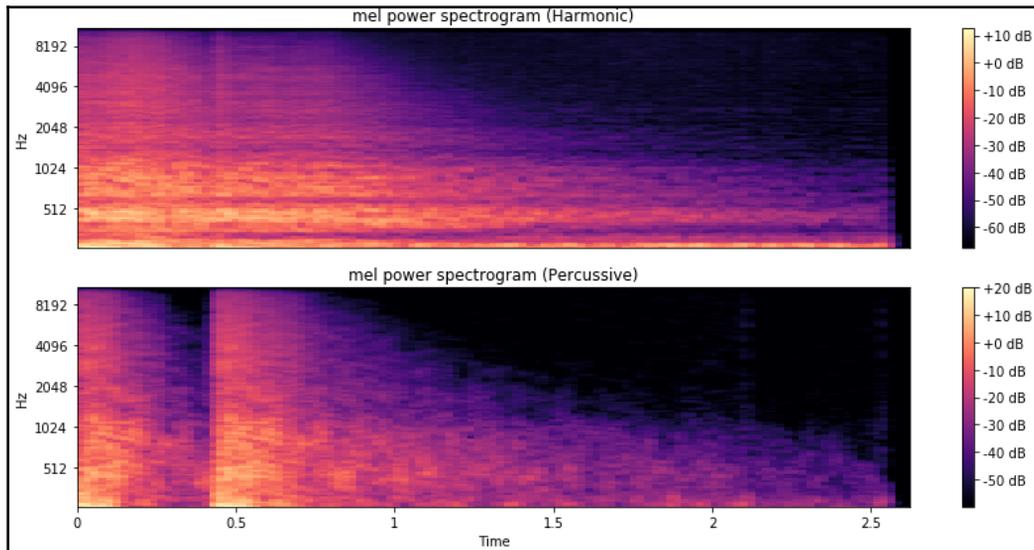
You can see the two different components of our audio sample shown as two distinctive spectrograms, depicting the harmonic and percussive components respectively.

Another very interesting depiction of audio data is to use a chromagram, which shows the pitch intensities of the audio sample signal based on the twelve different pitch classes—namely {C, C#, D, D#, E, F, F#, G, G#, A, A#, and B}. This can be an excellent visual tool for depicting the various pitch intensities in the audio signal over time. Typically, a Fourier transform or a Q-transform is done on the raw audio signal before building a chromagram:

```
C = librosa.feature.chroma_cqt(y=y_harmonic, sr=sr)
# Make a new figure
plt.figure(figsize=(12, 4))
# Display the chromagram: the energy in each chromatic pitch class
# as a function of time
librosa.display.specshow(C, sr=sr, x_axis='time', y_axis='chroma',
                         vmin=0, vmax=1)
plt.title('Chromagram')
plt.colorbar()
plt.tight_layout()
```

The chromogram will appear as follows:
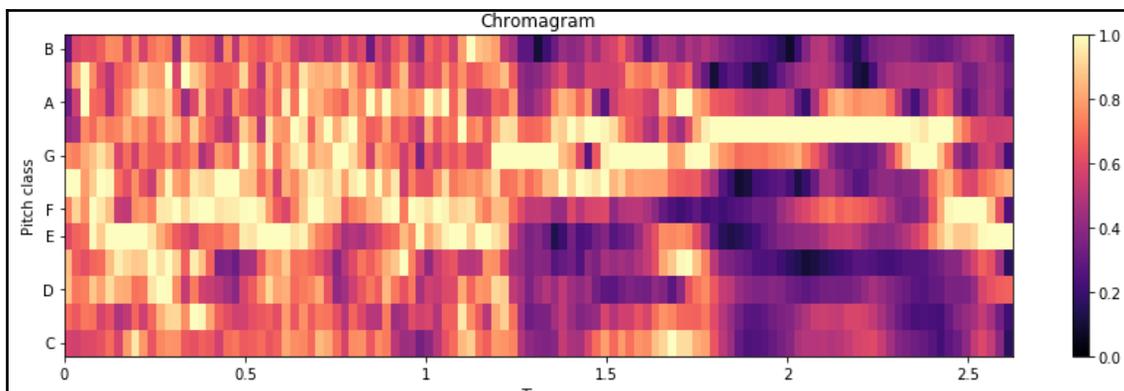
We can clearly see the various pitch intensities of our `gun_shot` audio sample over time, which will definitely be effective as a base image for feature extraction. We will use some of these techniques for feature extraction in the next section.

Feature engineering and representation of audio events

To build a robust classification model, we need robust and good feature representations from our raw audio data. We will leverage some of the techniques learned in the previous section for feature engineering. The code snippets used in this section are also available in the `Feature Engineering.ipynb` Jupyter Notebook, in case you want to run the examples yourself. We will reuse all the libraries we previously imported and we will also leverage `joblib` here to save our features to disk:

```
from sklearn.externals import joblib
```

Next, we will load up all our file names and define some utility functions to read in audio data and also enable us to get window indices for audio sub-samples, which we will be leveraging shortly:

```
# get all file names
ROOT_DIR = 'UrbanSound8K/audio/'
files = glob.glob(ROOT_DIR+'/**/*')

# load raw audio data
def get_sound_data(path, sr=22050):
    data, fsr = sf.read(path)
    data_resample = librosa.resample(data.T, fsr, sr)
    if len(data_resample.shape) > 1:
        data_resample = np.average(data_resample, axis=0)
    return data_resample, sr

# function to get start and end indices for audio sub-sample
def windows(data, window_size):
    start = 0
    while start < len(data):
        yield int(start), int(start + window_size)
        start += (window_size / 2)
```

The feature engineering strategy we will be following is slightly complex, but we will try and explain it here in a concise way. We have already seen that our audio data samples are of varying length. However, if we want to build a robust classifier, our features need to be consistent per sample. Hence, we will be extracting audio sub-samples (of a fixed length) from each audio file and extracting features from each of these sub-samples.

We will be using a total of three feature engineering techniques to build three feature representation maps, which will ultimately give us a three-dimensional image feature map for each of our audio sub-samples. The following diagram depicts the workflow we will be adopting:
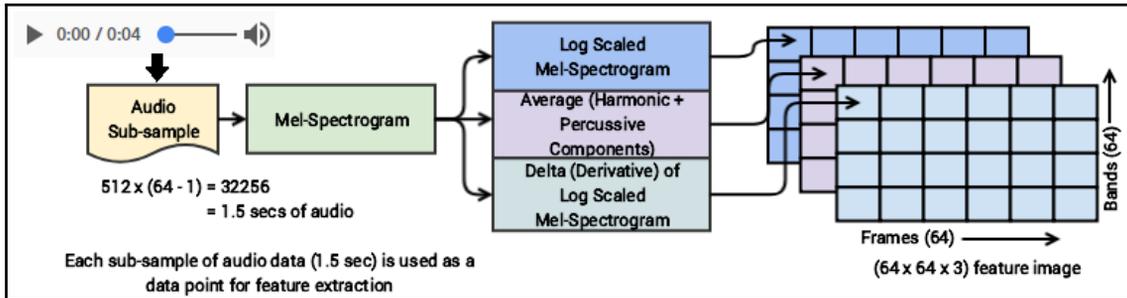
The idea for this came from an excellent paper by Karol J. Piczak, *Environmental sound classification with convolutional neural networks* (`https://ieeexplore.ieee.org/document/7324337/`), IEEE 2015. He leveraged mel spectrograms to general necessary features that can be consumed by CNNs for feature extraction. However, we have considered a couple more transformations for the final feature maps.

The first step is to define the total number of frames (columns) to be **64** and bands (rows) to be **64**, which forms the dimensions of each of our feature maps (64 x 64). Then, based on this, we extract windows of audio data, forming sub-samples from each audio data sample.

Considering each audio sub-sample, we start by creating a mel spectrogram. From this, we create a log-scaled mel spectrogram as one of the feature maps, an averaged feature map of the harmonic and percussive components (again log-scaled) of our audio sub-sample and the delta or derivative of our log-scaled mel spectrogram as the third feature map. Each of these feature maps can be represented as a 64 x 64 image and by combining them we get a 3-D feature map of dimensions (64, 64, 3) for each audio sub-sample. Let's define the function for this workflow now:

```
def extract_features(file_names, bands=64, frames=64):
    window_size = 512 * (frames - 1)
    log_specgrams_full = []
    log_specgrams_hp = []
    class_labels = []
    # for each audio sample
    for fn in file_names:
        file_name = fn.split('\')[-1]
        class_label = file_name.split('-')[1]
        sound_data, sr = get_sound_data(fn, sr=22050)
```

```
        # for each audio signal sub-sample window of data
        for (start,end) in windows(sound_data, window_size):
            if(len(sound_data[start:end]) == window_size):
                signal = sound_data[start:end]

                # get the log-scaled mel-spectrogram
                melspec_full = librosa.feature.melspectrogram(signal,
                                                        n_mels =
                                                            bands)
                logspec_full = librosa.logamplitude(melspec_full)
                logspec_full = logspec_full.T.flatten()[:,np.newaxis].T

                # get the log-scaled, averaged values for the
                # harmonic and percussive components
                y_harmonic, y_percussive =librosa.effects.hpss(signal)
                melspec_harmonic =
                        librosa.feature.melspectrogram(y_harmonic,
                                                    n_mels=bands)
                melspec_percussive =
                        librosa.feature.melspectrogram(y_percussive,
                                                    n_mels=bands)
                logspec_harmonic =
                        librosa.logamplitude(melspec_harmonic)
                logspec_percussive =
                        librosa.logamplitude(melspec_percussive)
                logspec_harmonic = logspec_harmonic.T.flatten()[:,
                                                    np.newaxis].T
                logspec_percussive = logspec_percussive.T.flatten()[:,
                                                    np.newaxis].T
                logspec_hp = np.average([logspec_harmonic,
                                        logspec_percussive],
                                        axis=0)
                log_specgrams_full.append(logspec_full)
                log_specgrams_hp.append(logspec_hp)
                class_labels.append(class_label)
    # create the first two feature maps
    log_specgrams_full = np.asarray(log_specgrams_full).reshape(
                                    len(log_specgrams_full), bands,
                                    frames, 1)
    log_specgrams_hp = np.asarray(log_specgrams_hp).reshape(
                                    len(log_specgrams_hp), bands,
                                    frames, 1)
    features = np.concatenate((log_specgrams_full,
                                log_specgrams_hp,
                                np.zeros(np.shape(
                                    log_specgrams_full))),
                                axis=3)
    # create the third feature map which is the delta (derivative)
```

```
                # of the log-scaled mel-spectrogram
                for i in range(len(features)):
                    features[i, :, :, 2] = librosa.feature.delta(features[i,
                                                                        :, :, 0])
                return np.array(features), np.array(class_labels, dtype = np.int)
```

We are now ready to use this function. We will be using it on all our 8,732 audio samples to create feature maps out of many sub-samples from this data, based on the strategy we discussed in the workflow earlier:

```
features, labels = extract_features(files)
features.shape, labels.shape
((30500, 64, 64, 3), (30500,))
```

We get a total of 30,500 feature maps from our 8,732 audio data files. This is excellent and, as we discussed earlier, each feature map is of dimensions (64, 64, 3). Let's now look at the overall class representation for our audio sources based on these 30,500 data points:

```
from collections import Counter
Counter(labels)
Counter({0: 3993, 1: 913, 2: 3947, 3: 2912, 4: 3405,
         5: 3910, 6: 336, 7: 3473, 8: 3611, 9: 4000})
```

We can see that the overall distribution of data points in different categories is quite uniform and decent. For some categories like 1 (car_horn) and 6 (gun_shot), the representation is quite low as compared to the other categories; this is expected, because audio data duration for these categories is typically much shorter than with the other categories. Let's go ahead and visualize these feature maps now:

```
class_map = {'0' : 'air_conditioner', '1' : 'car_horn', '2' :
             'children_playing','3' : 'dog_bark', '4' : 'drilling','5' :
             'engine_idling','6' : 'gun_shot', '7' : 'jackhammer', '8' :
             'siren', '9' : 'street_music'}
categories = list(set(labels))
sample_idxs = [np.where(labels == label_id)[0][0] for label_id in
               categories]
feature_samples = features[sample_idxs]

plt.figure(figsize=(16, 4))
for index, (feature_map, category) in enumerate(zip(feature_samples,
                                                categories)):
    plt.subplot(2, 5, index+1)
    plt.imshow(np.concatenate((feature_map[:,:,0],
                               feature_map[:,:,1],
                               feature_map[:,:,2]),
                              axis=1),
                              cmap='viridis')
```

```
    plt.title(class_map[str(category)])
plt.tight_layout()
t = plt.suptitle('Visualizing Feature Maps for Audio Clips')
```

The feature maps will appear as follows:
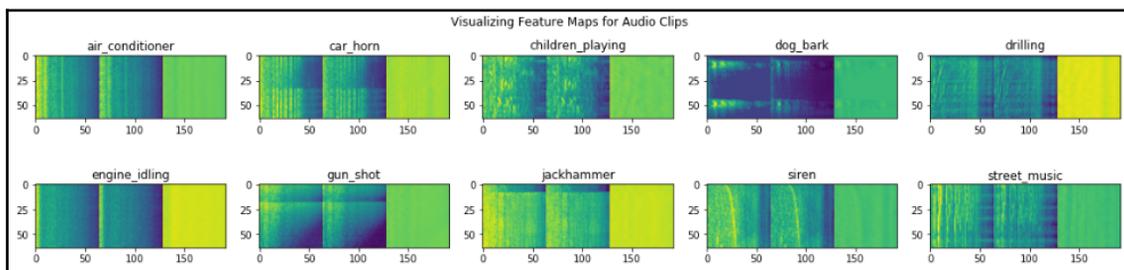
The preceding diagram shows us what some sample feature maps look like for each audio category and, as is evident, each feature map is a three-dimensional image. We will now save these base features to disk:

```
joblib.dump(features, 'base_features.pkl')
joblib.dump(labels, 'dataset_labels.pkl')
```

These base features will act as a starting point for further feature engineering in the next section, where we will unleash the true power of transfer learning.

Audio event classification with transfer learning

We are now ready to start working towards building our audio event classifier. We have our base feature maps, but we still need to do some more feature engineering. You can always build a CNN from scratch to ingest these images and then connect it to a fully connected deep **multilayer perceptron** (**MLP**) to build a classifier. However, we will be leveraging the power of transfer learning here by using one of the pretrained models for feature extraction. To be more specific, we will be using the VGG-16 model as a feature extractor and then train a fully-connected deep network on these features.

Building datasets from base features

The first step is to load our base features and create our train, validation and test datasets. For this, we will need to load our base features and labels from disk:

```
features = joblib.load('base_features.pkl')
labels = joblib.load('dataset_labels.pkl')
data = np.array(list(zip(features, labels)))
features.shape, labels.shape

((30500, 64, 64, 3), (30500,))
```

We will now randomly shuffle our data and create our train, validation, and test datasets:

```
np.random.shuffle(data)
train, validate, test = np.split(data,
[int(.6*len(data)),int(.8*len(data))])
train.shape, validate.shape, test.shape

((18300, 2), (6100, 2), (6100, 2))
```

Finally, we can also check the per class distribution in each of these datasets using the following snippet:

```
print('Train:', Counter(item[1] for item in train),'nValidate:',
Counter(item[1] for item in validate),'nTest:',Counter(item[1] for item
        in test))

Train: Counter({9: 2448, 2: 2423, 0: 2378, 5: 2366, 8: 2140,
                7: 2033, 4: 2020, 3: 1753, 1: 542, 6: 197})
Validate: Counter({0: 802, 5: 799, 2: 774, 9: 744, 8: 721,
                   7: 705, 4: 688, 3: 616, 1: 183, 6: 68})
Test: Counter({0: 813, 9: 808, 2: 750, 8: 750, 5: 745, 7: 735,
               4: 697, 3: 543, 1: 188, 6: 71})
```

Thus, we can see a consistent and uniform distribution of data points per class across the datasets.

Transfer learning for feature extraction

Now comes the interesting part. We are ready to leverage transfer learning to extract useful features from our base feature map images for each data point. For this, we will use an excellent pretrained deep learning model that has proven to be a very effective feature extractor on images. The VGG-16 model is what we will be using here. However, we will just use it as a plain feature extractor here without any fine-tuning (which is something we have explored in previous chapters).

Feel free to leverage fine-tuning, which may even lead to superior and better classifiers. We start by defining some basic utilities and functions to process our base images:

```
from keras.preprocessing import image
from keras.applications.imagenet_utils import preprocess_input
from PIL import Image
def process_sound_data(data):
    data = np.expand_dims(data, axis=0)
    data = preprocess_input(data)
    return data
```

We will now load up the VGG-16 model but only as a feature extractor; hence we will not be using its dense layers in the end:

```
from keras.applications import vgg16
from keras.models import Model
import keras
vgg = vgg16.VGG16(include_top=False, weights='imagenet',input_shape=
                (64, 64, 3))
output = vgg.layers[-1].output
output = keras.layers.Flatten()(output)
model = Model(vgg.input, output)
model.trainable = False
model.summary()
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 64, 64, 3)	0
block1_conv1 (Conv2D)	(None, 64, 64, 64)	1792
block1_conv2 (Conv2D)	(None, 64, 64, 64)	36928
... ...		

```
block5_conv3 (Conv2D)         (None, 4, 4, 512)          2359808
_____
block5_pool (MaxPooling2D)    (None, 2, 2, 512)          0
_____
flatten_2 (Flatten)           (None, 2048)               0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_____
```

From the preceding model summary, it is clear that our input base feature map image has dimensions of (64, 64, 3), from which we will end up getting a one-dimensional feature vector of size 2,048. Let's build a generic function that will help us leverage transfer learning and get these features, which are popularly known as **bottleneck features**:

```
def extract_tl_features(model, base_feature_data):
    dataset_tl_features = []
    for index, feature_data in enumerate(base_feature_data):
        if (index+1) % 1000 == 0:
            print('Finished processing', index+1, 'sound feature maps')
        pr_data = process_sound_data(feature_data)
        tl_features = model.predict(pr_data)
        tl_features = np.reshape(tl_features,
                                 tl_features.shape[1])
        dataset_tl_features.append(tl_features)
    return np.array(dataset_tl_features)
```

This function can now be used along with our VGG-16 model to extract useful features from each of our audio sub-sample base feature map images. We will do this for all our datasets:

```
# extract train dataset features
train_base_features = [item[0] for item in train]
train_labels = np.array([item[1] for item in train])
train_tl_features = extract_tl_features(model=model,
                          base_feature_data=train_base_features)

# extract validation dataset features
validate_base_features = [item[0] for item in validate]
validate_labels = np.array([item[1] for item in validate])
validate_tl_features = extract_tl_features(model=model,
                         base_feature_data=validate_base_features)

# extract test dataset features
test_base_features = [item[0] for item in test]
test_labels = np.array([item[1] for item in test])
```

```
test_tl_features = extract_tl_features(model=model,
                            base_feature_data=test_base_features)

train_tl_features.shape, validate_tl_features.shape, test_tl_features.shape

((18300, 2048), (6100, 2048), (6100, 2048))
```

We can now save these features and labels to disk so that we can use them later on, at any time, for building our classifiers, and we don't have to depend on keeping our Notebooks open all the time:

```
joblib.dump(train_tl_features, 'train_tl_features.pkl')
joblib.dump(train_labels, 'train_labels.pkl')
joblib.dump(validate_tl_features, 'validate_tl_features.pkl')
joblib.dump(validate_labels, 'validate_labels.pkl')
joblib.dump(test_tl_features, 'test_tl_features.pkl')
joblib.dump(test_labels, 'test_labels.pkl')
```

Building the classification model

We are now ready to build our classification model on the features we extracted in the previous section. The code for this section is available in the `Modeling.ipynb` Jupyter Notebook in case you want to run the examples yourself. To start, let's load some essential dependencies:

```
from sklearn.externals import joblib
import keras
from keras import models
from keras import layers
import model_evaluation_utils as meu
import matplotlib.pyplot as plt

%matplotlib inline
```

We will use our nifty model evaluation utilities module named `model_evaluation_utils` to evaluate our classifier and test its performance later on. Let's load up our feature sets and data point class labels now:

```
train_features = joblib.load('train_tl_features.pkl')
train_labels = joblib.load('train_labels.pkl')
validation_features = joblib.load('validate_tl_features.pkl')
validation_labels = joblib.load('validate_labels.pkl')
test_features = joblib.load('test_tl_features.pkl')
test_labels = joblib.load('test_labels.pkl')
train_features.shape, validation_features.shape, test_features.shape
```

```
((18300, 2048), (6100, 2048), (6100, 2048))


train_labels.shape, validation_labels.shape, test_labels.shape

((18300,), (6100,), (6100,))
```

Thus, we can see that all our feature sets and corresponding labels have been loaded. The input feature sets are one-dimensional vectors of size 2,048 obtained from the VGG-16 model we used in the previous section. We now need to one-hot encode the categorical class labels before we can feed them to a deep learning model. The following snippet helps us achieve this:

```
from keras.utils import to_categorical
train_labels_ohe = to_categorical(train_labels)
validation_labels_ohe = to_categorical(validation_labels)
test_labels_ohe = to_categorical(test_labels)
train_labels_ohe.shape, validation_labels_ohe.shape, test_labels_ohe.shape

((18300, 10), (6100, 10), (6100, 10))
```

We will now build our deep learning classifier by using a fully connected network that has four hidden layers. We will use the usual components like dropout to prevent overfitting and the Adam optimizer for our model. Details of the model architecture are depicted in the following code:

```
model = models.Sequential()
model.add(layers.Dense(1024, activation='relu',
          input_shape=(train_features.shape[1],)))
model.add(layers.Dropout(0.4))
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.Dropout(0.4))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(train_labels_ohe.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',metrics=['accuracy'])
model.summary()
```

```
_____
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1024)              2098176
_____
```

```
dropout_1 (Dropout)          (None, 1024)            0
_____
dense_2 (Dense)              (None, 1024)            1049600
_____
dropout_2 (Dropout)          (None, 1024)            0
_____
dense_3 (Dense)              (None, 512)             524800
_____
dropout_3 (Dropout)          (None, 512)             0
_____
dense_4 (Dense)              (None, 512)             262656
_____
dropout_4 (Dropout)          (None, 512)             0
_____
dense_5 (Dense)              (None, 10)              5130
=================================================================

Total params: 3,940,362
Trainable params: 3,940,362
Non-trainable params: 0
```

This model was then trained on an AWS **p2.x** instance for about fifty epochs with a batch size of 128. We recommend a GPU to train this model; you can play around with the epochs and batch size to get a robust model, as follows:

```
history = model.fit(train_features, train_labels_ohe,epochs=50,
                    batch_size=128,
                    validation_data=(validation_features,
                    validation_labels_ohe),shuffle=True, verbose=1)
Train on 18300 samples, validate on 6100 samples
Epoch 1/50
18300/18300 - 2s - loss: 2.7953 - acc: 0.3959 - val_loss: 1.0665 - val_acc:
0.6675
Epoch 2/50
18300/18300 - 1s - loss: 1.1606 - acc: 0.6211 - val_loss: 0.8179 - val_acc:
0.7444
...
...
Epoch 48/50
18300/18300 - 1s - loss: 0.2753 - acc: 0.9157 - val_loss: 0.4218 - val_acc:
0.8797
Epoch 49/50
18300/18300 - 1s - loss: 0.2813 - acc: 0.9142 - val_loss: 0.4220 - val_acc:
0.8810
Epoch 50/50
18300/18300 - 1s - loss: 0.2631 - acc: 0.9197 - val_loss: 0.3887 - val_acc:
0.8890
```

We get a validation accuracy of close to **89%**, which is pretty good and things look promising. We can also plot the overall accuracy and loss plots of the model to get a better idea of what things look like, as follows:

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
t = f.suptitle('Deep Neural Net Performance', fontsize=12)
f.subplots_adjust(top=0.85, wspace=0.2)
epochs = list(range(1,51))
ax1.plot(epochs, history.history['acc'], label='Train Accuracy')
ax1.plot(epochs, history.history['val_acc'], label='Validation Accuracy')
ax1.set_ylabel('Accuracy Value')
ax1.set_xlabel('Epoch')
ax1.set_title('Accuracy')
l1 = ax1.legend(loc="best")
ax2.plot(epochs, history.history['loss'], label='Train Loss')
ax2.plot(epochs, history.history['val_loss'], label='Validation Loss')
ax2.set_ylabel('Loss Value')
ax2.set_xlabel('Epoch')
ax2.set_title('Loss')
l2 = ax2.legend(loc="best")
```

This will create the following plots:
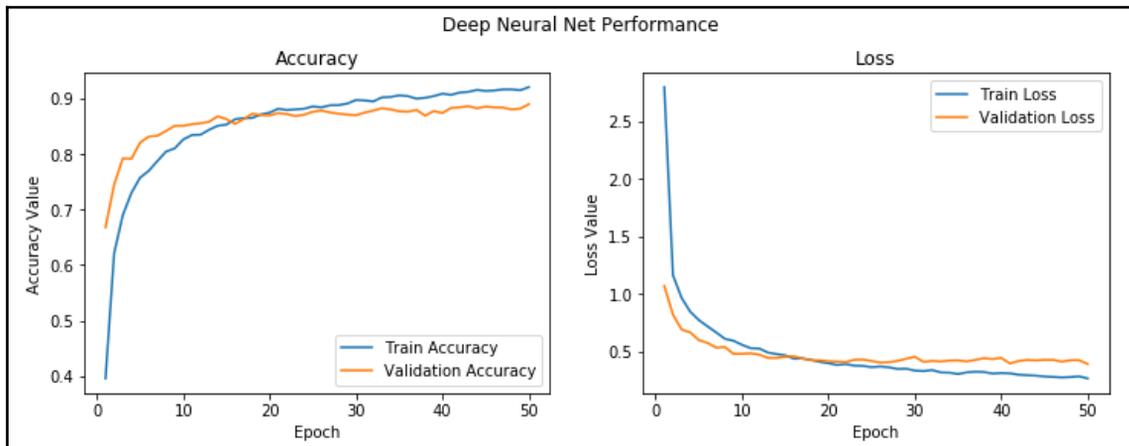
We can see that the model's loss and accuracy between training and validation is quite consistent; maybe it overfits slightly, but it is negligible considering their difference is quite low.

Evaluating the classifier performance

It is now time to put our model to the test, literally. We will use the test dataset to make predictions with our model and then evaluate them against the ground truth labels. For this, we first need to get our model's predictions on the test data and do a reverse mapping from the numeric labels to the actual text labels, using the following snippet:

```
predictions = model.predict_classes(test_features)
class_map = {'0' : 'air_conditioner', '1' : 'car_horn',
             '2' : 'children_playing', '3' : 'dog_bark',
             '4' : 'drilling', '5' : 'engine_idling',
             '6' : 'gun_shot', '7' : 'jackhammer',
             '8' : 'siren', '9' : 'street_music'}
test_labels_categories = [class_map[str(label)]for label in
                          test_labels]
prediction_labels_categories = [class_map[str(label)]for label in
                 predictions] category_names = list(class_map.values())
```

Let's use our `model_evaluation_utils` module now to evaluate our model's performance on the test data. We start by getting the overall performance metrics:

```
meu.get_metrics(true_labels=test_labels_categories,
                predicted_labels=prediction_labels_categories)

Accuracy: 0.8869
Precision: 0.8864
Recall: 0.8869
F1 Score: 0.8861
```

We get an overall model accuracy and `f1-score` of close to 89%, which is excellent and consistent with what we got from our validation dataset. Let's look at per class model performance next:

```
meu.display_classification_report(true_labels=test_labels_categories,
                       predicted_labels=prediction_labels_categories,
                       classes=category_names)
```

	precision	recall	f1-score	support
car_horn	0.87	0.73	0.79	188
siren	0.95	0.94	0.94	750
drilling	0.88	0.93	0.90	697
gun_shot	0.94	0.94	0.94	71
children_playing	0.83	0.79	0.81	750
air_conditioner	0.89	0.94	0.92	813
jackhammer	0.92	0.93	0.92	735
engine_idling	0.94	0.95	0.95	745

```
        dog_bark        0.87      0.83      0.85       543
     street_music       0.81      0.81      0.81       808

      avg / total       0.89      0.89      0.89      6100
```

This gives us a clearer perspective on the exact classes where the model is working really well and where it might be having trouble. Most of the classes seem to be working quite well, especially device sounds like `gun_shot`, `jackhammer`, `engine_idling`, and so on. It seems to have the most trouble with `street_music` and `children_playing`.

The confusion matrix might help us see where the most misclassifications might be happening and help us understand this even better:

```
meu.display_confusion_matrix_pretty(true_labels=test_labels_categories,
                        predicted_labels=prediction_labels_categories,
                        classes=category_names)
```

The matrix will appear as follows:

		car_horn	siren	drilling	gun_shot	children_playing	air_conditioner	jackhammer	engine_idling	dog_bark	street_music
	car_horn	137	4	15	0	3	3	4	1	2	19
	siren	1	705	5	0	7	18	1	2	6	5
	drilling	1	2	650	0	1	4	29	2	4	4
	gun_shot	0	0	1	67	0	0	0	1	2	0
Actual:	**children_playing**	2	11	13	1	592	14	2	7	31	77
	air_conditioner	2	0	6	1	8	768	7	10	2	9
	jackhammer	0	0	28	0	0	14	680	11	0	2
	engine_idling	1	0	3	0	4	10	4	707	8	8
	dog_bark	1	9	8	2	37	10	0	1	448	27
	street_music	12	15	12	0	64	18	10	7	14	656

We can see that most of the model predictions are correct, looking at the diagonal of the matrix, and this is excellent. With regard to misclassifications, we can see that a lot of samples belonging to `street_music`, `dog_bark`, and `children_playing` have been misclassified amongst each other, which is kind of expected considering all of these events happen outside in the open and there is a chance they might be happening together. The same holds good for `drilling` and `jackhammer`. Thankfully, there is very little overlap in misclassifications between `gun_shot` and `children_playing`.

Thus, we can see how effective transfer learning works in this complex case study, where we leveraged an image classifier to help us build a robust and effective audio event classifier. We can now save this model for future use with the following code:

```
model.save('sound_classification_model.h5')
```

You might be thinking now that this is good. But we did everything on a static dataset. How would we use this model in the real world for audio event identification and classification? We will talk about a strategy in the next section.

Building a deep learning audio event identifier

We will now look at a strategy using which we can build an actual audio event identifier by leveraging the classification model we built in the previous section. This will enable us to take any new audio file and predict which category it might belong to by making use of the entire workflow we defined in this chapter, starting from building the base feature maps, extracting features using the VGG-16 model, and then leveraging our classification model to make a prediction. The code snippets used in this section are also available in the `Prediction Pipeline.ipynb` Jupyter Notebook in case you want to run the examples yourself. The Notebook contains the `AudioIdentifier` class, which we have created by reusing all the components we have built in the previous sections of this chapter. Do refer to the Notebook to access the full code for this class as we will be focusing more on the actual prediction pipeline to keep the content more concise. We will start by initializing an instance of our class by feeding it the path of our classification model:

```
ai =
 AudioIdentifier(prediction_model_path='sound_classification_model.h5')
```

We have now downloaded three completely new audio data files belonging to three of the ten audio classes. Let's load them up, so we can test our model's performance on them:

```
siren_path = 'UrbanSound8K/test/sirenpolice.wav'
gunshot_path = 'UrbanSound8K/test/gunfight.wav'
dogbark_path = 'UrbanSound8K/test/dog_bark.wav'
siren_audio, siren_sr = ai.get_sound_data(siren_path)
gunshot_audio, gunshot_sr = ai.get_sound_data(gunshot_path)
dogbark_audio, dogbark_sr = ai.get_sound_data(dogbark_path)
actual_sounds = ['siren', 'gun_shot', 'dog_bark']
sound_data = [siren_audio, gunshot_audio, dogbark_audio]
sound_rate = [siren_sr, gunshot_sr, dogbark_sr]
sound_paths = [siren_path, gunshot_path, dogbark_path]
```

Let's visualize the waveforms of these three audio files and understand how they are structured:

```
i = 1
fig = plt.figure(figsize=(12, 3.5))
t = plt.suptitle('Visualizing Amplitude Waveforms for Audio Clips',
                    fontsize=14)
fig.subplots_adjust(top=0.8, wspace=0.2)

for sound_class, data, sr in zip(actual_sounds, sound_data,sound_rate):
    plt.subplot(1, 3, i)
    librosa.display.waveplot(data, sr=sr, color='r', alpha=0.7)
    plt.title(sound_class)
    i += 1
plt.tight_layout(pad=2.5)
```

The visualizations will appear as follows:
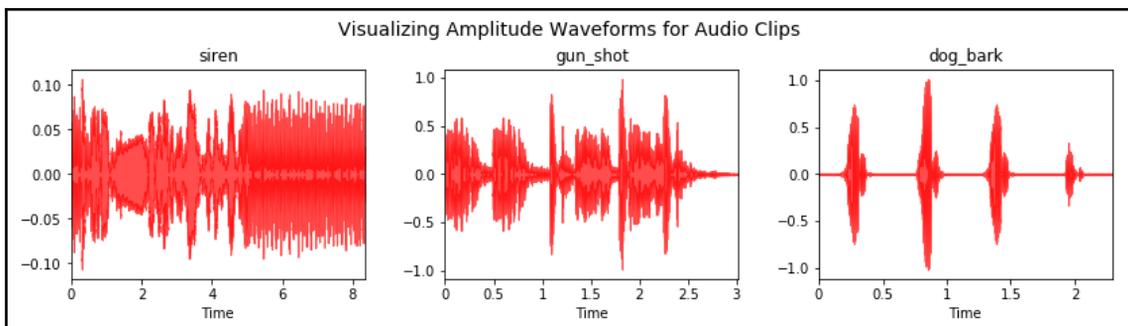
Based on the visualizations, they seem consistent based on the audio source and our pipeline is working well so far. Let's extract the base feature maps for these audio files now:

```
siren_feature_map = ai.extract_base_features(siren_audio)[0]
gunshot_feature_map = ai.extract_base_features(gunshot_audio)[0]
dogbark_feature_map = ai.extract_base_features(dogbark_audio)[0]
feature_maps = [siren_feature_map, gunshot_feature_map,dogbark_feature_map]
plt.figure(figsize=(14, 3))
t = plt.suptitle('Visualizing Feature Maps for Audio
                    Clips',fontsize=14)
fig.subplots_adjust(top=0.8, wspace=0.1)

for index, (feature_map, category) in
  enumerate(zip(feature_maps,actual_sounds)):
    plt.subplot(1, 3, index+1)
    plt.imshow(np.concatenate((feature_map[:,:,0],
                                feature_map[:,:,1],
```

```
                                         feature_map[:,:,2]), axis=1),
                                         cmap='viridis')
        plt.title(category)
        plt.tight_layout(pad=1.5)
```

The feature maps will appear as follows:
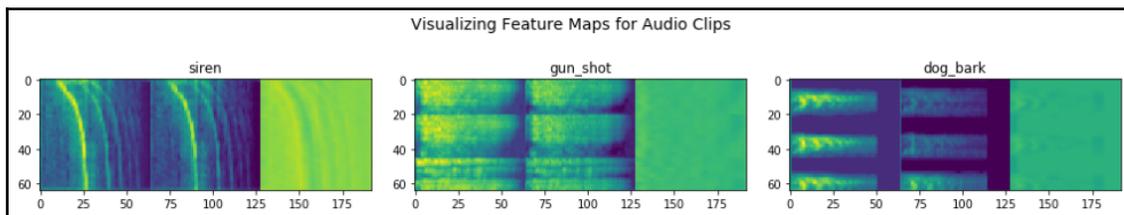
The image feature maps look quite consistent based on what we observed during our training phase. We are now ready to leverage our prediction pipeline to predict the audio source class for each of these sounds:

```
    predictions =
        [ai.prediction_pipeline(audiofile_path,return_class_label=True)
                        for audiofile_path in sound_paths]
    result_df = pd.DataFrame({'Actual Sound': actual_sounds,
                               'Predicted Sound': predictions,
                               'Location': sound_paths})
    result_df
```

We end up with the following predictions:

	Actual Sound	Location	Predicted Sound
0	siren	UrbanSound8K/test/sirenpolice.wav	siren
1	gun_shot	UrbanSound8K/test/gunfight.wav	gun_shot
2	dog_bark	UrbanSound8K/test/dog_bark.wav	dog_bark

Looks like our model was able to correctly identify all of these audio samples. We encourage you to check out the `AudioIdentifier` class in the Notebook to see how we have implemented the prediction pipeline behind the scenes. We have leveraged all the concepts we learned in this chapter to build out this pipeline.

Summary

We looked at a completely new problem and case study in this chapter, involving audio identification and classification. Concepts surrounding audio data and signals were covered, including effective techniques to visualize and understand this data type.

We also looked at effective feature engineering techniques and how we could use transfer learning to extract effective features from image representations of audio data. This shows us the promise of transfer learning and how you can leverage knowledge from one domain (images) to another domain (audio) and build an extremely robust and effective classifier. Finally, we built a complete end-to-end pipeline for identifying and classifying new samples of audio data. Do check out further datasets of annotated audio on the web, and see if you can build a bigger and better audio identifier and classifier by leveraging concepts from transfer learning that you learned here. Stay tuned for more interesting examples and case studies on transfer learning.

9
DeepDream

This chapter focuses on a gentle introduction to the domain of generative deep learning, which has been one of the core ideas at the forefront of true **artificial intelligence** (**AI**). We will be focusing on how **Convolutional Neural Networks** (**CNNs**) think or visualize patterns in images by leveraging transfer learning. They can generate image patterns never seen before depicting the way these convnets think or even dream! First released by Google in 2015, DeepDream became a viral sensation due to the interesting patterns deep networks started to generate from images. We will be covering the following major topics in this chapter:

- Motivation—psychological pareidolia
- Algorithmic pareidolia in computer vision
- Understanding what CNNs have learned by visualizing internal layers of CNN
- DeepDream algorithm and how to create your own dream

Just like the previous chapters, we will use a combination of conceptual knowledge powered with intuitive hands-on examples. The code for this chapter is available for quick reference in the `Chapter 9` folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

Introduction

Before we dive in detail of neural DeepDream, let's take a glance at a similar behavior we humans experience ourselves. Have you ever tried to look for shapes in clouds, the jitter and noisy signals in your television set or even seen a face burned into your toast?

Pareidolia is a psychological phenomenon that leads us to see patterns in a random stimulus; the tendency for humans to perceive a face or pattern where one actually doesn't exist. This often results in assigning human characteristics to objects. Please note the significance of the evolutionary consequences of seeing a pattern where there is none (a false positive) as opposed to failing to see a pattern where there is one (a false negative). For example, seeing a lion where this is no lion is rarely lethal; however, failing to see a predatory lion where there is one, of course, would often be deadly.

The neurological foundation for pareidolia resides primarily in an area of the temporal lobes of the brain called the **fusiform gyrus**, deep inside the brain, where humans and other animals have neurons dedicated to the recognition of faces and other objects.

Algorithmic pareidolia in computer vision

One of the major tasks in computer vision is object detection and face detection in particular. There are many electronic devices with face detection features that run such algorithms behind the scenes and detect a face. So, what happens when we put pareidolia-inducing objects in front of these software? Sometimes these software interpret faces the exact same way we do; sometimes it might agree with us, and sometimes it brings entirely new faces to our attention.

In the case of an object recognition system built using artificial neural network, higher-level features/layers correspond to more recognizable features, like faces or objects. Enhancing these features brings out what the computer sees. These reflect the training set of images that the network has seen previously. Let's take the Inception network, and ask it to predict the objects it sees in some pareidolia-inducing images. Let's take these pansy flowers in the following photo. To me, these flowers sometimes looks like a butterfly, and sometimes like a face of an angry man with a thick mustache:

Let's see what the Inception model sees in this. We will use the pretrained Inception network model trained on ImageNet data. To load the model, use the following code:

```
from keras.applications import inception_v3
from keras import backend as K
from keras.applications.imagenet_utils import decode_predictions
from keras.preprocessing import image
K.set_learning_phase(0)

model = inception_v3.InceptionV3(weights='imagenet',include_top=True)
```

To read the image file and convert it to a data batch with one image, which is the expected input for the `predict` function of the Inception network model, we use the following function:

```
def preprocess_image(image_path):
    img = image.load_img(image_path)
    img = image.img_to_array(img)
    #convert single image to a batch with 1 image
    img = np.expand_dims(img, axis=0)
    img = inception_v3.preprocess_input(img)
    return img
```

Now, let's use the preceding method to preprocess the input image and predict the object the model sees. We will use the `modeld.predict` method to get the predicted class probabilities of all 1,000 classes in ImageNet. To convert this array of probabilities to real class labels, ordered in decreasing order of probability score, we use the `decode_predictions` method from `keras`. The list of all 1,000 ImageNet classes or synsets can be found here: `http://image-net.org/challenges/LSVRC/2014/browse-synsets`. Note that the pansy flower is not in the known set of classes on which the model is trained:

```
img = preprocess_image(base_image_path)
preds = model.predict(img)
for n, label, prob in decode_predictions(preds)[0]:
    print (label, prob)
```

The predictions. None of the top-predicted classes are of any strong probability, and this is expected as the model has not seen this particular flower before:

```
bee 0.022255851
earthstar 0.018780833
sulphur_butterfly 0.015787734
daisy 0.013633176
cabbage_butterfly 0.012270376
```

In the preceding photo, the model finds a **bee**. Well, that's not a bad guess; as you can see, in the yellow flowers, the lower half of the central black/brown shade looks like a bee indeed. Also, it sees some yellow and white butterflies called **sulphur** and **cabbage** butterflies, like we humans may perceive at a quick glance. The following photo shows the actual images of these identified objects/classes. Clearly, some of the feature detector hidden layers in this network got activated with this input. Maybe the filters detecting wings in insects/birds got activated along with some color-related filters to arrive at the preceding conclusion:

cabbage butterfly   sulphur butterfly   bee

The ImageNet architecture and the number of feature maps in it is huge. Let's assume for a moment that we know the feature map layer that detects these wings. Now, given an input image, we can extract the features from this layer. Can we change the input image such that the activation from this layer increases? That means we have to modify the input image such that we see more of wing-like objects in the input image, even if they're not there. The resultant image will be one like a dream, with butterflies everywhere. This is exactly what is done in DeepDream.

Now let's take a look at some of the feature maps in the Inception network. To understand what a convolution model has learned, we can try to visualize the convolution filters.

Visualizing feature maps

Visualizing a CNN model involves looking at the intermediate layer feature maps that are output by various convolution and pooling layers in a network, given a certain input. This gives a view into how an input is processed by the network, and how various image features are hierarchically extracted. All feature maps have three dimensions: width, height, and depth (channels). We will try to visualize them for the InceptionV3 model.

Let's take the following input photo of a Labrador dog, and try to visualize various feature maps. As the InceptionV3 model has huge depth, we will visualize just a few of the layers:

First, let's create a model to take the input image and output all the internal activation layers. The activation layers in InceptionV3 are named as `activation_i`. So, we can filter out the activation layers from the loaded Inception model as shown in the following code:

```
activation_layers = [ layer.output for layer in model.layers if
                        layer.name.startswith("activation_")]

layer_names = [ layer.name for layer in model.layers if
                    layer.name.startswith("activation_")]
```

Now, let's create a model that takes the input image and outputs all aforementioned activation layer features as a list, as demonstrated in the following code:

```
from keras.models import Model
activation_model = Model(inputs=model.input, outputs=activation_layers)
```

Now, to get the output activations, we can use the `predict` function. We have to use the same preprocessing function defined precedingly to preprocess the image before feeding it to the Inception network:

```
img = preprocess_image(base_image_path)
activations = activation_model.predict(img)
```

We can plot these preceding activations. All the filter/feature maps in one activation layer can be plotted in a grid. So, based on the number of filters in a layer, we will define an image grid as a NumPy array, as shown in the following code (some parts of the following code are borrowed from https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html):

```python
import matplotlib.pyplot as plt

images_per_row = 8
idx = 1 #activation layer index

layer_activation=activations[idx]
# This is the number of features in the feature map
n_features = layer_activation.shape[-1]
# The feature map has shape (1, size1, size2, n_features)
r = layer_activation.shape[1]
c = layer_activation.shape[2]
# We will tile the activation channels in this matrix
n_cols = n_features // images_per_row
display_grid = np.zeros((r * n_cols, images_per_row * c))
print(display_grid.shape)
```

Now, we will loop through all the feature maps in the activation layer, and put the scaled output to the grid as shown in the following code:

```python
# We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,:, :, col *
                                                images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0,
                                    255).astype('uint8')
            display_grid[col * r : (col + 1) * r,
            row * c : (row + 1) * c] = channel_image
    # Display the grid
    scale = 1. / r
    plt.figure(figsize=(scale * display_grid.shape[1],
            scale * display_grid.shape[0]))
    plt.title(layer_names[idx]+" #filters="+str(n_features))
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')
```

Following are the outputs from various layers:
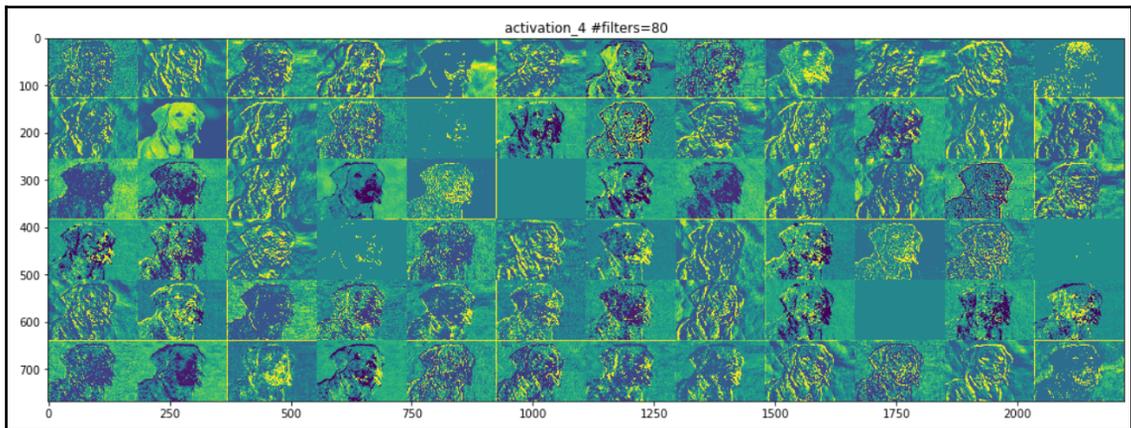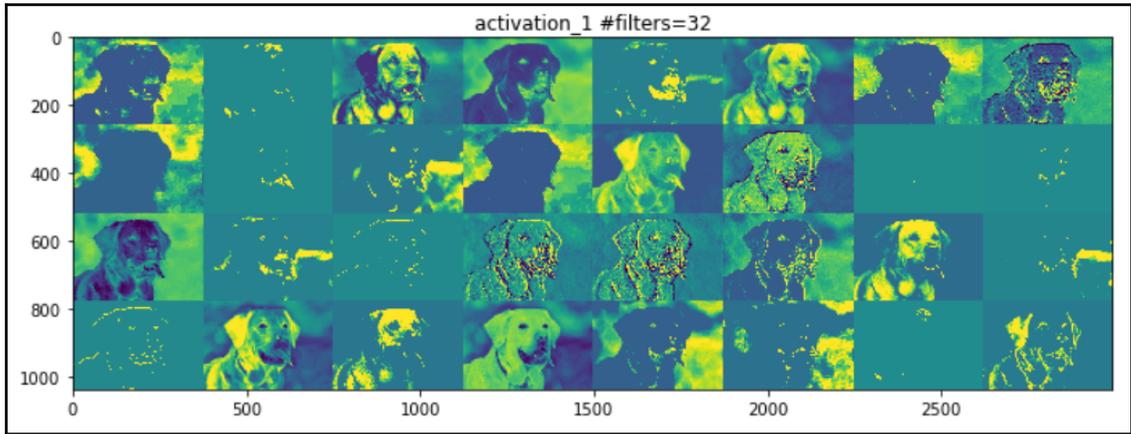
The first two preceding activation layers act as a collection of various edge detectors. These activations retain almost all the information present in the initial picture.

Let's look at the following screenshot, which shows a layer from the middle of the network. Here, it starts recognizing higher-level features like nose, eyes, tongue, mouth, and so on:
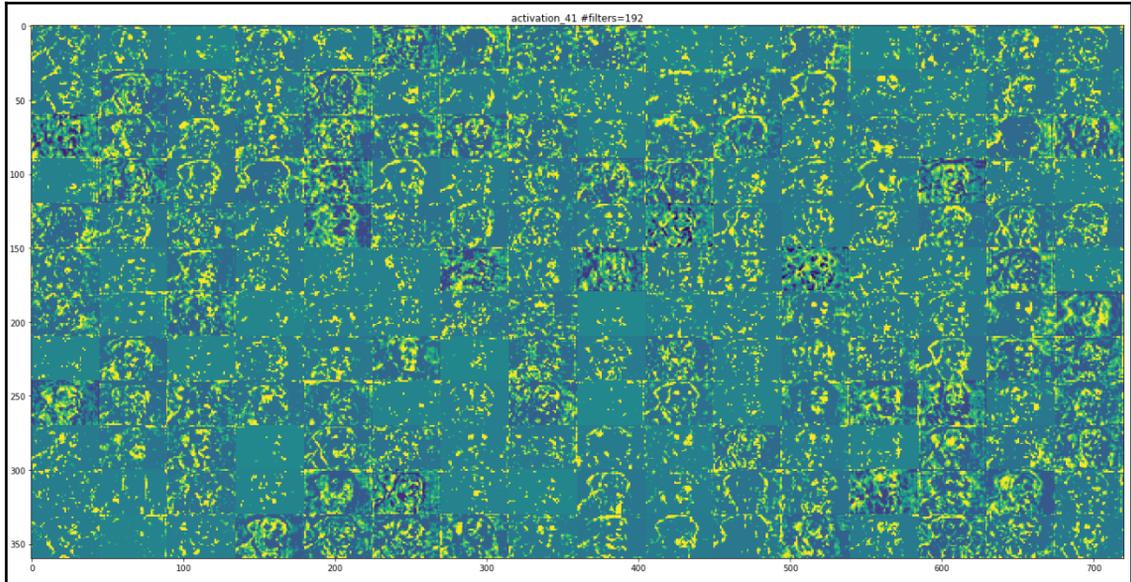
As we go higher up, the feature maps become less visually interpretable. The activations of layers higher up carry minimal information about the specific input being seen, and more information about the target class of the image, that is, in this case, a dog.

An alternative way to visualize the filters learned by InceptionV3 is to display the visual pattern for which each filter outputs maximum activation values. This can be done with gradient ascent in the input space. Basically, finding an input image that maximizes the activity of interest (activation of neurons in a layer) by carrying out an optimization using gradient ascent in the image space. The resulting input image would be one that the chosen filter is maximally responsive to.

Every activation layer has many feature maps. The following code demonstrates how we can extract a single feature map from the last activation layer. This activation value is actually out loss that we want to maximize:

```
layer_name = 'activation_94'
filter_index = 0
layer_output = model.get_layer(layer_name).output
loss = K.mean(layer_output[:, :, :, filter_index])
```

To calculate the gradient of the input image with respect to this `loss` function, we can use the `keras` backend gradient function as follows:

```
grads = K.gradients(loss, model.input)[0]
# We add 1e-5 before dividing so as to avoid accidentally dividing by
# 0.
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

So, given an activation layer and a starting input image that can be a random noise, we can apply gradient ascent using the gradient computation above to get the pattern the feature map represents. Following the `generate_pattern` function does the same. The output pattern is normalized so that we have feasible RGB values in the image matrix, and this is done by using the `deprocess_image` method. This following code is self-explanatory, and has inline comments to explain each line:

```
def generate_pattern(layer_name, filter_index, size=150):
    # Build a loss function that maximizes the activation
    # of the nth filter of the layer considered.
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index])
    # Compute the gradient of the input picture wrt this loss
    grads = K.gradients(loss, model.input)[0]
    # Normalization trick: we normalize the gradient
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
    # This function returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])
    # We start from a gray image with some noise
    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.
    # Run gradient ascent for 40 steps
    step = 1.
    for i in range(40):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step
        img = input_img_data[0]
    return deprocess_image(img)


def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1
    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)
```

```
# convert to RGB array
x *= 255
x = np.clip(x, 0, 255).astype('uint8')
return x
```

The following screenshot is the visualizations of some of the layers of filters. The first layers have various types of dot patterns:
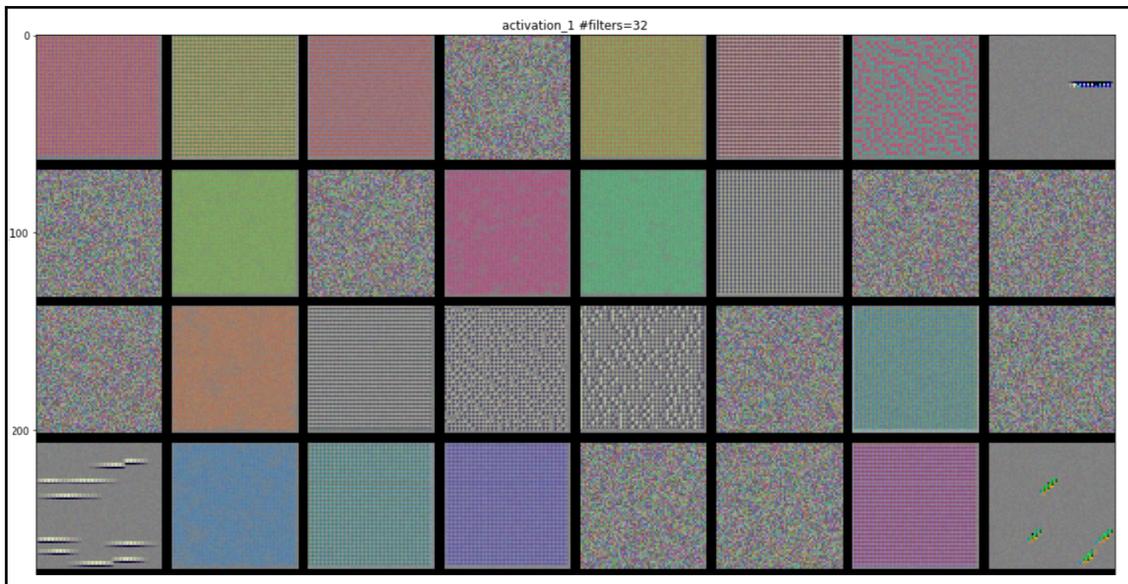
DeepDream

DeepDream is an artistic image modification technique that leverages the representations learned by deep CNN code *Inception*, named after the movie of the same name. We can take any input image and process it to generate trippy pictures, full of algorithmic pareidolia artifacts, bird feathers, dog-like faces, dog eyes—a by-product of the fact that the DeepDream convent was trained on ImageNet, where dog breeds and bird species are vastly over-represented.

The DeepDream algorithm is almost identical to the ConvNet filter
visualization technique using gradient ascent, except in a few differences as follows:

- In DeepDream, activation of entire layers is maximized, whereas in visualization
  only a specific filter is maximized, thus mixing together visualizations of large
  numbers of features maps
- We start not from a random noise input, but rather from an existing image; thus,
  the resulting visualizations will modify pre-existing visual patterns,
  distorting elements of the image in a somewhat artistic fashion
- The input images get processed at different scales (called **octaves**), which
  improves the quality of the visualizations

Let's now modify the visualization code in the previous section. First, we have to change
the loss function and the gradient calculation. The following is the code to do the same:

```
layer_name = 'activation_41'
activation = model.get_layer(layer_name).output

# We avoid border artifacts by only involving non-border pixels in the
#loss.
scaling = K.prod(K.cast(K.shape(activation), 'float32'))
loss = K.sum(K.square(activation[:, 2: -2, 2: -2, :])) / scaling

# This tensor holds our generated image
dream = model.input

# Compute the gradients of the dream with regard to the loss.
grads = K.gradients(loss, dream)[0]

# Normalize gradients.
grads /= K.maximum(K.mean(K.abs(grads)), 1e-7)

iterate_grad_ac_step = K.function([dream], [loss, grads])
```

The second change is in the input image, so we have to supply an input image on which we
want the DeepDream algorithm to run. The third change is, instead of applying gradient
accent on a single image, we create input images at various scales and apply gradient
accent, as shown in the following code:

```
num_octave = 4 # Number of scales at which to run gradient ascent
octave_scale = 1.4 # Size ratio between scales
iterations = 20 # Number of ascent steps per scale

# If our loss gets larger than 10,
# we will interrupt the gradient ascent process, to avoid ugly
# artifacts
```

```
max_loss = 20.

base_image_path = 'Path to Image You Want to Use'
# Load the image into a Numpy array
img = preprocess_image(base_image_path)
print(img.shape)
# We prepare a list of shape tuples
# defining the different scales at which we will run gradient ascent
original_shape = img.shape[1:3]
successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in
                   original_shape])
    successive_shapes.append(shape)

# Reverse list of shapes, so that they are in increasing order
successive_shapes = successive_shapes[::-1]

# Resize the Numpy array of the image to our smallest scale
original_img = np.copy(img)
shrunk_original_img = resize_img(img, successive_shapes[0])
print(successive_shapes)

#Example Octaves for image of shape (1318, 1977)
[(480, 720), (672, 1008), (941, 1412), (1318, 1977)]
```

The following code shows a few utility functions for the DeepDream algorithm. The function `deprocess_image` is basically an inverse operator of preprocess input for the InceptionV3 model:

```
import scipy

def deprocess_image(x):
    # Util function to convert a tensor into a valid image.
    if K.image_data_format() == 'channels_first':
        x = x.reshape((3, x.shape[2], x.shape[3]))
        x = x.transpose((1, 2, 0))
    else:
        x = x.reshape((x.shape[1], x.shape[2], 3))
    x /= 2.
    x += 0.5
    x *= 255.
    x = np.clip(x, 0, 255).astype('uint8')
    return x

def resize_img(img, size):
    img = np.copy(img)
    factors = (1,
```

```
                        float(size[0]) / img.shape[1],
                        float(size[1]) / img.shape[2],
                        1)
        return scipy.ndimage.zoom(img, factors, order=1)


    def save_img(img, fname):
        pil_img = deprocess_image(np.copy(img))
        scipy.misc. (fname, pil_img)
```

At each successive scale, from the smallest to the largest octaves, we run gradient ascent to maximize the loss we have previously defined, at that scale. After each gradiant ascent run, the resulting image is upscaled by 40%. At each upscaling step, some image details are lost; but we can restore it by adding the lost information, as we know the original image at that scale:

```
    MAX_ITRN = 20
    MAX_LOSS = 20
    learning_rate = 0.01

    for shape in successive_shapes:
        print('Processing image shape', shape)
        img = resize_img(img, shape)
        img = gradient_ascent(img,
                              iterations=MAX_ITRN,
                              step=learning_rate,
                              max_loss=MAX_LOSS)
        upscaled_shrunk_original_img = resize_img(shrunk_original_img,
                                                  shape)
        same_size_original = resize_img(original_img, shape)
        lost_detail = same_size_original - upscaled_shrunk_original_img
        print('adding lost details', lost_detail.shape)
        img += lost_detail
        shrunk_original_img = resize_img(original_img, shape)
        save_img(img, fname='dream_at_scale_' + str(shape) + '.png')

    save_img(img, fname='final_dream.png')
```

Examples

Following are a few examples of the DeepDream output:

- Running gradient accent on activation Layer 41. This is the same layer we visualized before, with a dog image input. In the following photo, you can see some animals emerging out of the clouds and blue sky:

- Running gradient accent on activation Layer 45. In the following photo, you can see some dog-like animal faces emerging out of the mountains:
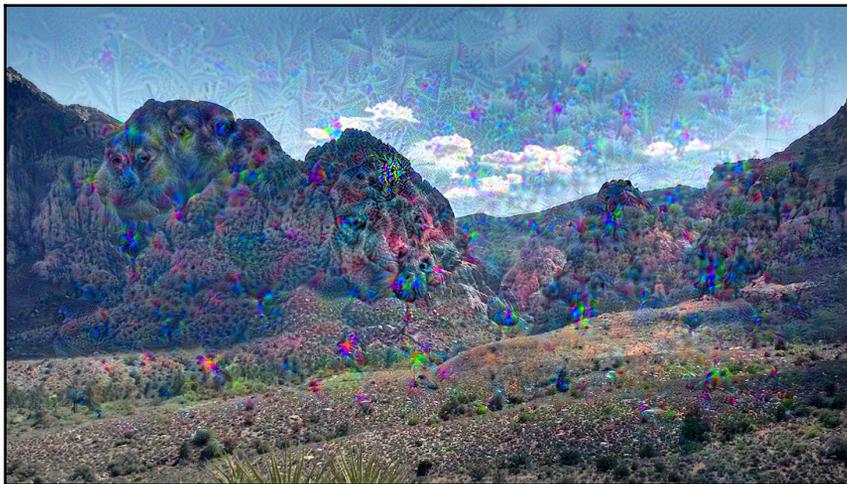
- Running gradient accent on activation Layer 50. In the following photo, you can see some particular leaf-like pattern dream over the blue sky with clouds:
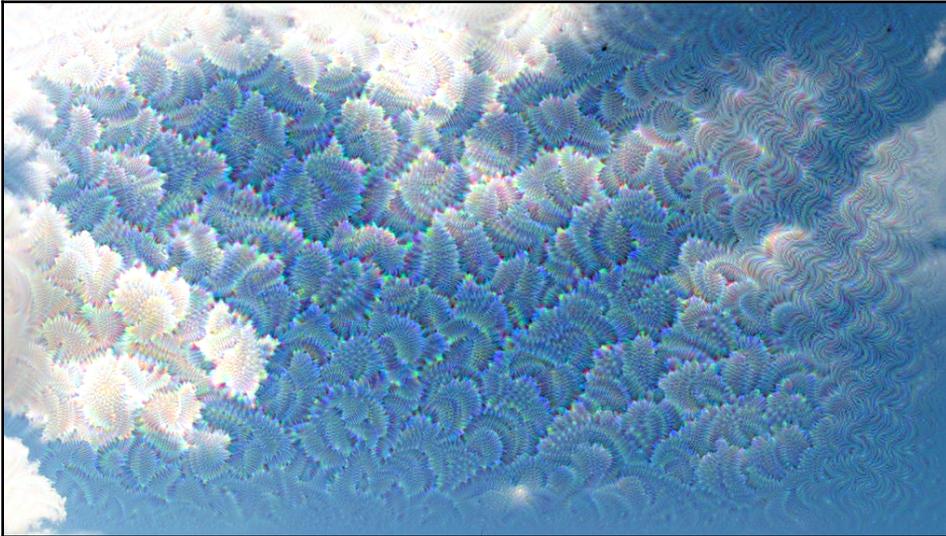
The original images for generating these dreams are shared as a part of the code repository.

Summary

In this chapter, we learned about algorithmic pareidolia in computer vision. We have explained how CNN models can be interpreted by various visualization techniques, like forward pass-based activation visualization, gradient ascent-based filter visualization. Finally, we introduced the DeepDream algorithm, which again, is a slight modification of the gradient ascent-based visualization technique. The DeepDream algorithm is an example of transfer learning being applied to a computer vision or an image-processing task.

We will see more similar applications in the next chapter, which will be focusing on style transfer.

<h1>10
Style Transfer</h1>

Paintings require a special skill only a few have mastered. Paintings present a complex interplay of content and style. Photographs, on the other hand, are a combination of perspectives and light. When the two are combined, the results are spectacular and surprising. This process is called **artistic style transfer**. The following is an example of this, where the input image is of Neckarfront in Tübingen, Germany, and the style image is of the famous painting *The Starry Night*, by Vincent van Gogh. Interesting, isn't it? Take a look at the following images:

Left image: The original photograph depicting the Neckarfront in Tübingen, Germany. Right image: The painting (inset: The Starry Night by Vincent van Gogh) that provided the style for the respective generated image. Source: *A Neural Algorithm of Artistic Style*, Gatys et al. (arXiv:1508.06576v2)

If you look at the preceding images closely, the painting-styled image on the right seems to have picked up the content from the photograph on the left. The style, colors, and stroke patterns from the painting generated the final outcome. This fascinating outcome is a result of a transfer learning algorithm presented in the paper, *A Neural Algorithm for Artistic Style*, by Gatys et al. (`https://arxiv.org/abs/1508.06576`). We will discuss the intricacies of this paper from an implementation point of view, and see how we can perform this technique ourselves.

In this chapter, we will be focusing on leveraging deep learning along with transfer learning for building a neural style transfer system. The key areas of focus in the chapter include the following:

- Understanding neural style transfer
- Image preprocessing methodology
- Building loss functions
- Constructing a custom optimizer
- Style transfer in action

We will be covering theoretical concepts around neural style transfer, loss functions and optimization. Besides this, we will use a hands-on approach to implement our own neural style transfer model. The code for this chapter is available for quick reference in the Chapter 10 folder in the GitHub repository at `https://github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter.

Understanding neural style transfer

Neural style transfer is the process of applying the **style** of a reference image to a specific target image, such that the original **content** of the target image remains unchanged. Here, style is defined as colors, patterns, and textures present in the reference image, while content is defined as the overall structure and higher-level components of the image.

Here, the main objective is to retain the content of the original target image, while superimposing or adopting the style of the reference image on the target image. To define this concept mathematically, consider three images: the original content (denoted as **c**), the reference style (denoted as **s**), and the generated image (denoted as **g**). We would need a way to measure how different images c and g might be in terms of their content. Also, the output image should have less difference compared to the style image, in terms of styling features of the output. Formally, the objective function for neural style transfer can be formulated as follows:

$$L_{\text{style transfer}} = \text{argmin}_g \, \alpha L_{\text{content}}(c, g) + \beta L_{\text{style}}(s, g)$$

Here, α and β are weights used to control the impact of content and style components on the overall loss. This depiction can be simplified further and represented as follows:

$$loss = dist(content(I_c) - content(I_g)) + dist(style(I_s) - style(I_g))$$

Here, we can define the following components from the preceding formula:

- `dist` is a norm function; for example, the L_2 norm distance
- `style(...)` is a function to compute representations of style for the reference style and generated images
- `content(...)` is a function to compute representations of content for the original content and generated images
- I_c, I_s, and I_g are the content, style, and generated images respectively

Thus, minimizing this loss causes $style(I_g)$ to be close to $style(I_s)$, and also $content(I_g)$ to be close to $content(I_c)$. This helps us in achieving the necessary stipulations for effective style transfer. The loss function we will try to minimize consists of three parts; namely, the **content loss**, the **style loss**, and the **total variation loss**, which we will be talking about soon. The key idea or objective is to retain the *content* of the original target image, while superimposing or adopting the *style* of the reference image on the target image. Also, you should remember the following in the context of neural style transfer:

- **Style** can be defined as color palettes, specific patterns, and textures that present in the reference image
- **Content** can be defined as the overall structure and higher-level components of the original target image

By now, we know that the true power of deep learning for computer vision lies in leveraging models like deep **Convolutional Neural Network** (**CNN**) models, which can be used to extract the right image representations when building these loss functions. In this chapter, we will be using the principles of transfer learning in building our system for neural style transfer to extract the best features. We have already discussed pretrained models for computer vision-related tasks in previous chapters. We will be using the popular VGG-16 model again in this chapter as a feature extractor. The main steps for performing neural style transfer are depicted as follows:

- Leverage VGG-16 to help compute layer activations for the style, content, and generated image
- Use these activations to define the specific loss functions mentioned earlier
- Finally, use gradient descent to minimize the overall loss

If you want to dive deeper into the core principles and theoretical concepts behind neural style transfer, we recommend you read up on the following papers:

- *A Neural Algorithm of Artistic Style*, by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge (`https://arxiv.org/abs/1508.06576`)
- *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*, by Justin Johnson, Alexandre Alahi, and Li Fei-Fei (`https://arxiv.org/abs/1603.08155`)

Image preprocessing methodology

The first and foremost step towards implementation of such a network is to preprocess the data or images in this case. The following code snippet shows some quick utilities to preprocess and postprocess images for size and channel adjustments:

```python
import numpy as np
from keras.applications import vgg16
from keras.preprocessing.image import load_img, img_to_array


def preprocess_image(image_path, height=None, width=None):
    height = 400 if not height else height
    width = width if width else int(width * height / height)
    img = load_img(image_path, target_size=(height, width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg16.preprocess_input(img)
    return img

def deprocess_image(x):
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

As we would be writing custom loss functions and manipulation routines, we would need to define certain placeholders. Remember that `keras` is a high-level library that utilizes tensor manipulation backends (like `tensorflow`, `theano`, and `CNTK`) to perform the heavy lifting. Thus, these placeholders provide high-level abstractions to work with the underlying tensor object. The following snippet prepares placeholders for style, content, and generated images, along with the input tensor for the neural network:

```
from keras import backend as K

# This is the path to the image you want to transform.
TARGET_IMG = 'lotr.jpg'
# This is the path to the style image.
REFERENCE_STYLE_IMG = 'pattern1.jpg'

width, height = load_img(TARGET_IMG).size
img_height = 480
img_width = int(width * img_height / height)


target_image = K.constant(preprocess_image(TARGET_IMG,
                          height=img_height,
                          width=img_width))
style_image = K.constant(preprocess_image(REFERENCE_STYLE_IMG,
                          height=img_height,
                          width=img_width))

# Placeholder for our generated image
generated_image = K.placeholder((1, img_height, img_width, 3))

# Combine the 3 images into a single batch
input_tensor = K.concatenate([target_image,
                              style_image,
                              generated_image], axis=0)
```

We will load the pretrained VGG-16 model as we did in previous chapters; that is, without the top fully-connected layers. The only difference here is that we would be providing the size dimensions of the input tensor for the model input. The following snippet helps us build the pretrained model:

```
model = vgg16.VGG16(input_tensor=input_tensor,
                    weights='imagenet',
                    include_top=False)
```

Building loss functions

As discussed in the background subsection, the problem with neural style transfer revolves around loss functions of content and style. In this subsection, we will discuss and define the required loss functions.

Content loss

In any CNN-based model, activations from top layers contain more global and abstract information (for example, high-level structures like a face), and bottom layers will contain local information (for example, low-level structures like eyes, nose, edges, and corners) about the image. We would want to leverage the top layers of a CNN for capturing the right representations for the content of an image. Hence, for the content loss, considering we will be using the pretrained VGG-16 model, we can define our loss function as the L2 norm (scaled and squared Euclidean distance) between the activations of a top layer (giving feature representations) computed over the target image, and the activations of the same layer computed over the generated image. Assuming we usually get feature representations relevant to the content of images from the top layers of a CNN, the generated image is expected to look similar to the base target image. The following snippet shows the function to compute the content loss:

```
def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

Style loss

The original paper on neural style transfer, *A Neural Algorithm of Artistic Style,* by Gatys et al. (https://arxiv.org/abs/1508.06576), leverages multiple convolutional layers in the CNN (instead of one) to extract meaningful patterns and representations capturing information pertaining to appearance or style from the reference style image across all spatial scales, irrespective of the image content. The style representation computes correlations between the different features in different layers of the CNN.

Staying true to the original paper, we will be leveraging the **Gram matrix** and computing the same over the feature representations generated by the convolution layers. The Gram matrix computes the inner product between the feature maps produced in any given conv layer. The inner product's terms are proportional to the co-variances of corresponding feature sets, and hence, captures patterns of correlations between the features of a layer that tend to activate together. These feature correlations help capture relevant aggregate statistics of the patterns of a particular spatial scale, which correspond to the style, texture, and appearance, and not the components and objects present in an image.

The style loss is thus defined as the scaled and squared Frobenius norm (Euclidean norm on a matrix) of the difference between the Gram matrices of the reference style and the generated images. Minimizing this loss helps ensure that the textures found at different spatial scales in the reference style image will be similar in the generated image. Thus, the following snippet defines a style loss function based on a Gram matrix calculation:

```
def style_loss(style, combination, height, width):
    def build_gram_matrix(x):
        features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
        gram_matrix = K.dot(features, K.transpose(features))
        return gram_matrix

    S = build_gram_matrix(style)
    C = build_gram_matrix(combination)
    channels = 3
    size = height * width
    return K.sum(K.square(S - C))/(4. * (channels ** 2) * (size ** 2))
```

Total variation loss

It was observed that optimization to reduce only the style and content losses led to highly pixelated and noisy outputs. To cover the same, total variation loss was introduced. The **total variation loss** is analogous to *regularization* loss. This is introduced for ensuring spatial continuity and smoothness in the generated image to avoid noisy and overly pixelated results. The same is defined in the function as follows:

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, 1:, :img_width
        - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] - x[:, :img_height -
        1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

Overall loss function

Having defined the components of the overall loss function for neural style transfer, the next step is to stitch together these building blocks. Since content and style information is captured by the CNNs at different depths in the network, we need to apply and calculate loss at appropriate layers for each type of loss. We will be taking the conv layers one to five for the style loss and setting appropriate weights for each layer.

Here is the code snippet to build the overall loss function:

```
# weights for the weighted average loss function
content_weight = 0.05
total_variation_weight = 1e-4


content_layer = 'block4_conv2'
style_layers =  ['block1_conv2', 'block2_conv2',
                 'block3_conv3','block4_conv3', 'block5_conv3']
style_weights = [0.1, 0.15, 0.2, 0.25, 0.3]

# initialize total loss
loss = K.variable(0.)

# add content loss
layer_features = layers[content_layer]
target_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss += content_weight * content_loss(target_image_features,
                                      combination_features)

# add style loss
for layer_name, sw in zip(style_layers, style_weights):
    layer_features = layers[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features,
                    height=img_height, width=img_width)
    loss += (sl*sw)

# add total variation loss
loss += total_variation_weight * total_variation_loss(generated_image)
```

Constructing a custom optimizer

The objective is to iteratively minimize the overall loss with the help of an optimization algorithm. In the paper by Gatys et al., optimization was done using the L-BFGS algorithm, which is an optimization algorithm based on Quasi-Newton methods, which are popularly used for solving non-linear optimization problems and parameter estimation. This method usually converges faster than standard gradient descent.

SciPy has an implementation available in `scipy.optimize.fmin_l_bfgs_b();` however, limitations include the function being applicable only to flat one-dimensional vectors, unlike three-dimensional image matrices that we are dealing with, and the fact that the value of loss function and gradients need to be passed as two separate functions. We build an `Evaluator` class based on patterns, followed by `keras` creator François Chollet, to compute both loss and gradient values in one pass instead of independent and separate computations. This will return the loss value when called the first time, and will cache the gradients for the next call. Thus, it would be more efficient than computing both independently. The following snippet defines the `Evaluator` class:

```python
class Evaluator(object):

    def __init__(self, height=None, width=None):
        self.loss_value = None
        self.grads_values = None
        self.height = height
        self.width = width

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, self.height, self.width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator(height=img_height, width=img_width)
```

Style transfer in action

The final piece of the puzzle is to use all the building blocks and perform style transfer in action! The art/style and content images are available from the data directory for reference. The following snippet outlines how loss and gradients are evaluated. We also write back outputs after regular intervals/iterations (5, 10, and so on) to understand how the process of neural style transfer transforms the images in consideration after a certain number of iterations as depicted in the following snippet:

```python
from scipy.optimize import fmin_l_bfgs_b
from scipy.misc import imsave
from imageio import imwrite
import time

result_prefix = 'st_res_'+TARGET_IMG.split('.')[0]
iterations = 20

# Run scipy-based optimization (L-BFGS) over the pixels of the
# generated image
# so as to minimize the neural style loss.
# This is our initial state: the target image.
# Note that `scipy.optimize.fmin_l_bfgs_b` can only process flat
# vectors.
x = preprocess_image(TARGET_IMG, height=img_height, width=img_width)
x = x.flatten()

for i in range(iterations):
    print('Start of iteration', (i+1))
    start_time = time.time()
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x,
                                     fprime=evaluator.grads, maxfun=20)
    print('Current loss value:', min_val)
    if (i+1) % 5 == 0 or i == 0:
        # Save current generated image only every 5 iterations
        img = x.copy().reshape((img_height, img_width, 3))
        img = deprocess_image(img)
        fname = result_prefix + '_iter%d.png' %(i+1)
        imwrite(fname, img)
        print('Image saved as', fname)
    end_time = time.time()
    print('Iteration %d completed in %ds' % (i+1, end_time - start_time))
```

It must be pretty evident by now that neural style transfer is a computationally expensive task. For the set of images in consideration, each iteration took between 500-1,000 seconds on an Intel i5 CPU with 8GB RAM (much faster on i7 or Xeon processors though!). The following code snippet shows the speedup we are getting using GPUs on a p2.x instance on AWS, where each iteration takes a mere 25 seconds! The following code snippet also shows the output of some of the iterations. We print the loss and time taken for each iteration, and save the generated image after every five iterations:

```
Start of iteration 1
Current loss value: 10028529000.0
Image saved as st_res_lotr_iter1.png
Iteration 1 completed in 28s
Start of iteration 2
Current loss value: 5671338500.0
Iteration 2 completed in 24s
Start of iteration 3
Current loss value: 4681865700.0
Iteration 3 completed in 25s
Start of iteration 4
Current loss value: 4249350400.0
.
.
.
Start of iteration 20
Current loss value: 3458219000.0
Image saved as st_res_lotr_iter20.png
Iteration 20 completed in 25s
```

Now you'll learn how the neural style transfer model has performed style transfer for the content images in consideration. Remember that we performed checkpoint outputs after certain iterations for every pair of style and content images. We utilize `matplotlib` and `skimage` to load and understand the style transfer magic performed by our system!

We have used the following image from the very popular *Lord of the Rings* movie as our content image, and a nice floral pattern-based artwork as our style image:
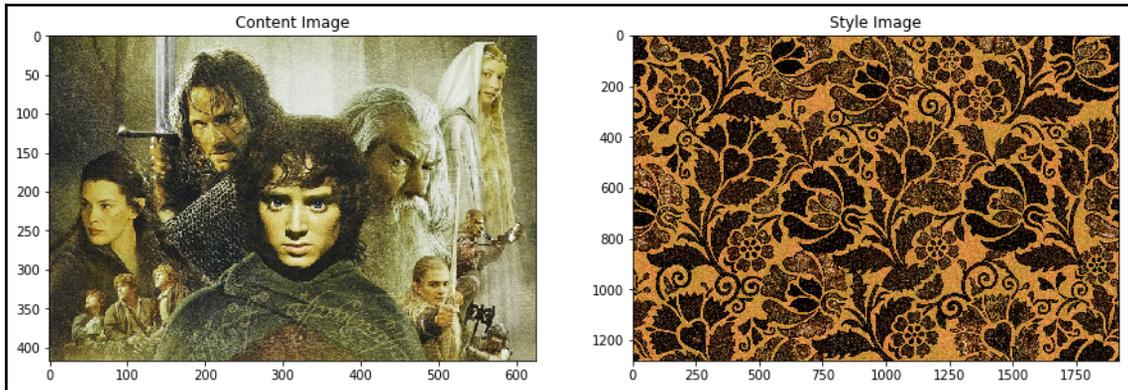
In the following code snippet, we are loading the generated styled images after various iterations:

```
from skimage import io
from glob import glob
from matplotlib import pyplot as plt

%matplotlib inline
content_image = io.imread('lotr.jpg')
style_image = io.imread('pattern1.jpg')

iter1 = io.imread('st_res_lotr_iter1.png')
iter5 = io.imread('st_res_lotr_iter5.png')
iter10 = io.imread('st_res_lotr_iter10.png')
iter15 = io.imread('st_res_lotr_iter15.png')
iter20 = io.imread('st_res_lotr_iter20.png')
fig = plt.figure(figsize = (15, 15))
ax1 = fig.add_subplot(6,3, 1)
ax1.imshow(content_image)
t1 = ax1.set_title('Original')

gen_images = [iter1,iter5, iter10, iter15, iter20]

for i, img in enumerate(gen_images):
    ax1 = fig.add_subplot(6,3,i+1)
    ax1.imshow(content_image)
    t1 = ax1.set_title('Iteration {}'.format(i+5))
plt.tight_layout()
fig.subplots_adjust(top=0.95)
t = fig.suptitle('LOTR Scene after Style Transfer')
```

Following is the output showcasing the original image and the generated styled images after every five iterations:
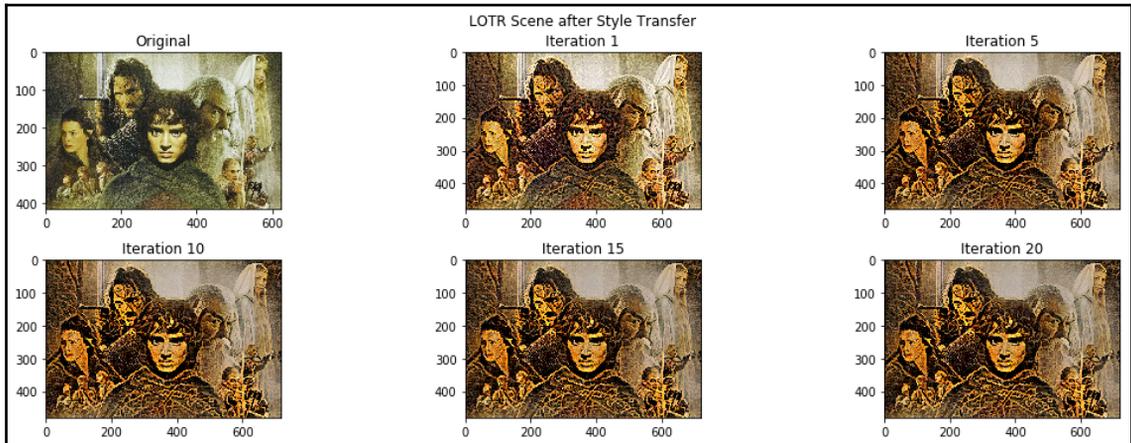
Following is the final styled image at a higher resolution. You can clearly see how the floral pattern textures and styles have slowly started propagating in the original *Lord of the Rings* movie image, giving it a nice vintage look:
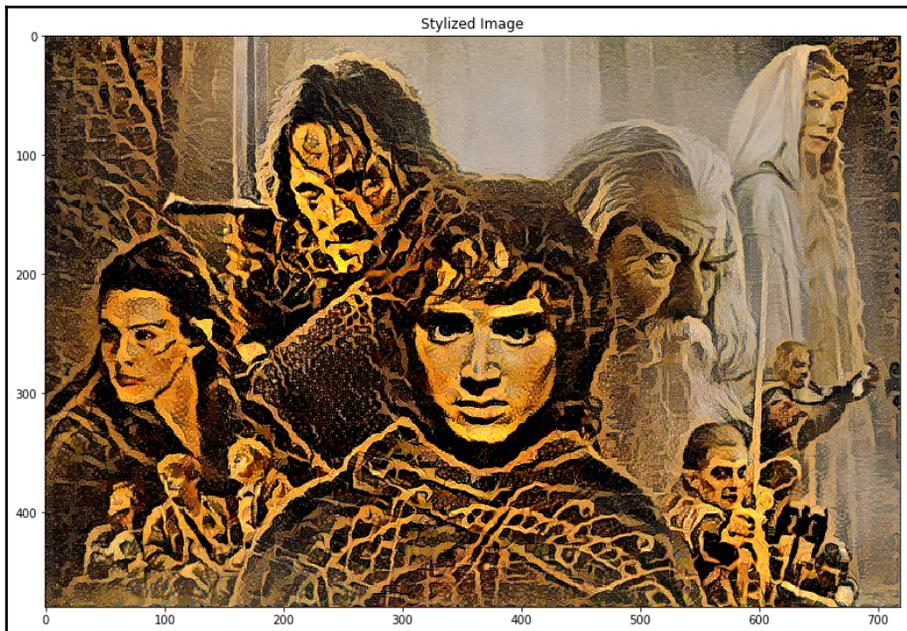
Let's take one more style transfer example. The following figure contains our content image, the famous fictional city of Wakanda from Black Panther. The style image is the very popular painting *The Starry Night*, by Van Gogh! We will be using these as input images in our style transfer system:
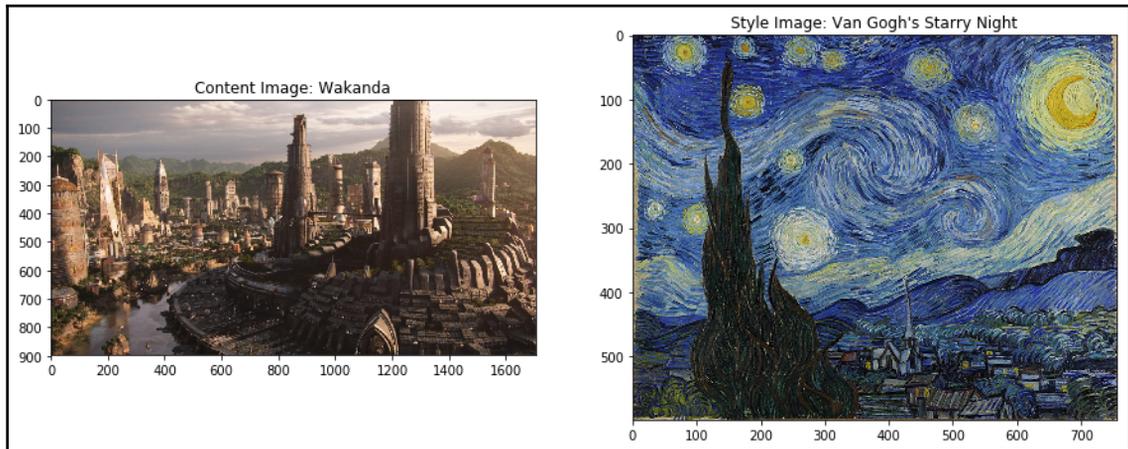
Following is the final styled image is depicted in the following image in high resolution. You can clearly see how the textures, edges, colors, and patterns from the style painting have propagated into the city content image:

The sky and buildings have taken a form very similar to what you can observe in the painting, and yet the overall structure in the content image has been retained. Fascinating, isn't it? Try this out now with your own images of interest!

Summary

This chapter presented a very novel technique in the deep learning landscape, leveraging the power of deep learning to create art! Indeed, data science is an art as well as a science of using data in the right way, and innovation is something that drives that. We covered the core concepts of neural style transfer, how to represent and formulate the problem using an effective loss function, and how to leverage the power of transfer learning and pretrained models like VGG-16 to extract the right feature representations.

The field of computer vision is ever evolving, and deep learning coupled with transfer learning has opened up doors for innovation and building novel applications. The examples in this chapter should help you appreciate the vast scope of novelty in this field, and enable you to get out there and try new techniques, models, and methods to build systems like neural style transfer! Coming up, more interesting and complex case-studies around image captioning and colorization. Stay tuned!

11
Automated Image Caption Generator

In the previous chapters, we looked at several case studies pertaining to applying transfer learning on problems in computer vision as well as **natural language processing** (**NLP**). However, these were problems in their own specific domains. In this chapter, we will be focusing on building an intelligent system that is a combination of these two popular domains—computer vision and NLP. To be more specific, we will be focusing on building an object-recognition system coupled with machine translation to build an automated image-caption generator.

The idea of image captioning is not something new. Typically, any image present in diverse sources of media, such as books, papers, or social media, usually needs to be captioned with a proper text description for better meaning and context. What makes this task tough is that an image caption is typically free-flowing natural language consisting of one or more sentences. Thus, due to the unstructured nature of text data for image captions, it is not a traditional image-classification problem.

Image captioning can be solved by using a combination of leveraging pretrained models that are experts in the computer vision domain, such as **Visual Geometry Group** (**VGG**) or Inception, and sequence models, such as **recurrent neural networks** (**RNNs**) or **Long Short-Term Memory** (**LSTM**), to generate sequences of words to form a proper image caption. In this chapter, we will explore an interesting approach to building an automated image captioning or scene-recognition system.

We will be covering the following major aspects for building this system, which is powered by deep learning and transfer learning:

- Understanding image captioning
- Formulating our objective
- Understanding the data
- Approach to automated image captioning
- Image feature extraction with transfer learning
- Building a vocabulary for our captions
- Building an image caption dataset generator
- Building our image language encoder-decoder deep learning model
- Training our image captioning deep learning model
- Automated image captioning in action

We will cover essential concepts from both computer vision and NLP to build our automated image caption generator. We will dive deep into a suitable deep learning architecture coupled with transfer learning to implement this system on top of a popular and easily-available image dataset. We will also showcase how to build and test our automated image caption generator on new photos and scenes. The code for this chapter is available for quick reference in the `Chapter 11` folder in the GitHub repository at `https:/ /github.com/dipanjanS/hands-on-transfer-learning-with-python` which you can refer to as needed to follow along with the chapter. We will also be posting some bonus examples there.

Understanding image captioning

By now, you should understand the significance and meaning of image captioning. This task can be simply defined as writing and recording a free-flowing and natural text description for any image. It is usually used to describe various scenes or events in images. This is also popularly termed **scene recognition**. Let's look at the following example:

Looking at this scene, what could be a suitable caption or description? The following are all valid descriptions of the scene:

- A motocross rider is on a dirt hill
- A guy on a bicycle midair above a hill
- A dirt bike rider is moving fast down a dirt path
- A biker riding a black motorbike in midair

You can see that all of these captions are valid and are similar yet use different words to convey the same meaning. This is why generating image captions automatically is not an easy task.

In fact, the exact same thing is mentioned by a popular paper *Show and Tell: A Neural Image Caption Generator*, Vinyals and its co-authors, 2015 (`https://arxiv.org/abs/1411.4555`) describing image captioning, from which we draw our inspiration for building this system:

Automatically describing the content of an image is a fundamental problem in artificial intelligence that connects computer vision and natural language processing.

For a human being, just glancing at a photo or image for a few seconds is enough to generate a natural language-based caption. However, enabling **artificial intelligence** (**AI**) to perform this task is extremely challenging given that the majority of computer vision problems have been focused on recognition and classification problems. Here are some of the major tasks with regard to core computer vision problems in terms of increasing complexity:

- **Image classification and recognition**: This involves a classic supervised learning problem where the main objective is to assign an image to a particular category based on several predefined class categories (often known as **class labels**). The popular ImageNet competition is one such task.
- **Image annotation**: A slightly more complex task, where we try to annotate an image with descriptions of various entities in the image. Typically, this involves categories or even natural language-based textual descriptions for specific sections or regions in the image.
- **Image captioning or scene recognition**: Another complex task where we try to describe an image with an accurate natural language-based textual description. This is the main area of focus for this chapter.

The task of image captioning is not a new thing. There have been several prior approaches of leveraging techniques such as stitching together text descriptions of individual entities from an image to form a description or even using template-based text-generation techniques. However, using deep learning is a more robust and efficient approach for this task.

Formulating our objective

The main objective of our real-world case study is image captioning or scene recognition. This is a supervised learning problem to an extent, but not a traditional classification problem. Here, we will be working on an image dataset, known as `Flickr8K`, with samples of images or scenes and corresponding natural language captions describing them. The idea is to build a system that can learn from these images and start captioning images automatically.

As I mentioned earlier, a traditional image classification system typically classifies or categorizes images into predefined classes. We have already built such a system in previous chapters. However, the output from an image captioning system is generally a sequence of words forming a textual description in natural language; this makes it more difficult than a traditional supervised classification system.

The nature of our model training will still be supervised since we will have to build it based on training image data and their corresponding caption descriptions. However, the approach toward building our model will be slightly different. We will be leveraging concepts from transfer learning and deep learning to build this system as usual. To be more specific, we will be using a combination of **Deep Convolutional Neural Networks** (**DCNNs**) and sequential models.

Understanding the data

Let's take a look at the data we will be using to build our model. To keep things simple, we will be using the `Flickr8K` dataset. This dataset includes images obtained from Flickr, a popular image sharing website. To download the dataset, you can request it by filling in a form at `https://forms.illinois.edu/sec/1713398` from the Department of Computer Science, University of Illinois, and you should get the download link in your email.

To check out the details pertaining to each image, you can refer to their website, `http://nlp.cs.illinois.edu/HockenmaierGroup/8k-pictures.html`, which talks about each image, its source, and five text-based captions for each image. In general, any sample image would have several captions similar to the following:

You can clearly see the image and its corresponding captions. It is quite evident that all the captions try to describe the same image or scene but it might focus on specific and different aspects of the image, making this a hard task to automate. We also recommend readers to check out the paper, *Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics*, Micah Hodosh et al., IJCAI 2015 (`https://pdfs.semanticscholar.org/f126/ec304cdad464f6248ac7f73a186ca26db526.pdf`).

There will be two files that you obtain when you click on the download link:

- `Flickr8k_Dataset.zip`: A 1 GB ZIP archive of all the raw images and photos
- `Flickr8k_text.zip`: A 3 MB ZIP archive of all natural-language textual descriptions for photographs, which are the captions

The `Flickr_8k.devImages.txt`, `lickr_8k.trainImages.txt`, and `Flickr_8k.testImages.txt` files consist of the filenames for 6,000, 1,000, and 1,000 images, respectively. We will be combining the `dev` and `train` images to build a training dataset of 7,000 images and use a test dataset of 1,000 images for evaluation. Each image has five different yet similar captions and is available in the `Flickr8k.token.txt` file.

Approach to automated image captioning

We will now discuss our approach toward building an automated image captioning system. As I mentioned previously, our approach will be leveraging a deep neural network-based approach coupled with transfer learning toward image captioning. This is inspired from the popular paper, *Show and Tell: A Neural Image Caption Generator*, Oriol Vinyals et al., 2015 (`https://arxiv.org/abs/1411.4555`). We will be talking about a conceptual overview of our approach followed by translating it to a practical approach that will be used to build our automated image captioning system. Let's get started!

Conceptual approach

A successful image captioning system needs a way to translate a given image into a sequence of words. For extracting the right and relevant features from images, we can leverage a DCNN and, coupled with a recurrent neural network model, such as RNNs or LSTMs, we can build a hybrid generative model to start generating sequences of words as a caption, given a source image.

Thus, conceptually, the idea is to build a single hybrid model that can take a source image I as input, and can be trained to maximize the likelihood, $P(S|I)$, such that S is the output of a sequence of words, which is our target output, and can be represented by $S = \{S_1, S_2, ..., S_n\}$, such that each word S_w comes from a given dictionary, which is our vocabulary. This caption S should be able to give a decent description for the input image.

Neural machine translation is an excellent inspiration for building such a system. Typically used in language models for language translation, the model architecture involves an encoder-decoder architecture built using RNNs or LSTMs. Typically, the encoder involves a LSTM model that reads an input sentence from the source language and transforms it into a dense fixed-length vector. This is then used as the initial hidden state of the decoder LSTM model, which ultimately generates an output sentence in the target language.

For image captioning, we will leverage a similar strategy, where our encoder that processes the input will leverage a DCNN model since our source data is images. By now, we have already seen the strength of CNN-based models for effective and rich feature extraction from images. Thus, source image data will be converted to dense numeric fixed-length vectors. Typically, a pretrained model leveraging a transfer learning approach will be the most effective here. This vector will serve as an input to our decoder LSTM model, which will generate a caption description as a sequence of words. Taking inspiration from the original paper, the objective to be maximized can be represented mathematically, as follows:

$$\Theta^* = arg \max_{\Theta} \sum_{I,S} log \, p(S, I)$$

Here, Θ signifies the model parameters, I signifies the input image, and S is its corresponding caption description consisting of a sequence of words. Considering a caption description of length N, indicating a total of N words, we can model the joint probability over $\{S_0, S_1, ..., S_N\}$ using the chain rule, as follows:

$$log \, p(S, I) = \sum_{t=0}^{N} log \, p(S_t | I, S_0, S_1, \ldots, S_{t-1})$$

Thus, during model training, we have a pair of (I, S) image captions as inputs and the idea is to optimize the sum of the log probabilities for the preceding equation over the entire training data using an efficient algorithm, such as stochastic gradient descent. Considering the sequence of terms in the RHS of the preceding equation, an RNN-based model is the apt choice, such that the variable number of words sequentially until *t-1* is expressed by memory state h_t. This is updated at every step, as follows, based on the previous *t-1* states and the input pairs (images and next word) x_t using a non-linear function `f(...)`:

$$h_{t+1} = f(h_t, x_t)$$

Typically, x_t represents our image features and words, which are our inputs. For image features, we leverage DCNNs as we mentioned before. For the function *f*, we choose to use LSTMs since they are very effective in dealing with problems like vanishing and exploring gradients, which we have discussed in the initial chapters in this book. Considering a brief refresher for an LSTM memory block, let us refer to the following diagram from the *Show and Tell* research paper:
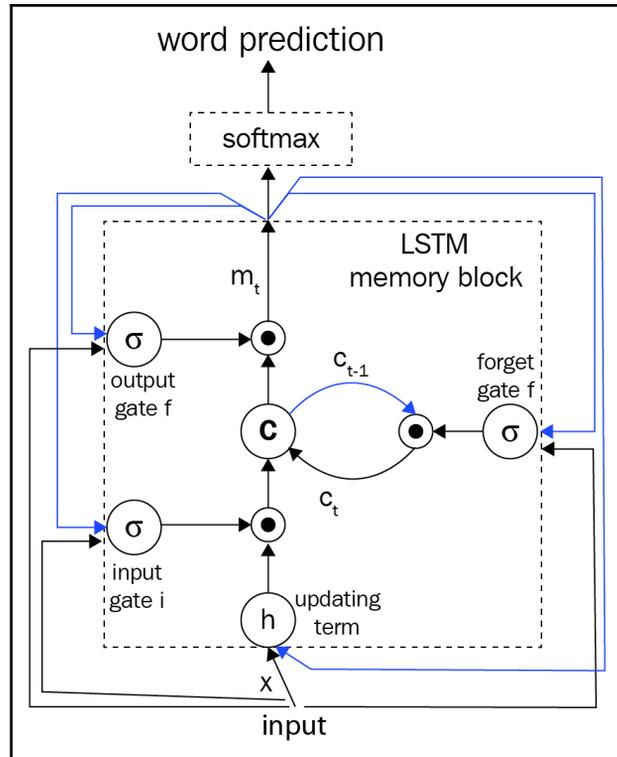
The memory block contains the LSTM cell **c**, which is controlled by the input, output, and forget gates. Cell *c* will encode knowledge at every time-step based on inputs until the previous time-steps. The three gates are layers that can be applied multiplicatively to either keep or reject a value from the gated layer if the gate is *1* or *0*. The recurrent connections are shown in blue in the preceding diagram. We generally have multiple LSTMs in the model and the output m_{t-1} of the LSTM at time *t -1* is fed to the next LSTM at time *t*. Thus, this output m_{t-1} at time *t-1* is fed back to the memory block at time *t* using the three gates we discussed earlier. The actual cell value is also fed back using the forget gate. The memory output at time *t*, which is m_t is typically fed to the softmax to predict the next word.

This is usually obtained from the output gate o_t and the current cell state c_t. Some of these definitions and operations are depicted in the following diagram with the necessary equations:

$$
\begin{aligned}
i_t &= \sigma(W_{ix}x_t + W_{im}m_{t-1}) \\
f_t &= \sigma(W_{fx}x_t + W_{fm}m_{t-1}) \\
o_t &= \sigma(W_{ox}x_t + W_{om}m_{t-1}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot h(W_{cx}x_t + W_{cm}m_{t-1}) \\
m_t &= o_t \odot c_t \\
p_{t+1} &= \text{Softmax}(m_t)
\end{aligned}
$$

Here, $\odot$ is the product operator used especially with current gate states and values. The W matrices are the trainable parameters in the network. These gates help in dealing with problems such as exploding and vanishing gradients. The non-linearity in the network is introduced by our regular sigmoid σ and hyperbolic-tangent h functions. As we discussed earlier, the memory output m_t is fed to the softmax to predict the next word, where the output is a probability distribution over all the words.

Thus, armed with this knowledge, you can consider that the LSTM-based sequence model needs to be combined with a necessary word embedding layer and the CNN-based model that produces dense features from the source image. Thus, the LSTM model's objective is to predict each word of the caption text, based on all the previous words predicted and also based on the input image, which is defined by our previous equation of $p(S_t|\ I, S_0, S_1, ..., S_{t-1})$. To simplify the recurrent connections in the LSTM, we can represent it in its unrolled form where we represent a series of LSTMs, which share the same parameters as depicted in the following diagram:
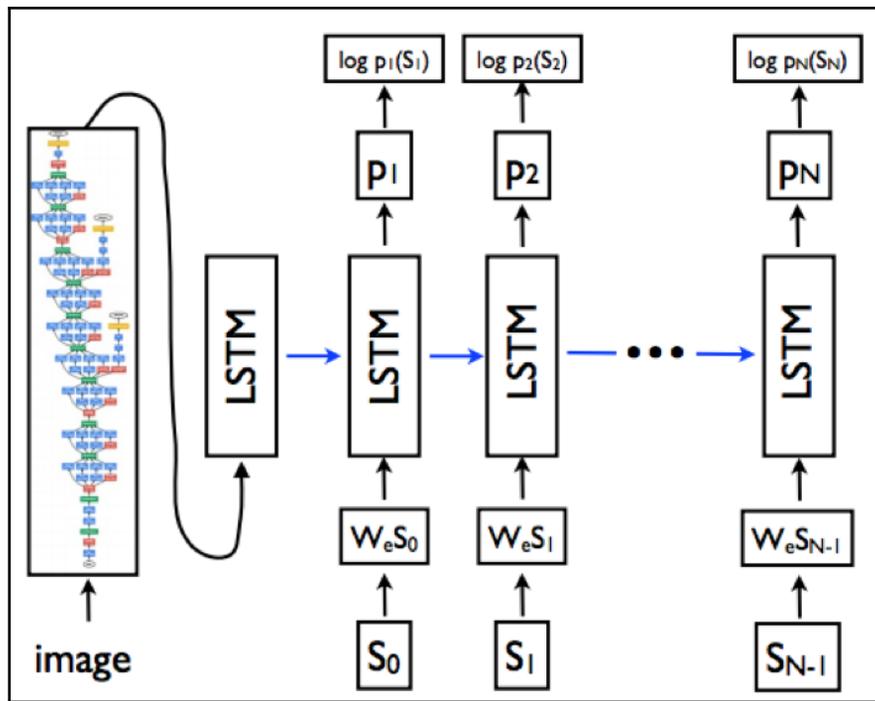
From the preceding diagram, it is pretty evident that the recurrent connections are represented by the blue horizontal arrows, based on the unrolled LSTM architecture, and have been transformed into feed-forward connections. Also, as evident, the output m_{t-1} of the LSTM at time t-1, is fed to the next LSTM at time t, and so on. Considering the source input image as I and the caption as $S = \{S_0, S_1, ..., S_N\}$, the following diagram depicts the major operations involved in the unrolled architecture depicted precedingly:

$$
\begin{aligned}
x_{-1} &= \mathbf{CNN}(I) \\
x_t &= W_e S_t, \quad t \in \{0 \ldots N-1\} \\
p_{t+1} &= \mathbf{LSTM}(x_t), \quad t \in \{0 \ldots N-1\}
\end{aligned}
$$

Here, each text word in the caption is represented by a one-hot vector S_t such that it's dimensions are equal to the size of our vocabulary (unique words). Also a point to note is that we have special markers or delimiter words for S_0, which we denote by <START> and S_N, which we denote by <END> to mark the start and end of a caption. This helps the LSTM understand when the caption has been generated completely.

The input image *I* is input to our DCNN model, which generates dense feature vectors and the words are transformed into dense word embeddings W_ϵ based on the embedding layer. The overall loss function to be minimized is thus the negative log-likelihood of the right word at every step, as depicted in the following equation:

$$L(I, S) = -\sum_{t=1}^{N} log\, p_t(S_t)$$

This loss is hence minimized during our model training, considering all the parameters in our model including our DCNN, LSTM, and embeddings. Let's now look at how we can put this into action.

Practical hands-on approach

Now that we know the essential concepts and theory behind what can be used to build a successful image caption generator, let's look at the major building blocks that we will need to follow a hands-on approach to solving this problem. Based on the major operations for image captioning, to build a model, we will need the following major components:

- Image feature extractor—DCNN model with transfer learning
- Text caption generator—sequence-based language model with LSTM
- Encoder-decoder model

Let's briefly talk about these three components before implementing them for our caption generation system.

Image feature extractor – DCNN model with transfer learning

One of the main inputs for our system will be a source image or photo. We all know that a **machine learning** (**ML**) or deep learning model can't just use a raw image. We need some processing and also we need to extract relevant features from the image, which can then be used for tasks such as recognition and classification.

An image-feature extractor should essentially take in an input image, extract a rich hierarchical feature representation from it, and represent the output in the form of a fixed-length dense vector. We have already seen the power of DCNNs for handling computer vision tasks. Here, we will leverage the power of transfer learning by using the pretrained VGG-16 model as a feature extractor to extract bottleneck features from all our images. Just as a quick refresher, the following diagram shows the VGG-16 model:
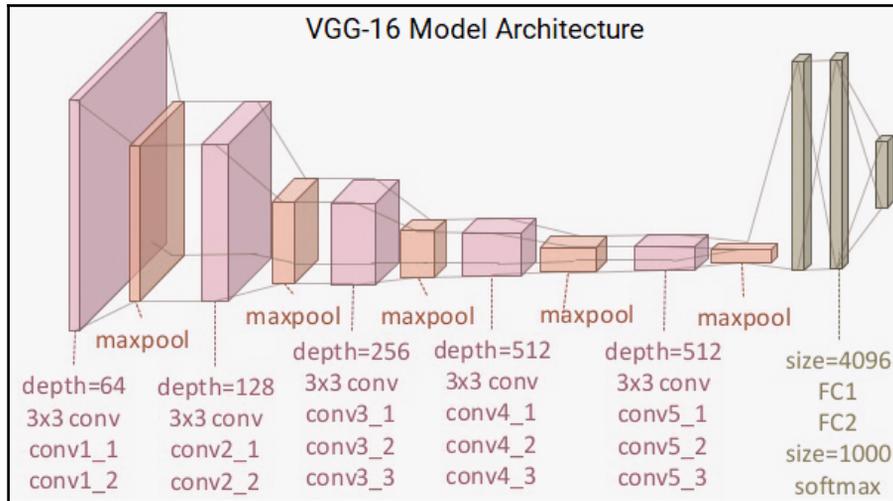
For the purpose of feature extraction, we will remove the top part of the model, the softmax layer, and use the remaining layers for getting dense feature vectors from our input images. This is typically part of the encoding process and the output is fed into the decoder where the captions are generated.

Text caption generator – sequence-based language model with LSTM

A traditional sequence-based language model will predict the next probable word if it knows the previous words already occurring in the sequence. For our image captioning problem, as discussed in the previous section, based on the features from the DCNN model and the words already generated in the caption sequence, an LSTM model should be able to predict the next probable word in our caption at every time step.

An embedding layer is used to generate word embeddings for every unique word in our caption data dictionary or vocabulary, which is usually fed as an input to the LSTM model (part of our decoder) to generate the next probable word in our caption based on the image features and previous word sequence. The idea is to finally generate a sequence of words that together make the most sense in describing the input image.

Encoder-decoder model

This is the model architecture that ties the two previous components together. It was originally a great success with neural machine translation, where you typically input words from one language into the encoder and the decoder outputs words in another language. The advantage is that with a single end-to-end architecture, you can connect these two components and solve the problem instead of trying to build separate and disconnected models to solve one problem.

The DCNN model typically forms the encoder that encodes the source input image into a fixed-length dense vector, which is then decoded by the LSTM-based sequence model into a sequence of words, giving us our desired caption. Also, as discussed earlier, this model has to be trained to maximize the likelihood of the caption text given the input image. For improvements, you can consider adding details to this model as a part of the future scope.

Let's now use this approach to implement our automated image caption generator.

Image feature extraction with transfer learning

The first step for our model is to leverage a pretrained DCNN model, using principles of transfer learning to extract the right features from our source images. To keep things simple, we will not be fine-tuning or connecting the VGG-16 model to the rest of our model architecture. We will be extracting the bottleneck features from all our images beforehand to speed up training later, since building a sequence model with several LSTMs will take a lot of training time even on GPUs, as we will see shortly.

To get started, we will load up all the source image filenames and their corresponding captions from the `Flickr8k_text` folder in the source dataset. Also we will combine the `dev` and `train` dataset images together, as we mentioned before:

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

```
# read train image file names
with open('../Flickr8k_text/Flickr_8k.trainImages.txt','r') as tr_imgs:
    train_imgs = tr_imgs.read().splitlines()

# read dev image file names
with open('../Flickr8k_text/Flickr_8k.devImages.txt','r') as dv_imgs:
    dev_imgs = dv_imgs.read().splitlines()

# read test image file names
with open('../Flickr8k_text/Flickr_8k.testImages.txt','r') as ts_imgs:
    test_imgs = ts_imgs.read().splitlines()

# read image captions
with open('../Flickr8k_text/Flickr8k.token.txt','r') as img_tkns:
    captions = img_tkns.read().splitlines()
# combine dev and train image names into one set
train_imgs = train_imgs + dev_imgs
```

Now that we have the input image filenames sorted out and the corresponding captions loaded up, we need to build a dictionary-based map that maps a source image with its corresponding captions. As we mentioned earlier, one image was captioned by five different people and hence we will have a list of five captions for each image. The following code helps us do this:

```
from collections import defaultdict

caption_map = defaultdict(list)
# store five captions in a list for each image
for record in captions:
    record = record.split('\t')
    img_name = record[0][:-2]
    img_caption = record[1].strip()
    caption_map[img_name].append(img_caption)
```

We will be leveraging this later on when we build our datasets for training and testing. Let's now focus on feature extraction. Before the image feature extraction, we need to preprocess the raw input images to the right size and scale the pixel values based on the model we will be using. The following code will help us with the necessary image preprocessing steps:

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input as
preprocess_vgg16_input

def process_image2arr(path, img_dims=(224, 224)):
    img = image.load_img(path, target_size=img_dims)
    img_arr = image.img_to_array(img)
```

```
        img_arr = np.expand_dims(img_arr, axis=0)
        img_arr = preprocess_vgg16_input(img_arr)
        return img_arr
```

We will also need to load up the pretrained VGG-16 model to leverage transfer learning. This is achieved with the following code snippet:

```
from keras.applications import vgg16
from keras.models import Model


vgg_model = vgg16.VGG16(include_top=True, weights='imagenet',
                        input_shape=(224, 224, 3))
vgg_model.layers.pop()
output = vgg_model.layers[-1].output
vgg_model = Model(vgg_model.input, output)
vgg_model.trainable = False

vgg_model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
...		
...		
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312

```
Total params: 134,260,544
Trainable params: 0
Non-trainable params: 134,260,544
```

It is quite evident that we remove the softmax layer and make the model non-trainable, since we are only interested in extracting dense feature vectors from the input images. We will now build a function that leverages our utility functions and helps extract the right features from input images:

```
def extract_tl_features_vgg(model, image_file_name,
                            image_dir='../Flickr8k_imgs/'):
    pr_img = process_image2arr(image_dir+image_file_name)
    tl_features = model.predict(pr_img)
    tl_features = np.reshape(tl_features, tl_features.shape[1])
    return tl_features
```

Let's now put all our previous functions and our pretrained model to the test by extracting image features and building out our train and test datasets:

```
img_tl_featureset = dict()
train_img_names = []
train_img_captions = []
test_img_names = []
test_img_captions = []

for img in train_imgs:
    img_tl_featureset[img] = extract_tl_features_vgg(model=vgg_model,
                                 image_file_name=img)
    for caption in caption_map[img]:
        train_img_names.append(img)
        train_img_captions.append(caption)
for img in test_imgs:
    img_tl_featureset[img] = extract_tl_features_vgg(model=vgg_model,
                                 image_file_name=img)
    for caption in caption_map[img]:
        test_img_names.append(img)
        test_img_captions.append(caption)
train_dataset = pd.DataFrame({'image': train_img_names, 'caption':
                              train_img_captions})
test_dataset = pd.DataFrame({'image': test_img_names, 'caption':
                             test_img_captions})
print('Train Dataset Size:', len(train_dataset), '\tTest Dataset Size:',
len(test_dataset))

Train Dataset Size: 35000  Test Dataset Size: 5000
```

We can also see what the train dataset looks like by using the following code:

```
train_dataset.head(10)
```

The output of the preceding code is as follows:

	caption	image
0	A black dog is running after a white dog in th...	2513260012_03d33305cf.jpg
1	Black dog chasing brown dog through snow	2513260012_03d33305cf.jpg
2	Two dogs chase each other across the snowy gro...	2513260012_03d33305cf.jpg
3	Two dogs play together in the snow .	2513260012_03d33305cf.jpg
4	Two dogs running through a low lying body of w...	2513260012_03d33305cf.jpg
5	A little baby plays croquet .	2903617548_d3e38d7f88.jpg
6	A little girl plays croquet next to a truck .	2903617548_d3e38d7f88.jpg
7	The child is playing croquette by the truck .	2903617548_d3e38d7f88.jpg
8	The kid is in front of a car with a put and a ...	2903617548_d3e38d7f88.jpg
9	The little boy is playing with a croquet hamme...	2903617548_d3e38d7f88.jpg

It is quite evident that we have five captions for each input image and we maintain that in our datasets. We will now save the record of these datasets and our image features learned from transfer learning to the disk so we can easily load it up in memory during model training instead of extracting these features every time we want to run our model:

```
# save dataset records
train_dataset = train_dataset[['image', 'caption']]
test_dataset = test_dataset[['image', 'caption']]

train_dataset.to_csv('image_train_dataset.tsv', sep='\t', index=False)
test_dataset.to_csv('image_test_dataset.tsv', sep='\t', index=False)

# save transfer learning image features
from sklearn.externals import joblib
joblib.dump(img_tl_featureset, 'transfer_learn_img_features.pkl')

['transfer_learn_img_features.pkl']
```

Also, if needed, you can validate how the image features look by using the following code snippets for some initial checks:

```
[(key, value.shape) for key, value in
                         img_tl_featureset.items()][:5]

[('3079787482_0757e9d167.jpg', (4096,)),
 ('3284955091_59317073f0.jpg', (4096,)),
 ('1795151944_d69b82f942.jpg', (4096,)),
 ('3532192208_64b069d05d.jpg', (4096,)),
 ('454709143_9c513f095c.jpg', (4096,))]


[(k, np.round(v, 3)) for k, v in img_tl_featureset.items()][:5]

[('3079787482_0757e9d167.jpg',
  array([0., 0., 0., ..., 0., 0., 0.], dtype=float32)),
 ('3284955091_59317073f0.jpg',
  array([0.615, 0.   , 0.653, ..., 0.   , 1.559, 2.614], dtype=float32)),
 ('1795151944_d69b82f942.jpg',
  array([0.   , 0.   , 0.   , ..., 0.   , 0.   , 0.538], dtype=float32)),
 ('3532192208_64b069d05d.jpg',
  array([0.   , 0.   , 0.   , ..., 0.   , 0.   , 2.293], dtype=float32)),
 ('454709143_9c513f095c.jpg',
  array([0.   , 0.   , 0.131, ..., 0.833, 4.263, 0.   ], dtype=float32))]
```

We will be using these features in the next part of our modeling.

Building a vocabulary for our captions

The next step involves doing some preprocessing for our caption data and building a vocabulary or metadata dictionary for our captions. We start by reading in our training dataset records and writing a function to preprocess our text captions:

```
train_df = pd.read_csv('image_train_dataset.tsv', delimiter='\t')
total_samples = train_df.shape[0]
total_samples

35000

# function to pre-process text captions
def preprocess_captions(caption_list):
    pc = []
    for caption in caption_list:
        caption = caption.strip().lower()
        caption = caption.replace('.', '').replace(',',
```

```
                                    '').replace("'", "").replace('"', '')
            caption = caption.replace('&','and').replace('(','').replace(')',
                                          '').replace('-', ' ')
            caption = ' '.join(caption.split())
            caption = '<START> '+caption+' <END>'
            pc.append(caption)
        return pc
```

We will now preprocess our captions and build some basic metadata for our vocabulary, including utilities for converting unique words into numeric representations and vice-versa:

```
# pre-process caption data
train_captions = train_df.caption.tolist()
processed_train_captions = preprocess_captions(train_captions)

tc_tokens = [caption.split() for caption in
                    processed_train_captions]
tc_tokens_length = [len(tokenized_caption) for tokenized_caption
                        in tc_tokens]

# build vocabulary metadata
from collections import Counter

tc_words = [word.strip() for word_list in tc_tokens for word in
                        word_list]
unique_words = list(set(tc_words))
token_counter = Counter(unique_words)

word_to_index = {item[0]: index+1 for index, item in
                    enumerate(dict(token_counter).items())}
word_to_index['<PAD>'] = 0
index_to_word = {index: word for word, index in
                        word_to_index.items()}
vocab_size = len(word_to_index)
max_caption_size = np.max(tc_tokens_length)
```

It is important to ensure we save this vocabulary metadata to disk so we can reuse it anytime in the future for model training as well as predictions; otherwise, if we re-generate our vocabulary, it is quite possible that a model may have been trained with some other version of a vocabulary where word-to-number mappings may have been different. This would give us the wrong results and we'd lose valuable time:

```
from sklearn.externals import joblib

vocab_metadata = dict()
vocab_metadata['word2index'] = word_to_index
```

```
vocab_metadata['index2word'] = index_to_word
vocab_metadata['max_caption_size'] = max_caption_size
vocab_metadata['vocab_size'] = vocab_size
joblib.dump(vocab_metadata, 'vocabulary_metadata.pkl')

['vocabulary_metadata.pkl']
```

If needed, you can check the contents of our vocabulary metadata using the following code snippet and also see how a general preprocessed text caption might look for one of the images:

```
# check vocabulary metadata
{k: v if type(v) is not dict
        else list(v.items())[:5]
            for k, v in vocab_metadata.items()}

{'index2word': [(0, '<PAD>'), (1, 'nearby'), (2, 'flooded'),
                (3, 'fundraising'), (4, 'snowboarder')],
 'max_caption_size': 39,
 'vocab_size': 7927,
 'word2index': [('reflections', 4122), ('flakes', 1829),
        ('flexing', 7684), ('scaling', 1057), ('pretend', 6788)]}

# check pre-processed caption
processed_train_captions[0]

'<START> a black dog is running after a white dog in the snow <END>'
```

We will be leveraging this shortly when we build a data-generator function that will serve as input for our deep learning model during model training.

Building an image caption dataset generator

One of the most essential steps in any complex deep learning system that consumes large amounts of data is to build an efficient dataset generator. This is very relevant in our system, especially because we will be dealing with image and text data. Besides that, we will be dealing with sequence models where we have to pass the same data multiple times to our model during training. Unpacking all the data in lists, pre-building datasets would be the most in-efficient way to tackle this problem. Hence we will be leveraging the power of generators for our system.

To start with, we will load up our image features learned from transfer learning, along with our vocabulary metadata, using the following code:

```
from sklearn.externals import joblib

tl_img_feature_map = joblib.load('transfer_learn_img_features.pkl')
vocab_metadata = joblib.load('vocabulary_metadata.pkl')


train_img_names = train_df.image.tolist()
train_img_features = [tl_img_feature_map[img_name] for img_name in
train_img_names]
train_img_features = np.array(train_img_features)

word_to_index = vocab_metadata['word2index']
index_to_word = vocab_metadata['index2word']
max_caption_size = vocab_metadata['max_caption_size']
vocab_size = vocab_metadata['vocab_size']

train_img_features.shape

(35000, 4096)
```

We can see that there are 35,000 images, where each image has a dense feature vector representation of size 4,096. The idea now is to build a model-dataset generator that will generate (input, output) pairs. For our input, we will use our source images converted into dense feature vectors, as well as the corresponding image caption, with one word added at each time step. The corresponding output will be the next word (which has to be predicted) for the same caption for the corresponding input image and caption. The following figure makes this approach clearer:
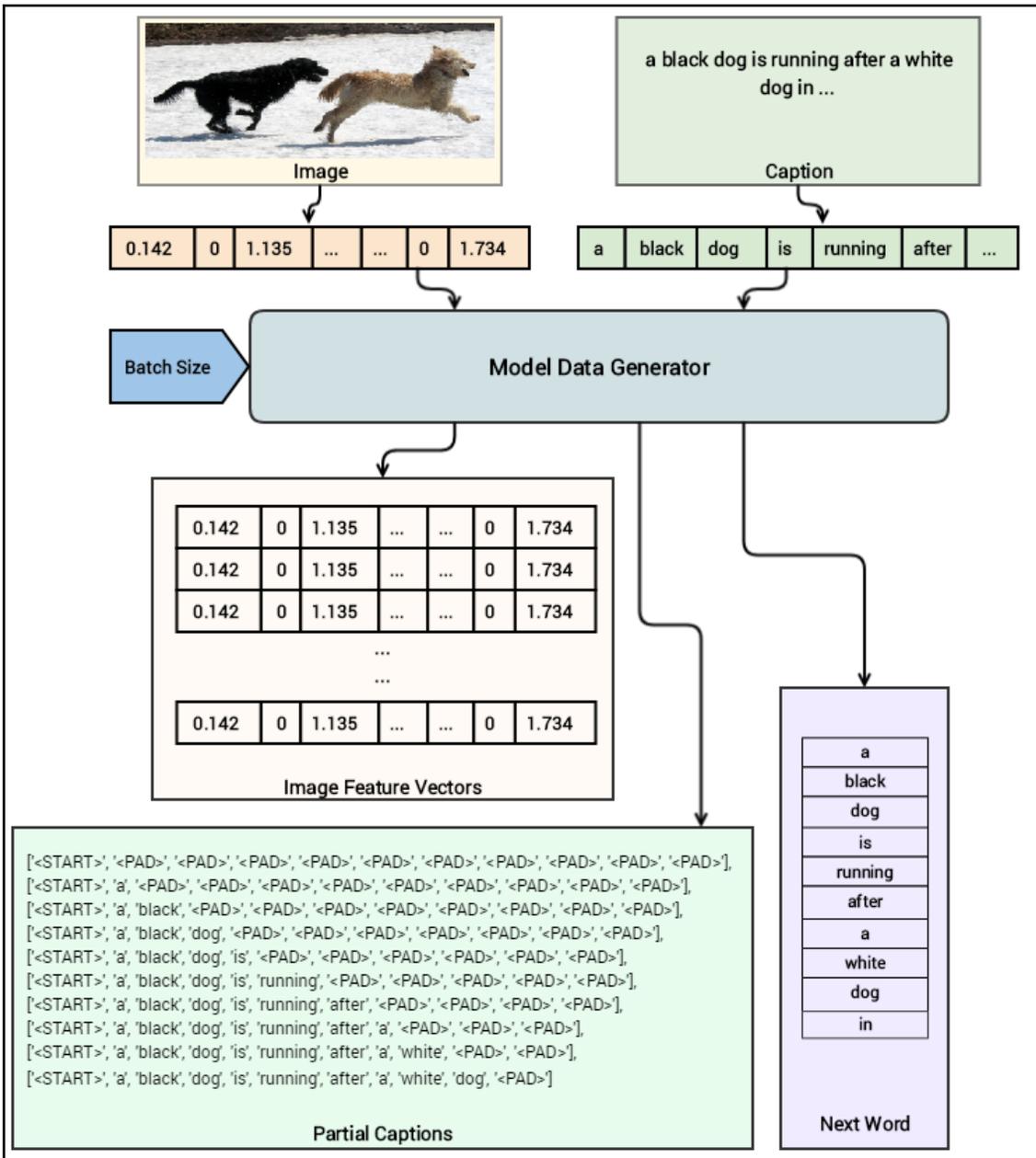
Based on the this architecture, it is clear that at each time step for the same image, we pass the same feature vector and keep adding one word of the caption at a time while passing the next word to be predicted as the corresponding output to train our model. The following function will help us achieve this, where we leverage Python generators for lazy loading and to be more memory-efficient:

```
from keras.preprocessing import sequence

def dataset_generator(processed_captions, transfer_learnt_features,
vocab_size, max_caption_size, batch_size=32):
    partial_caption_set = []
    next_word_seq_set = []
    img_feature_set = []
    batch_count = 0
    batch_num = 0
    while True:
        for index, caption in enumerate(processed_captions):
            img_features = transfer_learnt_features[index]
            for cap_idx in range(len(caption.split()) - 1):
                partial_caption = [word_to_index[word] for word in
                                    caption.split()[:cap_idx+1]]
                partial_caption_set.append(partial_caption)

                next_word_seq = np.zeros(vocab_size)
                next_word_seq[word_to_index
                                [caption.split()[cap_idx+1]]] = 1
                next_word_seq_set.append(next_word_seq)
                img_feature_set.append(img_features)
                batch_count+=1

                if batch_count >= batch_size:
                    batch_num += 1
                    img_feature_set = np.array(img_feature_set)
                    partial_caption_set =
                                sequence.pad_sequences(
                                    sequences=partial_caption_set,
                                    maxlen=max_caption_size,
                                    padding='post')
                    next_word_seq_set =
                                    np.array(next_word_seq_set)
                    yield [[img_feature_set, partial_caption_set],
                            next_word_seq_set]
                    batch_count = 0
                    partial_caption_set = []
                    next_word_seq_set = []
                    img_feature_set = []
```

Let's try to understand how this function really works! While we do have a nice visual depicted in the previous figure, we will now do a sample data-generation for a batch size of 10 using the following code:

```
MAX_CAPTION_SIZE = max_caption_size
VOCABULARY_SIZE = vocab_size
BATCH_SIZE = 10

print('Vocab size:', VOCABULARY_SIZE)
print('Max caption size:', MAX_CAPTION_SIZE)
print('Test Batch size:', BATCH_SIZE)

d = dataset_generator(processed_captions=processed_train_captions,
                      transfer_learnt_features=train_img_features,
                      vocab_size=VOCABULARY_SIZE,
                      max_caption_size=MAX_CAPTION_SIZE,
                      batch_size=BATCH_SIZE)
d = list(d)
img_features, partial_captions = d[0][0]
next_word = d[0][1]

Vocab size: 7927
Max caption size: 39
Test Batch size: 10
```

We can now verify the dimensions of the returned datasets from our data-generator function using the following code:

```
img_features.shape, partial_captions.shape, next_word.shape

((10, 4096), (10, 39), (10, 7927))
```

It is quite clear that our image features are essentially dense vectors of 4,096 features in each vector. The same feature vector is repeated for the same image at every time step of the caption. The caption-generated vector is of size `MAX_CAPTION_SIZE`, which is 39. The next word is typically returned in a one-hot encoded fashion, which is very useful to serve as the input for the softmax layer to check whether the correct word is predicted by the model. The following code shows us how the image feature vector looks for a batch size of 10 for an input image:

```
np.round(img_features, 3)

array([[0.    , 0.    , 1.704, ..., 0.    , 0.    , 0.    ],
       [0.    , 0.    , 1.704, ..., 0.    , 0.    , 0.    ],
       [0.    , 0.    , 1.704, ..., 0.    , 0.    , 0.    ],
       ...,
       [0.    , 0.    , 1.704, ..., 0.    , 0.    , 0.    ],
```

```
[0.    , 0.    , 1.704, ..., 0.    , 0.    , 0.    ],
[0.    , 0.    , 1.704, ..., 0.    , 0.    , 0.    ]], dtype=float32)
```

As we discussed before, the same feature vector for the image is repeated at each time-step during the batch data-generation process. We can check how the caption is formed at each time-step that is fed to the model as input. We show only the first 11 words for simplicity in the output:

```
# display raw caption tokens at each time-step
print(np.array([partial_caption[:11] for partial_caption in
                partial_captions]))

[[6917    0    0    0    0    0    0    0    0    0    0]
 [6917 2578    0    0    0    0    0    0    0    0    0]
 [6917 2578 7371    0    0    0    0    0    0    0    0]
 [6917 2578 7371 3519    0    0    0    0    0    0    0]
 [6917 2578 7371 3519 3113    0    0    0    0    0    0]
 [6917 2578 7371 3519 3113 6720    0    0    0    0    0]
 [6917 2578 7371 3519 3113 6720    7    0    0    0    0]
 [6917 2578 7371 3519 3113 6720    7 2578    0    0    0]
 [6917 2578 7371 3519 3113 6720    7 2578 1076    0    0]
 [6917 2578 7371 3519 3113 6720    7 2578 1076 3519    0]]


# display actual caption tokens at each time-step
print(np.array([[index_to_word[word] for word in cap][:11] for cap
                in partial_captions]))

[['<START>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' 'black' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' 'black' 'dog' '<PAD>' '<PAD>' '<PAD>' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' 'black' 'dog' 'is' '<PAD>' '<PAD>' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' 'black' 'dog' 'is' 'running' '<PAD>' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' 'black' 'dog' 'is' 'running' 'after' '<PAD>' '<PAD>'
  '<PAD>' '<PAD>']
 ['<START>' 'a' 'black' 'dog' 'is' 'running' 'after' 'a' '<PAD>' '<PAD>'
  '<PAD>']
 ['<START>' 'a' 'black' 'dog' 'is' 'running' 'after' 'a' 'white' '<PAD>'
  '<PAD>']
 ['<START>' 'a' 'black' 'dog' 'is' 'running' 'after' 'a' 'white' 'dog'
  '<PAD>']]
```

We can clearly see how one word is added at each step to the input caption after the `<START>` symbol, which signifies the start of the text caption. Let's look at the corresponding next word generation output now (which is typically the next word to be predicted based on the two input):

```
next_word

array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])


print('Next word positions:', np.nonzero(next_word)[1])
print('Next words:', [index_to_word[word] for word in
                np.nonzero(next_word)[1]])

Next word positions: [2578 7371 3519 3113 6720    7 2578 1076 3519 5070]
Next words: ['a', 'black', 'dog', 'is', 'running', 'after', 'a', 'white',
'dog', 'in']
```

It is quite clear that the next word typically points to the next correct word in the caption based on the sequence of words in the input caption for each time-step. This is the data that will be fed to our model at each epoch during training.

Building our image language encoder-decoder deep learning model

We have all the essential components and utilities needed to build our model now. As we mentioned earlier, we will be using an encoder-decoder deep learning model architecture to build our image-captioning system.

The following code helps us build the architecture for this model, where we take pairs of image features and caption sequences as input to predict the next possible word in the caption at each time-step:

```
from keras.models import Sequential, Model
from keras.layers import LSTM, Embedding, TimeDistributed, Dense,
RepeatVector, Activation, Flatten, concatenate

DENSE_DIM = 256
EMBEDDING_DIM = 256
MAX_CAPTION_SIZE = max_caption_size
VOCABULARY_SIZE = vocab_size

image_model = Sequential()
image_model.add(Dense(DENSE_DIM, input_dim=4096, activation='relu'))
image_model.add(RepeatVector(MAX_CAPTION_SIZE))

language_model = Sequential()
language_model.add(Embedding(VOCABULARY_SIZE, EMBEDDING_DIM,
input_length=MAX_CAPTION_SIZE))
language_model.add(LSTM(256, return_sequences=True))
language_model.add(TimeDistributed(Dense(DENSE_DIM)))

merged_output = concatenate([image_model.output, language_model.output])
merged_output = LSTM(1024, return_sequences=False)(merged_output)
merged_output = (Dense(VOCABULARY_SIZE))(merged_output)
merged_output = Activation('softmax')(merged_output)

model = Model([image_model.input, language_model.input], merged_output)
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])

model.summary()
```

The output for the preceding code is as follows:

```
Layer (type)                    Output Shape         Param #     Connected to
=================================================================================================
embedding_1_input (InputLayer)  (None, 39)           0

dense_1_input (InputLayer)      (None, 4096)         0

embedding_1 (Embedding)         (None, 39, 256)      2029312     embedding_1_input[0][0]

dense_1 (Dense)                 (None, 256)          1048832     dense_1_input[0][0]

lstm_1 (LSTM)                   (None, 39, 256)      525312      embedding_1[0][0]

repeat_vector_1 (RepeatVector)  (None, 39, 256)      0           dense_1[0][0]

time_distributed_1 (TimeDistribu (None, 39, 256)     65792       lstm_1[0][0]

concatenate_1 (Concatenate)     (None, 39, 512)      0           repeat_vector_1[0][0]
                                                                 time_distributed_1[0][0]

lstm_2 (LSTM)                   (None, 1024)         6295552     concatenate_1[0][0]

dense_3 (Dense)                 (None, 7927)         8125175     lstm_2[0][0]

activation_1 (Activation)       (None, 7927)         0           dense_3[0][0]
=================================================================================================
Total params: 18,089,975
Trainable params: 18,089,975
Non-trainable params: 0
```

We can see from the preceding architecture that we have an image model that focuses more on handling the image-based features as its input, and the language model leverages LSTMs to handle the sequence of words flowing in per image caption. The last layer is a softmax layer with 7,927 units because we have a total of 7,927 unique words in our vocabulary and the next predicted word in our caption will be one of these words which gets generated as the output. We can also visualize our model architecture using the following code snippet:

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model, show_shapes=True, show_layer_names=False,
    rankdir='TB').create(prog='dot', format='svg'))
```

The output for the preceding code is as follows:

InputLayer	input:	(None, 39)
	output:	(None, 39)

Embedding	input:	(None, 39)
	output:	(None, 39, 256)

InputLayer	input:	(None, 4096)
	output:	(None, 4096)

LSTM	input:	(None, 39, 256)
	output:	(None, 39, 256)

Dense	input:	(None, 4096)
	output:	(None, 256)

TimeDistributed(Dense)	input:	(None, 39, 256)
	output:	(None, 39, 256)

RepeatVector	input:	(None, 256)
	output:	(None, 39, 256)

Concatenate	input:	[(None, 39, 256), (None, 39, 256)]
	output:	(None, 39, 512)

LSTM	input:	(None, 39, 512)
	output:	(None, 1024)

Dense	input:	(None, 1024)
	output:	(None, 7927)

Activation	input:	(None, 7927)
	output:	(None, 7927)

Training our image captioning deep learning model

Before we start training our model, since we are dealing with some complex components in our model, we use a callback in our model to reduce the learning rate in case there is a plateau in the model's accuracy across successive epochs. This is extremely helpful to change the learning rate of the model on the fly without stopping training:

```
from keras.callbacks import ReduceLROnPlateau
reduce_lr = ReduceLROnPlateau(monitor='loss', factor=0.15,
                              patience=2, min_lr=0.000005)
```

Let's train our model now! We have trained our model to around 30 to 50 epochs and saved the model at around 30 epochs and again at 50 epochs:

```
BATCH_SIZE = 256
EPOCHS = 30
cap_lens = [(cl-1) for cl in tc_tokens_length]
total_size = sum(cap_lens)

history = model.fit_generator(
  dataset_generator(processed_captions=processed_train_captions,
                    transfer_learnt_features=train_img_features,
                    vocab_size=VOCABULARY_SIZE,
                    max_caption_size=MAX_CAPTION_SIZE,
                    batch_size=BATCH_SIZE),
  steps_per_epoch=int(total_size/BATCH_SIZE),
  callbacks=[reduce_lr],
  epochs=EPOCHS, verbose=1)

Epoch 1/30
1617/1617 - 724s 448ms/step - loss: 4.1236 - acc: 0.2823
Epoch 2/30
1617/1617 - 725s 448ms/step - loss: 3.9182 - acc: 0.3150
Epoch 3/30
1617/1617 - 724s 448ms/step - loss: 3.8286 - acc: 0.3281
...
...
Epoch 29/30
1617/1617 - 724s 447ms/step - loss: 3.6443 - acc: 0.3885
Epoch 30/30
1617/1617 - 724s 448ms/step - loss: 3.4656 - acc: 0.4078

model.save('ic_model_rmsprop_b256ep30.h5')
```

Once we save this model, we resume training and train it for 20 more epochs, stopping at
50. Of course, you are free to also use model-checkpointing in Keras to save it
automatically at regular intervals:

```
EPOCHS = 50

history_rest = model.fit_generator(
    dataset_generator(processed_captions=processed_train_captions,
                      transfer_learnt_features=train_img_features,
                      vocab_size=VOCABULARY_SIZE,
                      max_caption_size=MAX_CAPTION_SIZE,
                      batch_size=BATCH_SIZE),
    steps_per_epoch=int(total_size/BATCH_SIZE),
    callbacks=[reduce_lr],
    epochs=EPOCHS, verbose=1, initial_epoch=30)

Epoch 31/50
1617/1617 - 724s 447ms/step - loss: 3.3988 - acc: 0.4144
Epoch 32/50
1617/1617 - 724s 448ms/step - loss: 3.3633 - acc: 0.4184
...
...
Epoch 49/50
1617/1617 - 724s 448ms/step - loss: 3.1330 - acc: 0.4509
Epoch 50/50
1617/1617 - 724s 448ms/step - loss: 3.1260 - acc: 0.4523

model.save('ic_model_rmsprop_b256ep50.h5')
```

This concludes our model-training process; we have successfully trained our image
captioning model and can start using it to generate image captions for new images.

> **Model training tip**: An image captioning model usually uses a lot of data
> and involves a lot of parameters during training. It is recommended to use
> generators to build and generate the data to train the deep learning
> model. Otherwise, you are likely to run into memory issues. Also, this
> model takes close to 12 minutes to run per epoch on an Amazon AWS
> p2.x instance with a Tesla K80 GPU, so consider building this model on a
> GPU because it might take too long to train on a conventional system.

We can also view the trends with regard to model accuracy, loss, and learning rate based on
the different epochs during the training process:

```
epochs = list(range(1,51))
losses = history.history['loss'] + history_rest.history['loss']
accs = history.history['acc'] + history_rest.history['acc']
```

```
lrs = history.history['lr'] + history_rest.history['lr']

f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(14, 4))
title = f.suptitle("Model Training History", fontsize=14)
f.subplots_adjust(top=0.85, wspace=0.35)

ax1.plot(epochs, losses, label='Loss')
ax2.plot(epochs, accs,   label='Accuracy')
ax3.plot(epochs, lrs, label='Learning Rate')

ax1.set_xlabel('Epochs')
ax2.set_xlabel('Epochs')
ax3.set_xlabel('Epochs')
ax1.set_ylabel('Loss')
ax2.set_ylabel('Accuracy')
ax3.set_ylabel('Learning Rate')
```

The output for the preceding code is as follows:
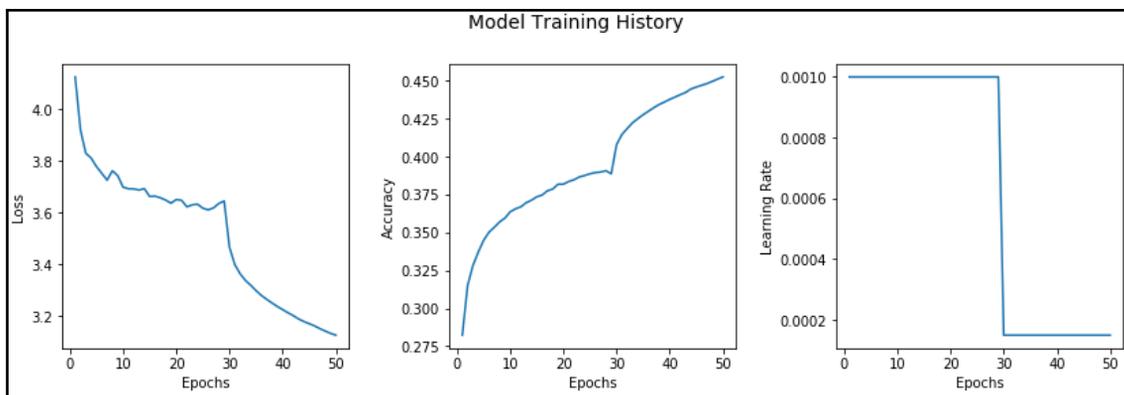
We can see that there was a slight decrease in accuracy and an increase in loss around epochs 28 and 29, which led to our callback successfully reducing the learning rate, which led to a boost in accuracy from epoch 30 onward. This definitely gives us some useful insights into our model's behavior!

Evaluating our image captioning deep learning model

Training a model and not evaluating its performance makes no sense at all. Hence, we will now be evaluating our deep learning model's performance on the test dataset, which has a total of 1,000 different images from the `Flickr8K` dataset. We start off by loading up the usual dependencies in case you don't already have them:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

pd.options.display.max_colwidth = 500

%matplotlib inline
```

Loading up data and models

The next steps include loading up the necessary data, model, and other assets from disk into memory. We first load up our test dataset and our trained deep learning models:

```
# load test dataset
test_df = pd.read_csv('image_test_dataset.tsv', delimiter='\t')

# load the models
from keras.models import load_model

model1 = load_model('ic_model_rmsprop_b256ep30.h5')
model2 = load_model('ic_model_rmsprop_b256ep50.h5')
```

We now need to load up necessary metadata assets, such as the image features we extracted before for our test data, and our vocabulary metadata:

```
from sklearn.externals import joblib

tl_img_feature_map = joblib.load('transfer_learn_img_features.pkl')
vocab_metadata = joblib.load('vocabulary_metadata.pkl')
word_to_index = vocab_metadata['word2index']
index_to_word = vocab_metadata['index2word']
max_caption_size = vocab_metadata['max_caption_size']
vocab_size = vocab_metadata['vocab_size']
```

Understanding greedy and beam search

For generating predictions from our deep learning-based neural image captioning model, remember that it is not as straightforward as a basic classification or categorization model. We will need to generate a sequence of words from our model at each time-step based on the input image features. There are multiple ways of generating these sequence of words for the captions.

One approach is known as **sampling**, or **greedy search**, where we start with the `<START>` token, input image features, and then generate the first word based on *p1* from the LSTM output. Then we feed in the corresponding predicted word embedding as input and generate the next word based on *p2* from the next LSTM (in the unrolled form we talked about previously). This step continues until we reach the `<END>` token, which signifies the end of the caption, or we reach the maximum possible length of the token based on a predefined threshold.

The second approach is known as **beam search**, which is slightly more effective than a greedy-based search, where we select the most likely word at each step based on the highest probability considering the previously-generated words at each sequence, which is exactly what sampling does. Beam search expands upon the greedy search technique and always returns a list of the most likely output sequence of terms. Thus, as each sequence is constructed, to generate the next term at time step *t + 1*, instead of doing a greedy search and generating the most probable next term, it iteratively considers a set of *k* best sentences based on expanding to all possible next terms in the next time step. The value of *k* is usually a user-specified parameter that is used to control the total number of parallel, or beam, searches conducted to generate the caption sequences. Thus in beam search, we start with *k* most likely words as the first time step output in the caption sequence and keep generating the next sequence terms until one of them reach the end state. The full scope of covering detailed concepts around beam search would be out of the current scope so if you are interested, we recommend you check out any standard literature on beam search in the context of AI.

Implementing a beam search-based caption generator

We will now be implementing a basic beam search-based algorithm to generate caption sequences:

```
from keras.preprocessing import image, sequence

def get_raw_caption_sequences(model, word_to_index, image_features,
```

```
                                 max_caption_size, beam_size=1):
    start = [word_to_index['<START>']]
    caption_seqs = [[start, 0.0]]

    while len(caption_seqs[0][0]) < max_caption_size:
        temp_caption_seqs = []
        for caption_seq in caption_seqs:
            partial_caption_seq = sequence.pad_sequences(
                                            [caption_seq[0]],
                                            maxlen=max_caption_size,
                                            padding='post')
            next_words_pred = model.predict(
                                    [np.asarray([image_features]),
                                    np.asarray(partial_caption_seq)])[0]
            next_words = np.argsort(next_words_pred)[-beam_size:]

            for word in next_words:
                new_partial_caption, new_partial_caption_prob =
                                    caption_seq[0][:], caption_seq[1]
                new_partial_caption.append(word)
                new_partial_caption_prob += next_words_pred[word]
                temp_caption_seqs.append([new_partial_caption,
                                        new_partial_caption_prob])
        caption_seqs = temp_caption_seqs
        caption_seqs.sort(key = lambda item: item[1])
        caption_seqs = caption_seqs[-beam_size:]
    return caption_seqs
```

This helps us generate captions based on input image features using beam search. However, it is a raw sequence of tokens based on previous tokens at each step. Hence we will build a wrapper function on top of this that will generate a clean text sentence as a caption for an input image by leveraging the preceding function:

```
def generate_image_caption(model, word_to_index_map, index_to_word_map,
                            image_features, max_caption_size,
                            beam_size=1):
    raw_caption_seqs = get_raw_caption_sequences(model=model,
                                    word_to_index=word_to_index_map,
                                    image_features=image_features,
                                    max_caption_size=max_caption_size,
                                    beam_size=beam_size)
    raw_caption_seqs.sort(key = lambda l: -l[1])
    caption_list = [item[0] for item in raw_caption_seqs]
    captions = [[index_to_word_map[idx] for idx in caption]
                                for caption in caption_list]
    final_captions = []
    for caption in captions:
        start_index = caption.index('<START>')+1
```

```
            max_len = len(caption)
                        if len(caption) < max_caption_size
                        else max_caption_size
        end_index = caption.index('<END>')
                        if '<END>' in caption
                        else max_len-1
        proc_caption = ' '.join(caption[start_index:end_index])
        final_captions.append(proc_caption)
    return final_captions
```

We also need our caption preprocessing function from earlier, which we used to preprocess the initial captions when training the model:

```
def preprocess_captions(caption_list):
    pc = []
    for caption in caption_list:
        caption = caption.strip().lower()
        caption = caption.replace('.', '')
                        .replace(',', '')
                        .replace("'", "")
                        .replace('"', '')
        caption = caption.replace('&','and')
                        .replace('(','')
                        .replace(')', '')
                        .replace('-', ' ')
        caption = ' '.join(caption.split())
        pc.append(caption)
    return pc
```

Understanding and implementing BLEU scoring

We now need to choose an appropriate model-performance evaluation metric to evaluate how well our model might be performing. A relevant metric here is the **Bilingual Evaluation Understudy** (**BLEU**) score. This is an excellent algorithm for evaluating the performance of models in machine translation when they translate languages. The motivation behind BLEU is that closer the generated output is to human-level translations, the better the score. To this day, it remains one of the most popular metrics to compare model output against human-level output.

The simple principle of the BLEU algorithm is to evaluate the generated text captions against a set of reference captions (typically one caption is evaluated against one or more captions—in this case, five captions for each image). Scores are calculated for each caption and then averaged over the whole corpus to get an overall estimate of the quality. The BLEU score always ranges from 0 to 1, with scores closer to 1 indicating high-quality translations. Even reference text data will not be perfect since humans also make errors during captioning images, so the idea is not to attain a perfect 1 but to get a good overall BLEU score.

We will be using the `corpus_bleu(...)` function from the translate module in NLTK (`http://www.nltk.org/api/nltk.translate.html#nltk.translate.bleu_score.corpus_bleu`) to compute the BLEU score. We will be computing the overall cumulative BLEU score for 1, 2, 3, and 4-grams. Equal weights are assigned for each n-gram score for `bleu2`, `bleu3`, and `bleu4` scores, as depicted in the evaluation function we have implemented:

```
from nltk.translate.bleu_score import corpus_bleu

def compute_bleu_evaluation(reference_captions,
                                         predicted_captions):
    actual_caps = [[caption.split() for caption in sublist]
                            for sublist in reference_captions]
    predicted_caps = [caption.split()
                            for caption in predicted_captions]
    bleu1 = corpus_bleu(actual_caps,
                        predicted_caps, weights=(1.0, 0, 0, 0))
    bleu2 = corpus_bleu(actual_caps,
                        predicted_caps, weights=(0.5, 0.5, 0, 0))
    bleu3 = corpus_bleu(actual_caps,
                        predicted_caps,
                        weights=(0.3, 0.3, 0.3, 0))
    bleu4 = corpus_bleu(actual_caps, predicted_caps,
                        weights=(0.25, 0.25, 0.25, 0.25))
    print('BLEU-1: {}'.format(bleu1))
    print('BLEU-2: {}'.format(bleu2))
    print('BLEU-3: {}'.format(bleu3))
    print('BLEU-4: {}'.format(bleu4))
    return [bleu1, bleu2, bleu3, bleu4]
```

Evaluating model performance on test data

All the components for the model's performance evaluation are now ready. To evaluate our model's performance on the test dataset, we will now be loading up the image features we had extracted earlier using transfer learning, which will serve as input to our models. We will also load up the captions, preprocess them, and separate them as lists of reference captions for each image, as depicted here:

```
test_images = list(test_df['image'].unique())
test_img_features = [tl_img_feature_map[img_name]
                            for img_name in test_images]
actual_captions = list(test_df['caption'])
actual_captions = preprocess_captions(actual_captions)
actual_captions = [actual_captions[x:x+5]
                    for x in range(0, len(actual_captions),5)]
actual_captions[:2]

[['the dogs are in the snow in front of a fence',
  'the dogs play on the snow',
  'two brown dogs playfully fight in the snow',
  'two brown dogs wrestle in the snow',
  'two dogs playing in the snow'],
 ['a brown and white dog swimming towards some in the pool',
  'a dog in a swimming pool swims toward sombody we cannot see',
  'a dog swims in a pool near a person',
  'small dog is paddling through the water in a pool',
  'the small brown and white dog is in the pool']]
```

You can clearly see how each image caption is now in neat separate lists that will form our reference set of captions during the computation of the BLEU scores. We can now generate BLEU scores and test the performance of our models using different values for beam size. A few examples are depicted here:

```
# Beam Size 1 - Model 1 with 30 epochs
predicted_captions_ep30bs1 = [generate_image_caption(model=model1,
                                word_to_index_map=word_to_index,
                                index_to_word_map=index_to_word,
                                      image_features=img_feat,
                                max_caption_size=max_caption_size,
                                beam_size=1)[0]
                                    for img_feat
                                        in test_img_features]
ep30bs1_bleu = compute_bleu_evaluation(
                    reference_captions=actual_captions,
                    predicted_captions=predicted_captions_ep30bs1)

BLEU-1: 0.5049574449416513
```

```
BLEU-2: 0.3224643449851107
BLEU-3: 0.22962263359362023
BLEU-4: 0.1201459697546317


# Beam Size 1 - Model 2 with 50 epochs
predicted_captions_ep50bs1 = [generate_image_caption(model=model2,
                               word_to_index_map=word_to_index,
                               index_to_word_map=index_to_word,
                                   image_features=img_feat,
                            max_caption_size=max_caption_size,
                                  beam_size=1)[0]
                                  for img_feat
                                     in test_img_features]
ep50bs1_bleu = compute_bleu_evaluation(
                   reference_captions=actual_captions,
                   predicted_captions=predicted_captions_ep50bs1)
```

You can clearly see that scores start dropping as we start considering higher levels of n-grams. Overall, running this process is extremely time-consuming and takes a lot of time for higher orders in beam search. We tried experiments with beam sizes of 1, 3, 5, and 10. The following table depicts the model's performance for each of these experiments:

	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Epoch30 BeamSearch 1	0.504957	0.322464	0.229623	0.120146
Epoch50 BeamSearch 1	0.472661	0.296896	0.213849	0.109014
Epoch30 BeamSearch 3	0.503568	0.325535	0.237255	0.124127
Epoch50 BeamSearch 3	0.535221	0.364593	0.252667	0.167056
Epoch30 BeamSearch 5	0.519058	0.334287	0.255741	0.143111
Epoch50 BeamSearch 5	0.541103	0.377383	0.262667	0.177056
Epoch30 BeamSearch 10	0.531228	0.348353	0.267133	0.158336
Epoch50 BeamSearch 10	0.551929	0.384170	0.270649	0.188850

We can also visualize this easily in the form of a chart to see which combination of model parameters gave us the best model with the highest BLEU scores:
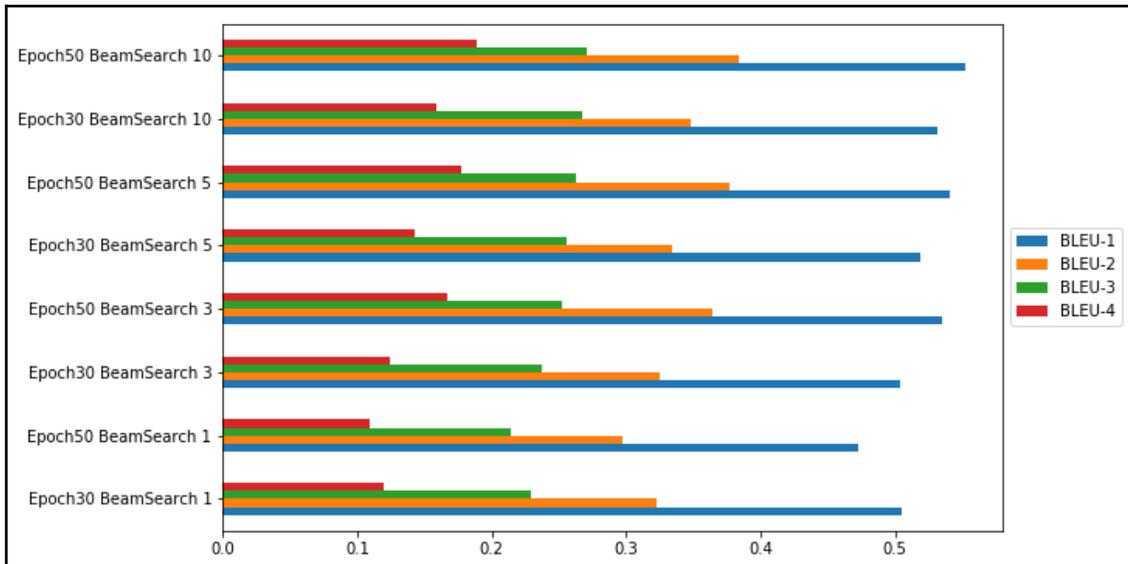
From the previous graph, it is quite clear that our model with 50 epochs and a beam size of 10 during beam search gives us the best performance based on the BLEU metrics.

Automated image captioning in action!

Evaluating on our test dataset was a good way to test the model's performance, but how do we start using the model in the real world and caption completely new photos? This is where we need some knowledge of building an end-to-end system, which takes in any image as an input and gives us a free-text natural-language caption as the output.

Here are the major components and functions for our automated caption generator:

- Caption model and metadata initializer
- Image feature extraction model initializer
- Transfer learning-based feature extractor
- Caption generator

To make this generic, we built a class that makes use of several utility functions we mentioned in the previous sections:

```python
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input as
preprocess_vgg16_input
from keras.applications import vgg16
from keras.models import Model

class CaptionGenerator:
    def __init__(self, image_locations=[],
                 word_to_index_map=None, index_to_word_map=None,
                 max_caption_size=None, caption_model=None,
                                                beam_size=1):
        self.image_locs = image_locations
        self.captions = []
        self.image_feats = []
        self.word2index = word_to_index_map
        self.index2word = index_to_word_map
        self.max_caption_size = max_caption_size
        self.vision_model = None
        self.caption_model = caption_model
        self.beam_size = beam_size
    def process_image2arr(self, path, img_dims=(224, 224)):
        img = image.load_img(path, target_size=img_dims)
        img_arr = image.img_to_array(img)
        img_arr = np.expand_dims(img_arr, axis=0)
        img_arr = preprocess_vgg16_input(img_arr)
        return img_arr
    def initialize_model(self):
        vgg_model = vgg16.VGG16(include_top=True, weights='imagenet',
                                input_shape=(224, 224, 3))
        vgg_model.layers.pop()
        output = vgg_model.layers[-1].output
        vgg_model = Model(vgg_model.input, output)
        vgg_model.trainable = False
        self.vision_model = vgg_model
    def process_images(self):
        if self.image_locs:
            image_feats = [self.vision_model.predict
```

```
                                      (self.process_image2arr
                                      (path=img_path)) for img_path
                                          in self.image_locs]
              image_feats = [np.reshape(img_feat, img_feat.shape[1]) for
                                      img_feat in image_feats]
              self.image_feats = image_feats
          else:
              print('No images specified')
      def generate_captions(self):
          captions = [generate_image_caption(model=self.caption_model,
                          word_to_index_map=self.word2index,
                          index_to_word_map=self.index2word,
                          image_features=img_feat,
   max_caption_size=self.max_caption_size, beam_size=self.beam_size)[0]
                              for img_feat in self.image_feats]
          self.captions = captions
```

Now that our caption generator has been implemented, it is time to put it into action! For the purpose of testing our caption generator, we have downloaded several images that are completely new and not present in the `Flickr8K` dataset. We downloaded specific images from Flickr that adhere to necessary commercial-usage-based licenses so that we can depict them in this book. We'll show some demonstrations in the next section.

Captioning sample images from outdoor scenes

We took several images from Flickr that focus on a wide variety of outdoor scenes and used both our image captioning models to generate captions for each image, which is depicted as follows:

```
# load files
import glob
outdoor1_files = glob.glob('real_test/outdoor1/*')


# initialize caption generators and generate captions
cg1 = CaptionGenerator(image_locations=outdoor1_files,
word_to_index_map=word_to_index, index_to_word_map=index_to_word,
                      max_caption_size=max_caption_size,
caption_model=model1, beam_size=3)
cg2 = CaptionGenerator(image_locations=outdoor1_files,
word_to_index_map=word_to_index, index_to_word_map=index_to_word,
                      max_caption_size=max_caption_size,
caption_model=model2, beam_size=3)
cg1.initialize_model()
cg1.process_images()
```

```
cg1.generate_captions()
cg2.initialize_model()
cg2.process_images()
cg2.generate_captions()

model30ep_captions_outdoor1 = cg1.captions
model50ep_captions_outdoor1 = cg2.captions

# plot images and their captions
fig=plt.figure(figsize=(13, 11))
plt.suptitle('Automated Image Captioning: Outdoor Scenes 1',
verticalalignment='top', size=15)
columns = 2
rows = 3
for i in range(1, columns*rows +1):
    fig.add_subplot(rows, columns, i)
    image_name = outdoor1_files[i-1]
    img = image.load_img(image_name)
    plt.imshow(img, aspect='auto')
    modelep30_caption_text = 'Caption(ep30): '+
model30ep_captions_outdoor1[i-1]
    modelep50_caption_text = 'Caption(ep50): '+
model50ep_captions_outdoor1[i-1]
    plt.xlabel(modelep30_caption_text+'\n'+modelep50_caption_text,size=11,
wrap=True)
fig.tight_layout()
plt.subplots_adjust(top=0.955)
```

The output of the preceding code is as follows:

Based on the preceding images, you can clearly see that it has correctly identified each scene. It is not a perfect model because we can clearly see it hasn't identified the dogs in the second image in the second row, beside the group of people it has clearly identified. Also, our model does make some color-identification mistakes, such as identifying a green ball as a red ball. Overall, the generated captions are definitely applicable to the source images!

The following images have been captioned from more diverse outdoor scenes and are based on popular outdoor activities. We'll focus on different activities to see how well our model performs on different types of scenes instead of focusing on only one specific scene:

In the preceding images, we have focused on a wide variety of outdoor activities, including dirt biking, skiing, surfing, kayaking, and rock climbing. If you look at the generated captions, they are relevant to each scene and describe them quite well. In several cases, our models get really specific and even describe what each person is wearing. However as we mentioned before, it misidentifies colors in several scenarios, which could perhaps be improved with some more data as well as training on higher-resolution images.

Captioning sample images from popular sports

For the last part of our model test, we took several images from Flickr that focus on a wide variety of sports that are typically played around the world. We definitely got some interesting results since we did not focus on only one or two sports-based scenes. The code for generating this is exactly same as what we used in the previous section with only the source images changing. As always, detailed code is available in the notebooks for reference. Here are the results from our caption generators on the first batch of sports scenes:

Automated Image Captioning: Sports Scenes 1

Caption(ep30): a baseball player is playing soccer
Caption(ep50): a baseball player in a white uniform is playing baseball

Caption(ep30): a man in a blue uniform is holding a soccer ball
Caption(ep50): two men playing basketball

Caption(ep30): a football player in a red jersey is playing football
Caption(ep50): a football player in red is being tackled by a player in a white jersey

Caption(ep30): the hockey player is racing down the ice
Caption(ep50): a hockey player looks at the goal

Caption(ep30): a race car is driving around a car
Caption(ep50): a race car is driving on a dirt track

Caption(ep30): a baseball player in a white uniform is playing tennis
Caption(ep50): a tennis player in a white uniform is playing tennis

In the preceding images, we can clearly see that the model that was trained with 50 epochs out-performs the model with 30 epochs in terms of visually describing the images in a more detailed way. This includes specific jersey and clothing colors such as white, blue, and red. We also see specific activities being mentioned in the captions, such as tacking in football, looking at the goal in hockey, or driving on a dirt track for racing. This definitely gives more depth and meaning to the generated captions. Our model with 30 epochs also makes some mistakes with regard to the exact sports being played in some of the images.

Let's now look at the final set of sports scenes to understand how our caption generators perform on several totally different sports activities from the previous set:

We can observe from the preceding output that both our models perform well, with the model trained on 30 epochs doing quite well in several scenarios, such as identifying children or boys playing soccer or even the color and accessories of the BMX bikers in a race. Overall, both models perform well and explain the scenery to an extent that is similar to how a human would describe these scenes.

The major area of success here is that our model not only identified each activity correctly, but it was also able to generate captions that were meaningful and applicable. We encourage you to try building and testing your own caption generator on different scenes!

Future scope for improvement

There are various ways to improve this model based on the approach we took in this chapter. Here are some of the specific aspects that could be improved upon:

- Using a better image feature extraction model, such as Google's Inception model
- Higher-resolution and better-quality training images (needs for GPU power!)
- More training data based on datasets such as Flickr30K or even image augmentation
- Introducing attention in models

If you have the necessary data and infrastructure, these are some ideas worth exploring!

Summary

This was definitely one of the toughest real-world problems we tackled in the entire book. It was a perfect combination of transfer learning and generative deep learning being applied on a combination of data from images and text that combine different domains around computer vision and NLP. We covered essential concepts around understanding image captioning, the major components needed to build a caption generator, and built our own model from scratch. We made effective use of transfer learning principles by leveraging pretrained computer vision models to extract the right features from images to be captioned and then coupled them with some sequential models, such as LSTMs, to generate captions. The efficient and effective evaluation of sequential models is tough and we leveraged the industry standard BLEU score metric for our purpose. We implemented a scoring function from scratch and evaluated our models on the test dataset.

Finally, we built a generic automated image-captioning system from scratch by using all our previously-built assets and components, and tested it on a wide variety of images from diverse domains. We hope this gives you a good introduction into the world of image-captioning, which is a beautiful combination of computer vision and NLP, and we definitely encourage you to build your own image-captioning system!

12
Image Colorization

Colors are smiles of nature.

– Leigh Hunt

The world was captured in black and white until the 1840s. With a Nobel prize in physics in 1908, Gabriel Lippmann started the age of color captures. It was in 1935 when Eastman Kodak came out with an integral tripack color film, called *Kodachrome*, to capture color photographs.

Color images are not just about aesthetics and beauty, they capture a whole lot more information than black and white images. Color is an important property of real-world objects and it adds another dimension to our perception of the world around us. The importance of colors is such that there have been a number of projects to color historical works of art and photography throughout history. With the advent of tools such as Adobe Photoshop and GIMP, people have been painstakingly turning old photographs into color ones. The reddit r/Colorization subgroup is an online community where people share their experience and work on transforming black and white images into color ones.

So far in this book, we have covered different domains and scenarios to showcase the amazing benefits of transfer learning. In this chapter, we will introduce the concept of image colorization using deep learning and utilize transfer learning to improve upon the results. This chapter will cover the following topics:

- The problem statement
- Understanding image colorization
- Color images
- Building deep neural network-based colorization networks
- Improvements
- Challenges

In the coming sections, we will be using the terms black and white, monochrome, and grayscale to refer to images captured without any color information. We will use these terms interchangeably.

Problem statement

Photographs help us preserve events in time. They don't just help us relive memories but also provide insights into important events from the past. Until color photography became mainstream, our photographic history was captured in black and white. The task of image colorization is to transform a given grayscale image into a plausible color version.

The task of image colorization can be undertaken from different perspectives. The manual process is very time-consuming and requires amazing skills (see the **r/Colorization** subreddit at `https://www.reddit.com/r/Colorization/`). Researchers in the field of computer vision and deep learning have been working on different ways of automating the process. Through this chapter, we will work toward understanding how a deep neural network can be leveraged for such a task. We will also try to utilize the power of transfer learning to improve upon the results.

We encourage readers to give the problem statement a thought before proceeding. Think about how you would approach such a task. Before we dive into the solution, let's get some information about color images and associated concepts. The following section covers the basic concepts required to work on the task at hand.

Color images

Less than 100 years ago, monochrome capture was a limitation, not a choice. The advent of digital and mobile photography has rendered black and white, or grayscale, images an artistic choice. Surely such images have a dramatic effect, yet there is more to black and white images than just changing an option on the capturing device (be it a digital camera or a phone).

Our understanding of colors and formal color models predates color images. Thomas Young, in 1802, postulated the presence of three types of photoreceptors or cone cells (as shown in the following image). His theory detailed how each one these three cone cells are sensitive to only a particular range of visible light. The theory was further developed to classify these cone cells into short, middle, and long, preferring or blue, green, and red, respectively:
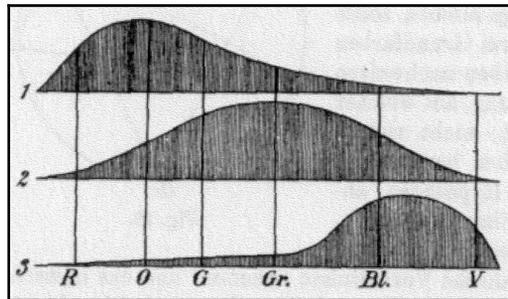
Thomas Young and Hermann Helmholtz: Three Cone Cell Theory (Source: https://en.wikipedia.org/wiki/Color_space#/media/File:YoungHelm.jpg)

The advancements in our understanding of colors and how we perceive them led to the formalization of color theory. Since color theory is a complete field of its own, for the purposes of this chapter, we will look at it briefly. A detailed discussion on these topics is beyond the scope of this book.

Color theory

Color theory, in simple terms, is a formal framework for guidance on color perception, mixing, matching, and methods for replication. There have been different attempts over the years to formally define colors based on color wheels, primary colors, secondary colors, and so on. Color theory is thus a broader field under which we formally define attributes associated with colors like chroma, hue, formulae, and so on.

Color models and color spaces

Color models are formulations of color theory to color representation. A color model is an abstract mathematical concept which, when associated with a precise understanding of its components, is termed a **color space**. Most color models are represented as tuples of three to four numbers representing specific color component.

RGB

A continuation of Thomas Young's tri-cone theory, **Red Green Blue** (**RGB**), is one of the oldest and most widely used color models and color spaces. RGB is an additive color model. In this model, the three components of light—red, green, and blue—are added in different concentrations to achieve the complete spectrum of visible colors. The additive colorspace is as shown in the following diagram:
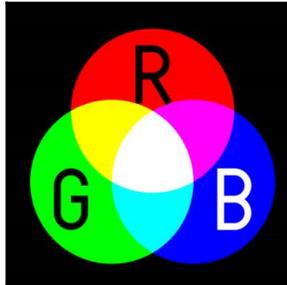
RGB Colorspace (Source: By SharkD at English Wikipedia. Later versions were uploaded by Jacobolus Transferred from en.wikipedia to Commons.Public Domain, https://commons.wikimedia.org/w/index.php?curid=2529435)

Zero intensity of each component results in black, while full intensity leads to the perception of the color white. Despite its simplicity, this color model and color space forms the basis of most electronic displays, including CRTs, LCDs, and LEDs.

YUV

The **Y** stands for **luminance**, while the **U** and **V** channels stand for **chrominance**. This encoding scheme is utilized widely in video systems to map human color perception. The UV channels mainly assist in determining relative amounts of red and blue colors. This scheme was widely used due to its capability of using a lower bandwidth and being less prone to transmission errors, illustrated as follows:
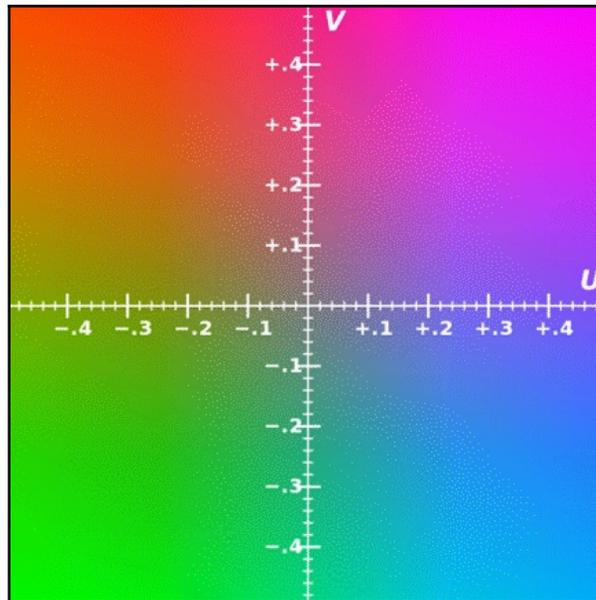
YUV colorospace (Source: By Tonyle—Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6977944)

This image is a sample representation of UV color channels at 0.5 Y.

LAB

This device-independent colorspace reference was developed by the International Commission on Illumination. The **L** channel represents lightness of color (0 is black, while 100 is diffuse white).

A represents positions between green and magenta, while **B** represents positions between blue and yellow, illustrated as follows:
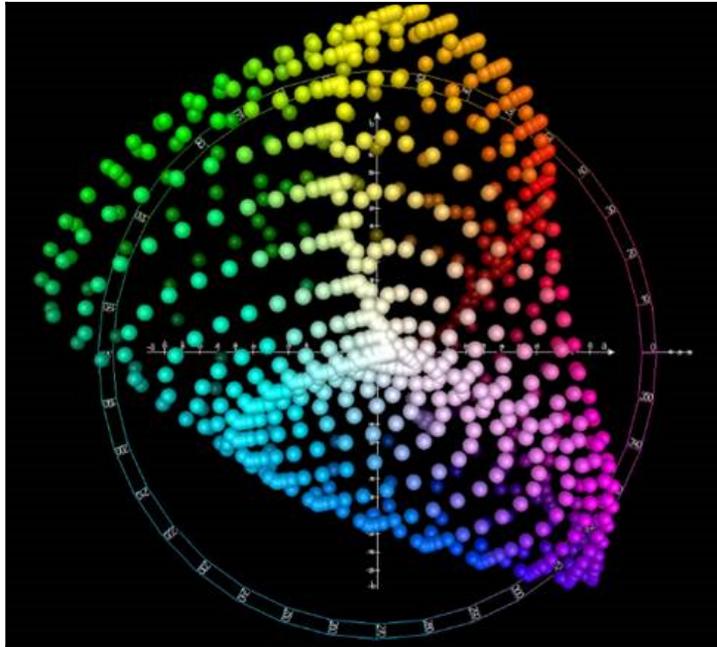
LAB colorspace (Source: By Holger kkk Everding – Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38366968)

Apart from these three, there are various other color models in existence. For the purposes of our current use case around image colorization, we will take a rather interesting approach.

Problem statement revisited

If we follow the most widely used color model, RGB, it turns out that training a model to map an input monochrome image to a color one would be quite a daunting task.

Researchers in the deep learning space have been quite creative when it comes to approaching and formulating a problem. In the case of image colorization, researchers have cleverly looked into ways of utilizing different input to achieve realistic color hallucinations of grayscale images.

In the initial attempts, different variants of color-guided input, in the form of reference images and color scribbles, were used to a great effect; see Welsh and co-authors as well as Levin and co-authors.

Recent works have focused on automating the entire process by utilizing transfer learning in deep CNNs. The results have been quite encouraging and sometimes good enough to fool humans.

Recent works, together with the power of transfer learning, have cleverly tried to utilize color models that contain the grayscale channel as one of its components. Does that ring a bell? Let's now look at the problem statement from a different perspective.

Apart from the omnipresent RGB colorspace, we discussed LAB as well. The LAB colorspace contains grayscale values as the *L* channel (for luminance), while the remaining two channels (*a* and *b*) impart color attributes. Thus, the colorization problem can be modeled mathematically as:

$$\mathcal{F} : \chi_L \rightarrow (\widetilde{\chi}_a, \widetilde{\chi}_b)$$

In the aforementioned equation, we denote a function that maps the *L* channel to the *a* and *b* channels of the same image from the given data. This is illustrated in the following image:
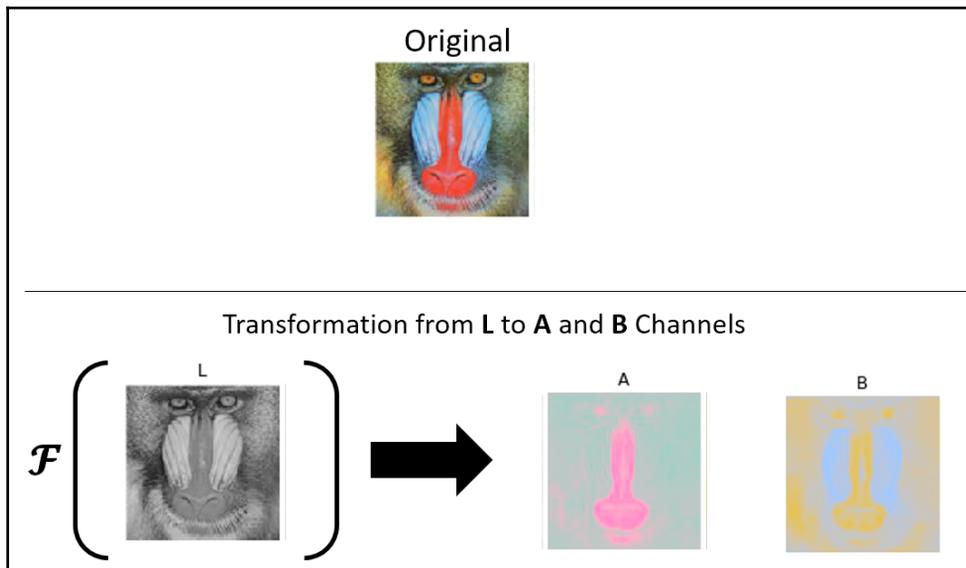
The colornet transformation

In simple words, we have transformed the task of image colorization into that of transforming one channel (grayscaled **L** channel) into two color channels (**A** and **B**), illustrated as follows:
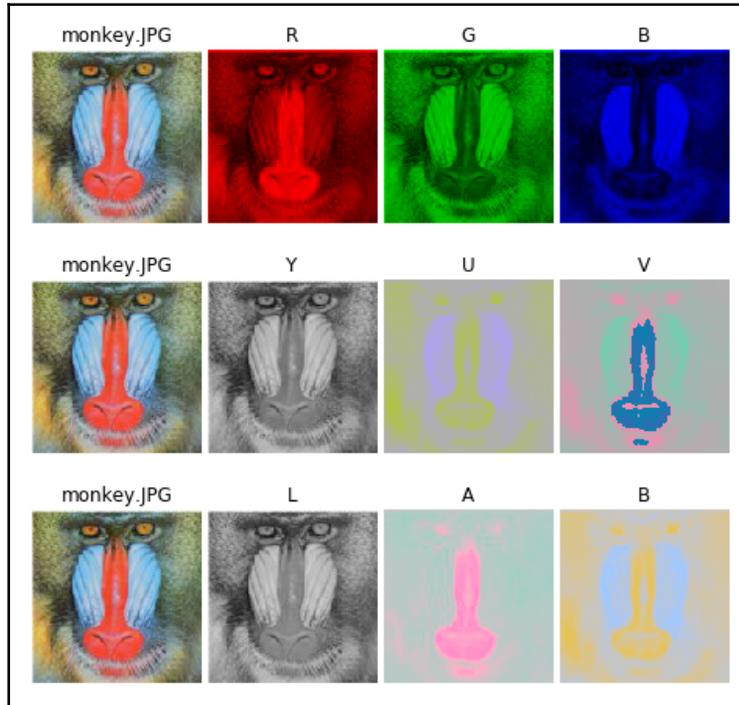
Color image and its components—RGB, YUV, and LAB

The preceding image shows the **L**, **A**, and **B** channels of a color image. Works by Zhang and co-authors (2016) and Federico and his co-authors (2017) (`https://arxiv.org/abs/1603.08511`) are based on a similar premise. We will study them in detail in coming sections.

> We encourage readers to go through the paper titled *Deep Koalarization: Image Colorization using CNNs and Inception-ResNet-v2* (`https://arxiv.org/abs/1712.03400`). We would like to thank Federico Baldassarre, Diego Gonzalez-Morin, and Lucas Rodes-Guirao for providing detailed information and insights into their work and its implementation. We would also like to thank Emil Wallner for an amazingly wonderful implementation of this paper using Keras.

Readers should note that a similar process can be applied to YUV colorspace as well. Attempts at utilizing this colorspace has also worked out well, as discussed in the paper titled Image Colorization with Deep Convolutional Neural Networks by Jeff Hwang and his co-authors (`http://cs231n.stanford.edu/reports/2016/pdfs/219_Report.pdf`).

Building a coloring deep neural network

It's time to build a coloring deep neural network or colornet. As discussed in the previous section, if we utilize an alternate colorspace, such as LAB (or YUV), we can transform the colorization task into a mathematical transformation. The transformation is as follows:

$$\mathcal{F} : \chi_L \rightarrow (\widetilde{\chi}_a, \widetilde{\chi}_b)$$

Mathematical formulations and creativity are fine, but where are the images to learn these transformations? Deep learning networks are data hungry, but luckily, we have a huge collection of diverse images from various open source datasets. For the purposes of this chapter, we will rely on a few sample images from ImageNet itself. Since ImageNet is a huge dataset, we have randomly selected a few color images for our problem statement. In later sections, we will discuss why we selected this subset and a few of its nuances.

> We relied upon the image-extraction utility developed by Baldassarre and his co-authors for their *Deep Koalarization: Image Colorization using CNNs and Inception-ResNet-v2* paper (`https://arxiv.org/abs/1712.03400`) to curate the subset of ImageNet samples used in this chapter. The code for data-extraction is available at `https://github.com/baldassarreFe/deep-koalarization/tree/master/dataset`.

The code and sample images used for this chapter along with the `colornet_vgg16.ipynb` notebook are available in the GitHub repository for this book.

Preprocessing

The first step after obtaining/curating the required dataset is preprocessing. For this current task of image colorization, we need to perform the following preprocessing steps:

- **Rescaling**: ImageNet is a diverse dataset with all kinds of images, in terms of classes and sizes (dimensions). For the purpose of this implementation, we rescale all images to a fixed size.

- **Utilize 24-bit RGB**: Since the human eye can distinguish between 2 and 10 million colors only, we can utilize 24-bit RGB to approximate 16 million colors. Reducing the number of bits per channel would help us train our models faster with fewer resources. This can be achieved by simply dividing the pixel values by 255.
- **RGB to LAB**: Since image colorization is easier to solve in LAB colorspace, we would utilize skimage to transform and extract LAB channels from RGB images.

Standardization

The LAB color space has values from -128 to +128. Since neural networks are sensitive to the scale of input values, we normalize the transformed pixel values from -128 to +128 and bring them within the -1 to +1 range. The same is showcased in the following code snippet:

```
def prep_data(file_list=[],
              dir_path=None,
              dim_x=256,
              dim_y=256):

    #Get images

    X = []
for filename in file_list:
    X.append(img_to_array(
                        sp.misc.imresize(
                        load_img(
                        dir_path+filename),
                        (dim_x, dim_y))

        )

    )
    X = np.array(X, dtype=np.float64)
    X = 1.0/255*X
    return X
```

Once transformed, we then split the data into train and test sets. For splitting, we utilize `train_test_split utility` from sklearn.

Loss function

A model learns by improving upon the loss function or the objective function. The task is to learn optimal parameters using backpropagation to minimize the difference between the original color image and the output of the model. The output color image from the model is also referred to as the hallucinated colorization of the grayscale image. In this implementation, we utilize the **mean squared error** (**MSE**) as our loss function. The following equation summarizes it:

$$C(X, \theta) = \frac{1}{2HW} \sum_{k \epsilon \{a,b\}} \sum_{i=1}^{H} \sum_{j=1}^{W} (X_{ki,j} - \widetilde{X}_{ki,j})^2$$

Loss function between original color and colornet output (Source: Baldassarre and co-author)

In the case of Keras, using this loss function is as easy as setting a parameter while compiling the Keras model. We utilize the RMSprop optimizer to train our model (the paper uses Adam instead).

Encoder

Convolutional Neural Networks (**CNNs**) are amazing image classifiers. They do so by extracting location-invariant features. In this process, they tend to distort the input image.

In the case of image colorization, such a distortion would be disastrous. To this end, we use an encoder to transform our input grayscale image of *H x W* dimensions to *H/8 x W/8*. The encoder maintains the aspect ratio of the image through different layers by using zero padding. The following snippet shows the described encoder using Keras:

```
#Encoder
enc_input = Input(shape=(DIM, DIM, 1,))
enc_output = Conv2D(64, (3,3),
                activation='relu',
                padding='same', strides=2)(enc_input)
enc_output = Conv2D(128, (3,3),
                activation='relu',
                padding='same')(enc_output)
enc_output = Conv2D(128, (3,3),
                activation='relu',
                padding='same', strides=2)(enc_output)
```

```
enc_output = Conv2D(256, (3,3),
               activation='relu',
               padding='same')(enc_output)
enc_output = Conv2D(256, (3,3),
               activation='relu',
               padding='same', strides=2)(enc_output)
enc_output = Conv2D(512, (3,3),
               activation='relu',
               padding='same')(enc_output)
enc_output = Conv2D(512, (3,3),
               activation='relu',
               padding='same')(enc_output)
enc_output = Conv2D(256, (3,3),
               activation='relu',
               padding='same')(enc_output)
```

In the preceding snippet, the interesting aspect is the use of a stride size of 2 for layers 1, 3, and 5. The stride length of 2 halves the image dimension but still manages to maintain the aspect ratio. This helps to increase the information density without distorting the original image.

Transfer learning – feature extraction

The image colorization network discussed in this chapter is quite a unique one. Its uniqueness comes from the way we use transfer learning to enhance our model. We understand that pretrained networks can be leveraged as feature extractors to help transfer the learned patterns and to boost our model's performance.

In this current setting, we utilize a pretrained VGG16 (the paper refers to utilizing a pretrained Inception model) for transfer learning purposes. Since VGG16 requires input in a specific format, we transform the input grayscale image (the same grayscale image that is input into the encoder part of the network) by resizing it and concatenating the same image three times to compensate for missing channel information.

The following snippet takes the input grayscale image and produces the required embedding:

```
#Create embedding
def create_vgg_embedding(grayscaled_rgb):
    gs_rgb_resized = []
    for i in grayscaled_rgb:
        i = resize(i, (224, 224, 3),
                    mode='constant')
        gs_rgb_resized.append(i)
    gs_rgb_resized = np.array(gs_rgb_resized)
    gs_rgb_resized = preprocess_input(gs_rgb_resized)
    with vgg16.graph.as_default():
      embedding = vgg16.predict(gs_rgb_resized)
    return embedding
```

The preceding snippet generates an output feature vector of size 1,000 x 1 x 1.

Fusion layer

Most of the networks we have built across previous chapters have utilized the sequential API from Keras. Fusion layer is an innovative way to utilize transfer learning in this context. Remember that we have utilized the input grayscale image as input for two different networks, an encoder and a pretrained VGG16. Since the output of both networks is of different shapes, we repeat the output from VGG16 by 1,000 times and concatenate or perform a fusion with the encoder output. The following snippet prepares the fusion layer:

```
#Fusion
fusion_layer_output = RepeatVector(32*32)(emd_input)
fusion_layer_output = Reshape(([32,32,
                        1000]))(fusion_layer_output)
fusion_layer_output = concatenate([enc_output,
                              fusion_layer_output], axis=3)
fusion_layer_output = Conv2D(DIM, (1, 1),
                      activation='relu',
                      padding='same')(fusion_layer_output)
```

The repetition of output from VGG16 is attached along the depth axis of the encoder output. This ensures the feature embedding of the image, as extracted from VGG16, is evenly spread across the whole image:
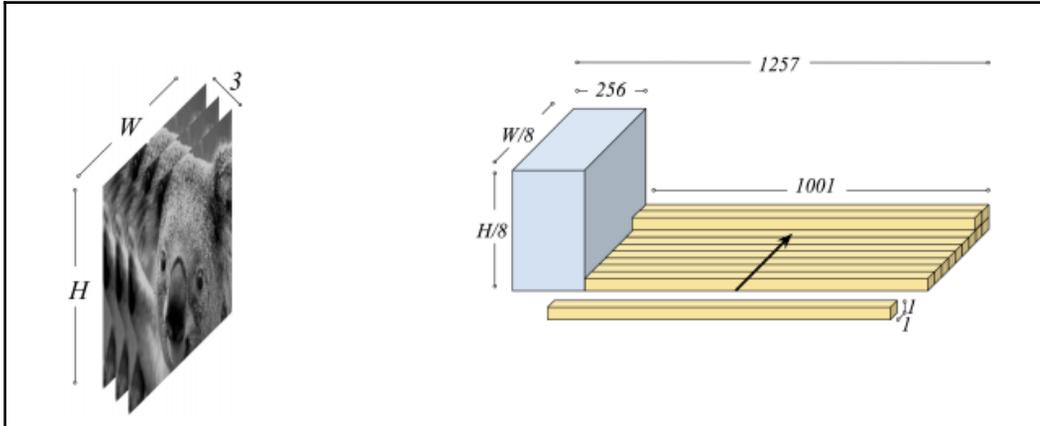
Concatenated grayscale input the pretrained network (left), Fusion layer (right)
Source: Baldassarre et al

The input to the feature extractor or the pretrained VGG16 and the structure of the fusion layer are shown in the preceding image.

Decoder

The final stage of the network is a decoder. In the first two sections of the network, we utilized an encoder and a pretrained model to learn different features and generate an embedding. The output from the fusion layer is a tensor of size H/8 x W/8 x 256, where *H* and *W* are the original height and width of the grayscale image (in our case, it was 256 x 256). This input is passed through an eight-layer decoder built using five convolutional and three upsampling layers. The upsampling layers help us to double the size of image using a basic nearest-neighbor approach. The following snippet showcases the decoder section of the network:

```
#Decoder
dec_output = Conv2D(128, (3,3),
                    activation='relu',
                    padding='same')(fusion_layer_output)
dec_output = UpSampling2D((2, 2))(dec_output)
dec_output = Conv2D(64, (3,3),
                    activation='relu',
                    padding='same')(dec_output)
```

```
dec_output = UpSampling2D((2, 2))(dec_output)
dec_output = Conv2D(32, (3,3),
                        activation='relu',
                        padding='same')(dec_output)
dec_output = Conv2D(16, (3,3),
                        activation='relu',
                        padding='same')(dec_output)
dec_output = Conv2D(2, (3, 3),
                        activation='tanh',
                        padding='same')(dec_output)
dec_output = UpSampling2D((2, 2))(dec_output)
```

The output of the decoder network is an image of the original size with two channels, that is, the output is a tensor of shape *H x W x 2*. The final convolutional layer uses an activation function of tanh to keep the predicted pixel values in the -1 to +1 range.

The network, with its three components, is as shown in the following diagram:
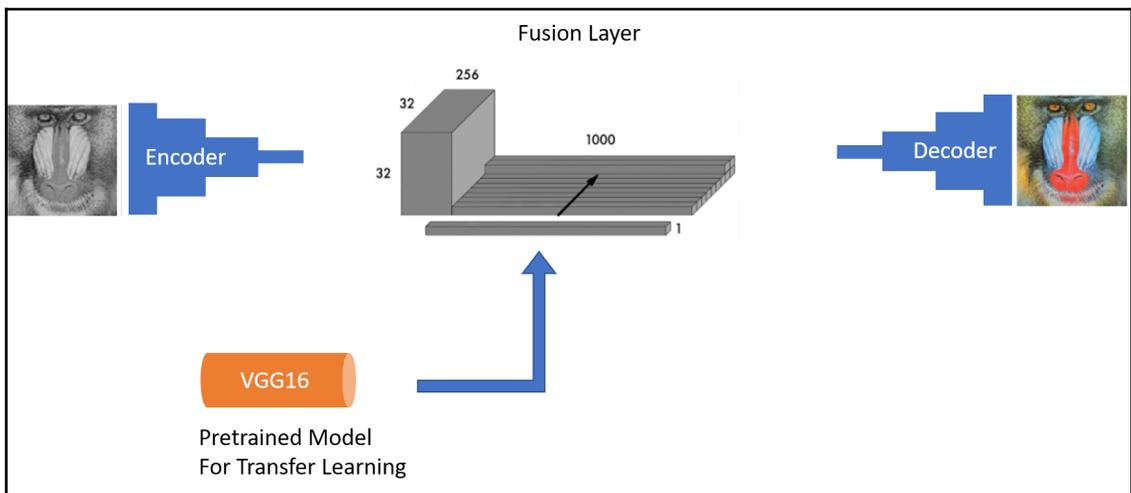
Colornet consisting of Encoder, pretrained model as feature extractor, fusion layer, and a decoder

Deep learning models built using Keras are usually built using the Sequential API. In this case, our coloring network, or colornet, utilizes the functional API to be able to implement the fusion layer.

Postprocessing

The tricks in solving the problem aren't over yet. As mentioned in the *Preprocessing* subsection, we standardize the pixel values between -1 to +1 for the purpose of ensuring that our network trains properly. Also, the LAB colorspace has values between -128 and +128 for the two color channels. Thus, the following two postprocessing steps are performed:

- We multiply each pixel value by 128 to bring values into the required range of color channels
- We concatenate the grayscale input image with the output two-channel image to achieve the hallucinated color image

The following snippet performs the postprocessing steps to achieve hallucinated color images:

```
sample_img = []
for filename in test_files:
    sample_img.append(sp.misc.imresize(load_img(IMG_DIR+filename),
                                      (DIM, DIM)))
sample_img = np.array(sample_img,
                      dtype=float)
sample_img = 1.0/255*sample_img
sample_img = gray2rgb(rgb2gray(sample_img))
sample_img = rgb2lab(sample_img)[:,:,:,0]
sample_img = sample_img.reshape(sample_img.shape+(1,))
#embedding input
sample_img_embed = create_vgg_embedding(sample_img)
```

As shown in the preceding code snippet, we use the `lab2rgb` utility from skimage to transform the produced output into RGB colorspace. This is done for the ease of visualizing the output image.

Training and results

Training such a complex network can be a tricky thing. For the purposes of this chapter, we have chosen a very small subset of images from ImageNet. To help our network learn and generalize, we use the `ImageDataGenerator` class from Keras to augment the dataset and produce variation in the input dataset. The following snippet showcases image augmentation and model training:

```
# Image transformer
datagen = ImageDataGenerator(
        shear_range=0.2,
        zoom_range=0.2,
        rotation_range=20,
        horizontal_flip=True)
def colornet_img_generator(X,
                  batch_size=BATCH_SIZE):
    for batch in datagen.flow(X, batch_size=batch_size):
        gs_rgb = gray2rgb(rgb2gray(batch))
        batch_lab = rgb2lab(batch)

        batch_l = batch_lab[:,:,:,0]
        batch_l = batch_l.reshape(batch_l.shape+(1,))
        batch_ab = batch_lab[:,:,:,1:] / 128
        yield ([batch_l,
               create_vgg_embedding(gs_rgb)], batch_ab)
history = model.fit_generator(colornet_img_generator(X_train,
                                              BATCH_SIZE),
                              epochs=EPOCH,
                              steps_per_epoch=STEPS_PER_EPOCH)
```

The loss can be a bit misleading in the case of coloring networks. It may seem to have stabilized in under 100 epochs, yet the results produced were more in sepia than color. We thus experimented a bit more to achieve the following results:

Colornet output: first column represents grayscale input, second column is the model output, while third column is original image

The preceding results, though not breathtaking, are encouraging enough. The aforementioned results were achieved by training the model for 600 epochs with a batch size of 64.

Challenges

Deep neural networks are extremely powerful models with hundreds and thousands of learnable parameters. The current scenario of training a coloring network presents a new set of challenges, some of which are discussed as follows:

- The current network seems to have learned high-level features, such as grass and sports jerseys (to a certain extent), while it found learning color patterns for smaller objects a bit too difficult.
- The training set was limited to a very specific subset of images and hence that is reflected in the test dataset. The model has poor performance on objects that are either not present in the training set or not many samples contain them.
- Even though the training loss seems to have stabilized in under 50 epochs, we see that the model's performance on coloring is quite poor unless trained for a few hundred epochs.
- The model has a high tendency to color most objects gray or sepia. This is observed in models trained for fewer epochs.

Apart from these challenges, the computation and memory requirements are also quite high for such a complex architecture.

Further improvements

The current implementation, though it shows promising results, can be tuned further. Further improvements could be achieved by utilizing a bigger and more diverse dataset.

Improvements could also be made by utilizing an even more powerful state-of-the-art pretrained image classification model, such as InceptionV3 or InceptionResNetV2.

We could also leverage the functional API from Keras by preparing an ensemble network composed of an even more complex architecture. One of the next steps could be to provide temporal information to the network and see whether there is a scope to learn to colorize videos as well.

Summary

Image colorization is one of the leading-edge topics from the deep learning world. As our understanding of transfer learning and deep learning is maturing, the application scope is getting exciting and more creative. Image colorization is an active area of research and lately some exciting work has been shared by deep learning experts.

In this chapter, we learned about color theory, different color models, and color spaces. This understanding helped us reformulate the problem statement to that of mapping from a single-channel grayscale image to a two-channel output. We then worked toward building a colornet based on the works of Baldassarre and his co-authors. The implementation involved a unique three-layer network consisting of an encoder, a decoder, and a fusion layer. The fusion layer allowed us to utilize transfer learning by concatenating VGG16 embeddings with the encoder output. The network required a few specific preprocessing and postprocessing steps to train on the given set of images. Our training and test datasets consisted of a subset of ImageNet samples. We trained our colornet for a few hundred epochs. Finally, we presented a few hallucinated images to understand how well the model has learned the task of coloring. The trained colornet learned certain high-level objects, such as grass, but did not perform well on smaller or more infrequent objects. We also discussed a few challenges this type of network presents.

This concludes the final chapter in a series of use-case-driven chapters in this book. We presented different use cases across different domains. Each of the use cases helped us leverage the concepts of transfer learning, which was discussed in detail in the first two sections of the book. Andrew Ng, one of the leading names in the field of machine learning and deep learning, said during his NIPS 2016 tutorial that:

Transfer learning would be the next driver for the commercial success of machine learning.

With the variety of applications and their advantages discussed and showcased throughout this book, you should now understand the immense potential of transfer learning.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:
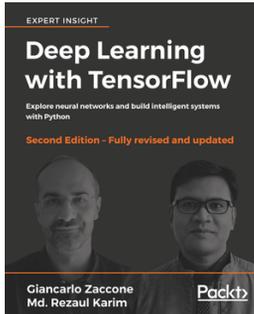
Deep Learning with TensorFlow - Second Edition
Giancarlo Zaccone

ISBN: 9781788831109

- Apply deep machine intelligence and GPU computing with TensorFlow
- Access public datasets and use TensorFlow to load, process, and transform the data
- Discover how to use the high-level TensorFlow API to build more powerful applications
- Use deep learning for scalable object detection and mobile computing
- Train machines quickly to learn from data by exploring reinforcement learning techniques
- Explore active areas of deep learning research and applications
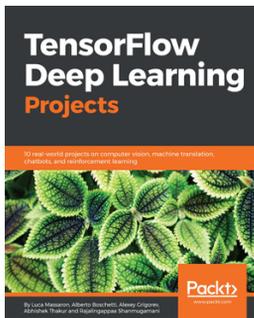
TensorFlow Deep Learning Projects

Luca Massaron

ISBN: 9781788398060

- Set up the TensorFlow environment for deep learning
- Construct your own ConvNets for effective image processing
- Use LSTMs for image caption generation
- Forecast stock prediction accurately with an LSTM architecture
- Learn what semantic matching is by detecting duplicate Quora questions
- Set up an AWS instance with TensorFlow to train GANs
- Train and set up a chatbot to understand and interpret human input
- Build an AI capable of playing a video game by itself –and win it!

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