Python Deep Learning

Second Edition

Exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow

Packt
www.packt.com

Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants and Valentino Zocca

Python Deep Learning
Second Edition

Exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow

Ivan Vasilev
Daniel Slater
Gianmario Spacagna
Peter Roelants
Valentino Zocca

Packt

BIRMINGHAM - MUMBAI

Python Deep Learning
Second Edition

Commissioning Editor: Pravin Dhandre
Acquisition Editor: Yogesh Deokar
Content Development Editor: Nathanya Dias
Technical Editor: Kushal Shingote
Copy Editor: Safis Editing
Project Coordinator: Kirti Pisat
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Jisha Chirayil
Production Coordinator: Priyanka Dhadke

First published: October 2016
Second edition: January 2019

Production reference: 1110119

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

ISBN 978-1-78934-846-0

`www.packtpub.com`

Contributors

About the authors

Ivan Vasilev started working on the first open source Java Deep Learning library with GPU support in 2013. The library was acquired by a German company, where he continued its development. He has also worked as machine learning engineer and researcher in the area of medical image classification and segmentation with deep neural networks. Since 2017 he has focused on financial machine learning. He is working on a Python open source algorithmic trading library, which provides the infrastructure to experiment with different ML algorithms. The author holds an MSc degree in Artificial Intelligence from The University of Sofia, St. Kliment Ohridski.

Daniel Slater started programming at age 11, developing mods for the id Software game Quake. His obsession led him to become a developer working in the gaming industry on the hit computer game series Championship Manager. He then moved into finance, working on risk- and high-performance messaging systems. He now is a staff engineer working on big data at Skimlinks to understand online user behavior. He spends his spare time training AI to beat computer games. He talks at tech conferences about deep learning and reinforcement learning; his blog can be found at `www.danielslater.net`. His work in this field has been cited by Google.

Gianmario Spacagna is a senior data scientist at Pirelli, processing sensors and telemetry data for **internet of things** (**IoT**) and connected-vehicle applications. He works closely with tire mechanics, engineers, and business units to analyze and formulate hybrid, physics-driven, and data-driven automotive models. His main expertise is in building ML systems and end-to-end solutions for data products. He holds a master's degree in telematics from the Polytechnic of Turin, as well as one in software engineering of distributed systems from KTH, Stockholm. Prior to Pirelli, he worked in retail and business banking (Barclays), cyber security (Cisco), predictive marketing (AgilOne), and did some occasional freelancing.

Peter Roelants holds a master's in computer science with a specialization in AI from KU Leuven. He works on applying deep learning to a variety of problems, such as spectral imaging, speech recognition, text understanding, and document information extraction. He currently works at Onfido as a team leader for the data extraction research team, focusing on data extraction from official documents.

Valentino Zocca has a PhD degree and graduated with a Laurea in mathematics from the University of Maryland, USA, and University of Rome, respectively, and spent a semester at the University of Warwick. He started working on high-tech projects of an advanced stereo 3D Earth visualization software with head tracking at Autometric, a company later bought by Boeing. There he developed many mathematical algorithms and predictive models, and using Hadoop he automated several satellite-imagery visualization programs. He has worked as an independent consultant at the U.S. Census Bureau, in the USA and in Italy. Currently, Valentino lives in New York and works as an independent consultant to a large financial company.

About the reviewer

Greg Walters, since 1972, has been involved with computers and computer programming. Currently, he is extremely well versed in Visual Basic, Visual Basic .NET, Python, and SQL using MySQL, SQLite, Microsoft SQL Server, and Oracle. He also has experience in C++, Delphi, Modula-2, Pascal, C, 80x86 Assembler, COBOL, and Fortran.

He is a programming trainer and has trained numerous people in the use of various computer software packages, such as MySQL, Open Database Connectivity, Quattro Pro, Corel Draw!, Paradox, Microsoft Word, Excel, DOS, Windows 3.11, Windows for Workgroups, Windows 95, Windows NT, Windows 2000, Windows XP, and Linux.

He is currently retired and in his spare time, he is a musician, loves to cook, and lives in central Texas.

Table of Contents

Preface

With the surge in artificial intelligence in applications catering to both business and consumer needs, deep learning is more important than ever for meeting current and future market demands. With this book, you'll explore deep learning, and learn how to put machine learning to use in your projects.

This second edition of Python Deep Learning will get you up to speed with deep learning, deep neural networks, and how to train them with high-performance algorithms and popular Python frameworks. You'll uncover different neural network architectures, such as convolutional networks, recurrent neural networks, **long short-term memory** (**LSTM**) networks, and capsule networks. You'll also learn how to solve problems in the fields of computer vision, **natural language processing** (**NLP**), and speech recognition. You'll study generative model approaches such as variational autoencoders and **Generative Adversarial Networks** (**GANs**) to generate images. As you delve into newly evolved areas of reinforcement learning, you'll gain an understanding of state-of-the-art algorithms that are the main components behind popular game Go, Atari, and Dota.

By the end of the book, you will be well-versed with the theory of deep learning along with its real-world applications.

Who this book is for

This book is for data science practitioners, machine learning engineers, and those interested in deep learning who have a basic foundation in machine learning and some Python programming experience. A background in mathematics and conceptual understanding of calculus and statistics will help you gain maximum benefit from this book.

What this book covers

`Chapter 1`, *Machine Learning – an Introduction*, will introduce you to the basic ML concepts and terms that we'll be using throughout the book. It will give an overview of the most popular ML algorithms and applications today. It will also introduce the DL library that we'll use throughout the book.

`Chapter 2`, *Neural Networks*, will introduce you to the mathematics of neural networks. We'll learn about their structure, how they make predictions (that's the feedforward part), and how to train them using gradient descent and backpropagation (explained through derivatives). The chapter will also discuss how to represent operations with neural networks as vector operations.

`Chapter 3`, *Deep Learning Fundamentals*, will explain the rationale behind using *deep* neural networks (as opposed to shallow ones). It will take an overview of the most popular DL libraries and real-world applications of DL.

`Chapter 4`, *Computer Vision with Convolutional Networks*, teaches you about convolutional neural networks (the most popular type of neural network for computer vision tasks). We'll learn about their architecture and building blocks (the convolutional, pooling, and capsule layers) and how to use a convolutional network for an image classification task.

`Chapter 5`, *Advanced Computer Vision*, will build on the previous chapter and cover more advanced computer vision topics. You will learn not only how to classify images, but also how to detect an object's location and segment every pixel of an image. We'll learn about advanced convolutional network architectures and the useful practical technique of transfer learning.

`Chapter 6`, *Generating Images with GANs and VAEs*, will introduce generative models (as opposed to discriminative models, which is what we'll have covered up until this point). You will learn about two of the most popular unsupervised generative model approaches, VAEs and GANs, as well some of their exciting applications.

`Chapter 7`, *Recurrent Neural Networks and Language Models*, will introduce you to the most popular recurrent network architectures: LSTM and **gated recurrent unit** (**GRU**). We'll learn about the paradigms of NLP with recurrent neural networks and the latest algorithms and architectures to solve NLP problems. We'll also learn the basics of speech-to-text recognition.

`Chapter 8`, *Reinforcement Learning Theory*, will introduce you to the main paradigms and terms of RL, a separate ML field. You will learn about the most important RL algorithms. We'll also learn about the link between DL and RL. Throughout the chapter, we will use toy examples to better demonstrate the concepts of RL.

`Chapter 9`, *Deep Reinforcement Learning for Games*, you will understand some real-world applications of RL algorithms, such as playing board games and computer games. We'll learn how to combine the knowledge from the previous parts of the book to create better-than-human computer players on some popular games.

`Chapter 10`, *Deep Learning in Autonomous vehicles,* we'll discuss what sensors autonomous vehicles use, so they can create the 3D model of the environment. These include cameras, radar sensors, ultrasound sensors, Lidar, as well as accurate GPS positioning. We'll talk about how to apply deep learning algorithms for processing the input of these sensors. For example, we can use instance segmentation and object detection to detect pedestrians and vehicles using the vehicle cameras. We'll also make an overview of some of the approaches vehicle manufacturers use to solve this problem (for example Audi, Tesla, and so on).

To get the most out of this book

To get the most out of this book, you should be familiar with Python. You'd benefit from some basic knowledge of calculus and statistics. The code examples are best run on a Linux machine with an NVIDIA GPU capable of running PyTorch, TensorFlow, and Keras.

Download the example code files

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

You can download the code files by following these steps:

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

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

- WinRAR/7-Zip for Windows
- 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/Python-Deep-Learning-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

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: `http://www.packtpub.com/sites/default/files/downloads/9781789348460_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: "We can parameterize this house with a five-dimensional vector, `x = (100, 25, 3, 2, 7)`."

A block of code is set as follows:

```
import torch

torch.manual_seed(1234)

hidden_units = 5

net = torch.nn.Sequential(
 torch.nn.Linear(4, hidden_units),
 torch.nn.ReLU(),
 torch.nn.Linear(hidden_units, 3)
)
```

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

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

1
Machine Learning - an Introduction

"Machine Learning (CS229) is the most popular course at Stanford. Why? Because, increasingly, machine learning is eating the world." - Laura Hamilton, Forbes

Machine learning(**ML**) techniques are being applied in a variety of fields, and data scientists are being sought after in many different industries. With machine learning, we identify the processes through which we gain knowledge that is not readily apparent from data in order to make decisions. Applications of machine learning techniques may vary greatly, and are found in disciplines as diverse as medicine, finance, and advertising.

In this chapter, we'll present different machine learning approaches, techniques, some of their applications to real-world problems, and we'll also introduce one of the major open source packages available in Python for machine learning, `PyTorch`. This will lay the foundation for the later chapters in which we'll focus on a particular type of machine learning approach using neural networks, which will aim to emulate brain functionality. In particular, we will focus on deep learning. Deep learning makes use of more advanced neural networks than those used during the 1980s. This is not only a result of recent developments in the theory, but also advancements in computer hardware. This chapter will summarize what machine learning is and what it can do, preparing the reader to better understand how deep learning differentiates itself from popular traditional machine learning techniques.

This chapter will cover the following topics:

- Introduction to machine learning
- Different machine learning approaches
- Neural networks
- Introduction to PyTorch

Introduction to machine learning

Machine learning is often associated with terms such as **big data** and **artificial intelligence** (**AI**). However, both are quite different to machine learning. In order to understand what machine learning is and why it's useful, it's important to understand what big data is and how machine learning applies to it.

Big data is a term used to describe huge datasets that are created as the result of large increases in data that is gathered and stored. For example, this may be through cameras, sensors, or internet social sites.

It's estimated that Google alone processes over 20 petabytes of information per day, and this number is only going to increase. IBM estimated that every day, 2.5 quintillion bytes of data is created, and that 90% of all the data in the world has been created in the last two years (`https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/`).

Clearly, humans alone are unable to grasp, let alone analyze, such huge amounts of data, and machine learning techniques are used to make sense of these very large datasets. Machine learning is the tool used for large-scale data processing. It is well-suited to complex datasets that have huge numbers of variables and features. One of the strengths of many machine learning techniques, and deep learning in particular, is that they perform best when used on large datasets, thus improving their analytic and predictive power. In other words, machine learning techniques, and deep learning neural networks in particular, learn best when they can access large datasets where they can discover patterns and regularities hidden in the data.

On the other hand, machine learning's predictive ability can be successfully adapted to artificial intelligence systems. Machine learning can be thought of as the brain of an AI system. AI can be defined (though this definition may not be unique) as a system that can interact with its environment. Also, AI machines are endowed with sensors that enable them to know the environment they are in, and tools with which they can relate back to the environment. Machine learning is therefore the brain that allows the machine to analyze the data ingested through its sensors to formulate an appropriate answer. A simple example is Siri on an iPhone. Siri hears the command through its microphone and outputs an answer through its speakers or its display, but to do so, it needs to understand what it's being told. Similarly, driverless cars will be equipped with cameras, GPS systems, sonars, and LiDAR, but all this information needs to be processed in order to provide a correct answer. This may include whether to accelerate, brake, or turn. Machine learning is the information-processing method that leads to the answer.

We explained what machine learning is, but what about **deep learning** (**DL**)? For now, let's just say that deep learning is a subfield of machine learning. DL methods share some special common features. The most popular representatives of such methods are deep neural networks.

Different machine learning approaches

As we have seen, the term machine learning is used in a very general way, and refers to the general techniques used to extrapolate patterns from large sets, or it is the ability to make predictions on new data based on what is learned by analyzing available known data. Machine learning techniques can roughly be divided in two large classes, while one more class is often added. Here are the classes:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning

Supervised learning algorithms are a class of machine learning algorithms that use previously-labeled data to learn its features, so they can classify similar but unlabeled data. Let's use an example to understand this concept better.

Let's assume that a user receives a large amount of emails every day, some of which are important business emails and some of which are unsolicited junk emails, also known as spam. A supervised machine algorithm will be presented with a large body of emails that have already been labeled by a teacher as spam or not spam (this is called **training data**). For each sample, the machine will try to predict whether the email is spam or not, and it will compare the prediction with the original target label. If the prediction differs from the target, the machine will adjust its internal parameters in such a way that the next time it encounters this sample it will classify it correctly. Conversely, if the prediction was correct, the parameters will stay the same. The more training data we feed to the algorithm, the better it becomes (this rule has caveats, as we'll see next).

In the example we used, the emails had only two classes (spam or not spam), but the same principles apply for tasks with arbitrary numbers of classes. For example, we could train the software on a set of labeled emails where the classes are Personal, Business/Work, Social, or Spam.

In fact, Gmail, the free email service by Google, allows the user to select up to five categories, which are labeled as the following:

- **Primary**: Includes person-to-person conversations
- **Social**: Includes messages from social networks and media-sharing sites
- **Promotions**: Includes marketing emails, offers, and discounts
- **Updates**: Includes bills, bank statements, and receipts
- **Forums**: Includes messages from online groups and mailing lists

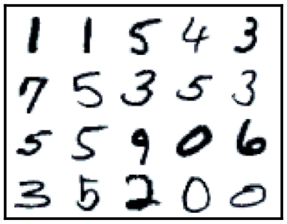
In some cases, the outcome may not necessarily be discrete, and we may not have a finite number of classes to classify our data into. For example, we may try to predict the life expectancy of a group of people based on their predetermined health parameters. In this case, the outcome is a continuous function, that is, the number years the person is expected to live, and we don't talk about classification but rather regression.

One way to think of supervised learning is to imagine we are building a function, *f*, defined over a dataset, which comprises information organized by **features**. In the case of email classification, the features can be specific words that may appear more frequently than others in spam emails. The use of explicit sex-related words will most likely identify a spam email rather than a business/work email. On the contrary, words such as *meeting*, *business*, or *presentation* are more likely to describe a work email. If we have access to metadata, we may also use the sender's information as a feature. Each email will then have an associated set of features, and each feature will have a value (in this case, how many times the specific word is present in the email body). The machine learning algorithm will then seek to map those values to a discrete range that represents the set of classes, or a real value in the case of regression. The definition of the *f* function is as follows:

$$f : \text{space of features} \rightarrow \text{classes} = (\text{discrete values or real values})$$

In later chapters, we'll see several examples of either classification or regression problems. One such problem we'll discuss is the classification of handwritten digits (the famous Modified National Institute of Standards and Technology, or `MNIST`, database). When given a set of images representing 0 to 9, the machine learning algorithm will try to classify each image in one of the 10 classes, wherein each class corresponds to one of the 10 digits. Each image is 28x28 (= 784) pixels in size. If we think of each pixel as one feature, then the algorithm will use a 784-dimensional feature space to classify the digits.

The following screenshot depicts the handwritten digits from the MNIST dataset:



Example of handwritten digits from the MNIST dataset

In the next sections, we'll talk about some of the most popular **classical** supervised algorithms. The following is by no means an exhaustive list or a thorough description of each machine learning method. We can refer to the book *Python Machine Learning* by *Sebastian Raschka* (`https://www.packtpub.com/big-data-and-business-intelligence/python-machine-learning`). It's a simple review meant to provide the reader with a flavor of the different techniques. Also, at the end of this chapter in the *Neural networks* section, we'll introduce neural networks and we'll talk about how deep learning differs from the classical machine learning techniques.

Linear and logistic regression

Regression algorithms are a type of supervised algorithm that uses features of the input data to predict a value, such as the cost of a house, given certain features, such as size, age, number of bathrooms, number of floors, and location. Regression analysis tries to find the value of the parameters for the function that best fits an input dataset.

In a linear-regression algorithm, the goal is to minimize a **cost function** by finding appropriate parameters for the function, over the input data that best approximates the target values. A cost function is a function of the error, that is, how far we are from getting a correct result. A popular cost function is the **mean square error** (**MSE**), where we take the square of the difference between the expected value and the predicted result. The sum over all the input examples gives us the error of the algorithm and represents the cost function.

Say we have a 100-square-meter house that was built 25 years ago with 3 bathrooms and 2 floors. Let's also assume that the city is divided into 10 different neighborhoods, which we'll denote with integers from 1 to 10, and say this house is located in the area denoted by 7. We can parameterize this house with a five-dimensional vector, `x = (100, 25, 3, 2, 7)`. Say that we also know that this house has an estimated value of €100,000. What we want is to create a function, `f`, such that `f(x) = 100000`.

In linear regression, this means finding a vector of weights, `w= (w1, w2, w3, w4, w5)`, such that the dot product of the vectors, `x • w = 10000`, would be `100*w1 + 25*w2 + 3*w3 + 2*w4 + 7*w5 = 100000` or $\sum_i w_i x_i$. If we had 1,000 houses, we could repeat the same process for every house, and ideally we would like to find a single vector, `w`, that can predict the correct value that is close enough for every house. The most common way to train a linear regression model can be seen in the following pseudocode block:

```
Initialize the vector w with some random values
repeat:
   E = 0 # initialize the cost function E with 0
    for every sample/target pair ($x_i$, $t_i$) of the training set:
        E += $(\vec{x}_i \cdot \vec{w} - t_i)^2$ # here ti is the real cost of the house
    MSE = E / total_number_of_samples # Mean Square Error
    use gradient descent to update the weights w based on MSE
 until MSE falls below threshold
```

First, we iterate over the training data to compute the cost function, MSE. Once we know the value of MSE, we'll use the gradient-descent algorithm to update *w*. To do this, we'll calculate the derivatives of the cost function with respect to each weight, w_i . In this way, we'll know how the cost function changes (increase or decrease) with respect to w_i . Then we'll update its value accordingly. In `Chapter 2`, *Neural Networks*, we will see that training neural networks and linear/logistic regressions have a lot in common.

We demonstrated how to solve a regression problem with linear regression. Let's take another task: trying to determine whether a house is overvalued or undervalued. In this case, the target data would be categorical `[1, 0]` - `1` for overvalued, `0` for undervalued, if the price of the house will be an input parameter instead of target value as before. To solve the task, we'll use **logistic regression**. This is similar to linear regression but with one difference: in linear regression, the output is $\vec{x} \cdot \vec{w}$. However, here the output will be a special logistic function (`https://en.wikipedia.org/wiki/Logistic_function`), $\sigma(\vec{x} \cdot \vec{w})$. This will **squash** the value of $\vec{x} \cdot \vec{w}$ in the `(0:1)` interval. You can think of the logistic function as a probability, and the closer the result is to 1, the more chance there is that the house is overvalued, and vice versa. Training is the same as with linear regression, but the output of the function is in the `(0:1)` interval and the labels is either 0 or 1.

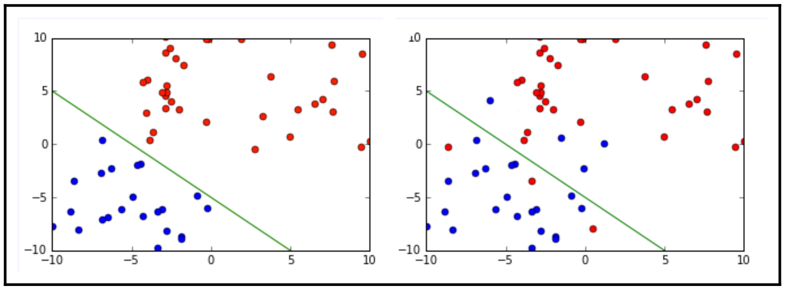
Logistic regression is not a classification algorithm, but we can turn it into one. We just have to introduce a rule that determines the class based on the logistic function output. For example, we can say that a house is overvalued if the value of $\sigma(\vec{x} \cdot \vec{w}) > 0.5$ and undervalued otherwise.

Support vector machines

A **support vector machine (SVM**) is a supervised machine learning algorithm that is mainly used for classification. It is the most popular member of the kernel method class of algorithms. An SVM tries to find a hyperplane, which separates the samples in the dataset.

A **hyperplane** is a plane in a high-dimensional space. For example, a hyperplane in a one-dimensional space is a point, and in a two-dimensional space, it would just be a line. We can think of classification as a process of trying to find a hyperplane that will separate different groups of data points. Once we have defined our features, every sample (in our case, an email) in the dataset can be thought of as a point in the multidimensional space of features. One dimension of that space represents all the possible values of one feature. The coordinates of a point (a sample) are the specific values of each feature for that sample. The ML algorithm task will be to draw a hyperplane to separate points with different classes. In our case, the hyperplane would separate spam from non-spam emails.

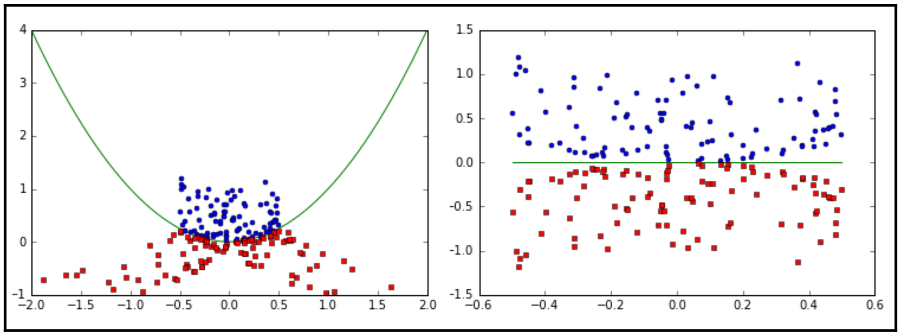
In the following diagram, on the top and bottom, you can see two classes of points (red and blue) that are in a two-dimensional feature space (the x and y axes). If both the x and y values of a point are below five, then the point is blue. In all other cases, the point is red. In this case, the classes are linearly-separable, meaning we can separate them with a hyperplane. Conversely, the classes in the image at the bottom are linearly-inseparable:



The SVM tries to find a hyperplane that maximizes the distance between itself and the points. In other words, from all possible hyperplanes that can separate the samples, the SVM finds the one that has the maximum distance from all points. In addition, SVMs can also deal with data that is not linearly-separable. There are two methods for this: introducing soft margins or using the **kernel trick**.

Soft margins work by allowing a few misclassified elements while retaining the most predictive ability of the algorithm. In practice, it's better not to overfit the machine learning model, and we could do so by relaxing some of the support-vector-machine hypotheses.

The kernel trick solves the same problem in a different way. Imagine that we have a two-dimensional feature space, but the classes are linearly-inseparable. The kernel trick uses a kernel function that transforms the data by adding more dimensions to it. In our case, after the transformation, the data will be three-dimensional. The linearly-inseparable classes in the two-dimensional space will become linearly-separable in the three dimensions and our problem is solved:



In the graph on the left image, we can see a non-linearly-separable set before the kernel was applied and on the bottom. On the right, we can see the same dataset after the kernel has been applied, and the data can be linearly separated

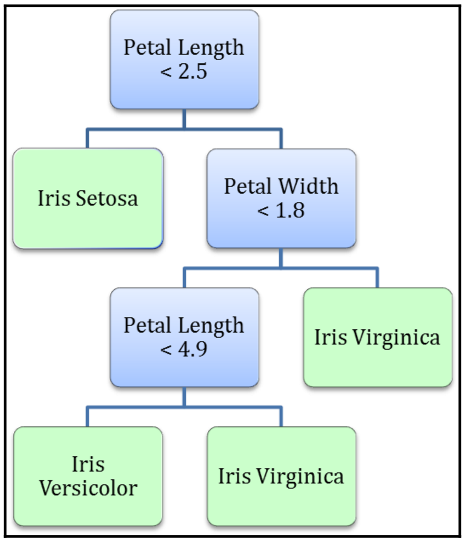
Decision Trees

Another popular supervised algorithm is the decision tree. A decision tree creates a classifier in the form of a tree. This is composed of decision nodes, where tests on specific attributes are performed; and leaf nodes, which indicate the value of the target attribute. To classify a new sample, we start at the root of the tree and navigate down the nodes until we reach a leaf.

A classic application of this algorithm is the Iris flower dataset (`http://archive.ics.uci.edu/ml/datasets/Iris`), which contains data from 50 samples of three types of Irises (Iris Setosa, Iris Virginica, and Iris Versicolor). Ronald Fisher, who created the dataset, measured four different features of these flowers:

- The length of their sepals
- The width of their sepals
- The length of their petals
- The width of their petals

Based on the different combinations of these features, it's possible to create a decision tree to decide which species each flower belongs to. In the following diagram, we have defined a decision tree that will correctly classify almost all the flowers using only two of these features, the petal length and width:



To classify a new sample, we start at the root note of the tree (petal length). If the sample satisfies the condition, we go left to the leaf, representing the Iris Setosa class. If not, we go right to a new node (petal width). This process continues until we reach a leaf. There are different ways to build decision trees, and we will discuss them later, in the chapter.

In recent years, decision trees have seen two major improvements. The first is Random Forests, which is an ensemble method that combines the predictions of multiple trees. The second is Gradient-Boosting Machines, which creates multiple sequential decision trees, where each tree tries to improve the errors made by the previous tree. Thanks to these improvements, decision trees have become very popular when working with certain types of data. For example, they are one of the most popular algorithms used in Kaggle competitions.

Naive Bayes

Naive Bayes is different from many other machine learning algorithms. Most machine learning techniques try to evaluate the probability of a certain event, Y , and given conditions, X, which we denote with $p(Y|X)$. For example, when we are given a picture that represents digits (that is, a picture with a certain distribution of pixels), what is the probability that the number is five? If the pixel's distribution is close to the pixel distribution of other examples that were labeled as five, the probability of that event will be high. If not, the probability will be low.

Sometimes we have the opposite information, given the fact that we know that we have an event, Y. We also know the probability, that our sample is X. The Bayes theorem states that $p(X|Y) = p(Y|X)p(X) / p(Y)$, where $p(X|Y)$ means the probability of event, X, given Y, which is also why naive Bayes is called a generative approach. For example, we may calculate the probability that a certain pixel configuration represents the number five, knowing what the probability is. Given that we have a five, that a random pixel configuration may match the given one.

This is best understood in the realm of medical testing. Let's say we conduct a test for a specific disease or cancer. Here, we want to know the probability of a patient having a particular disease, given that our test result was positive. Most tests have a reliability value, which is the percentage chance of the test being positive when administered on people with a particular disease. By reversing the $p(X|Y) = p(Y|X)p(X) / p(Y)$ expression, we get the following:

```
p(cancer | test=positive) = p(test=positive | cancer) * p(cancer) /
p(test=positive)
```

Let's assume that the test is 98% reliable. This means that if the test is positive, it will also be positive in 98% of cases. Conversely, if the person does not have cancer, the test result will be negative. Let's make some assumptions on this kind of cancer:

- This particular kind of cancer only affects older people
- Only 2% of people under 50 have this kind of cancer
- The test administered on people under 50 is positive only for 3.9% of the population (we could have derived this fact from the data, but we provide this information for the purpose of simplicity)

We can ask the following question: if a test is 98% accurate for cancer and if a 45-year-old person took the test, which turned out to be positive, what is the probability that they may have cancer? Using the preceding formula, we can calculate the following:

```
p(cancer | test=positive) = 0.98 * 0.02 / 0.039 = 0.50
```

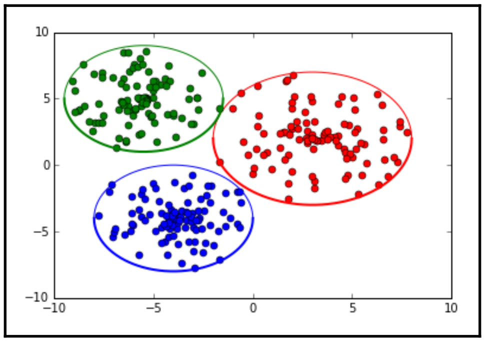
We call this classifier naive because it assumes the independence of different events to calculate their probability. For example, if the person had two tests instead of one, the classifier will assume that the outcome of test 2 did not know about the outcome of test 1, and the two tests were independent from one another. This means that taking test 1 could not change the outcome of test 2, and therefore its result was not biased by the first test.

Unsupervised learning

The second class of machine learning algorithms is unsupervised learning. Here, we don't label the data beforehand, but instead we let the algorithm come to its conclusion. One of the most common, and perhaps simplest, examples of unsupervised learning is clustering. This is a technique that attempts to separate the data into subsets.

To illustrate this, let's view the spam-or-not-spam email classification as an unsupervised learning problem. In the supervised case, for each email, we had a set of features and a label (spam or not spam). Here, we'll use the same set of features, but the emails will not be labeled. Instead, we'll ask the algorithm, when given the set of features, to put each sample in one of two separate groups (or clusters*)*. Then the algorithm will try to combine the samples in such a way that the intraclass similarity (which is the similarity between samples in the same cluster) is high and the similarity between different clusters is low. Different clustering algorithms use different metrics to measure similarity. For some more advanced algorithms, you don't have to specify the number of clusters.

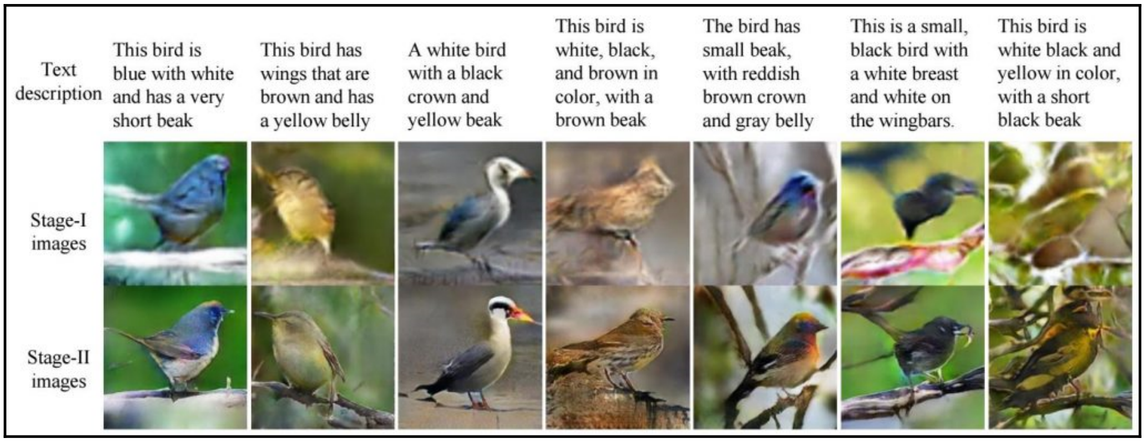
In the following graph, we show how a set of points can be classified to form three subsets:



Deep learning also uses unsupervised techniques, albeit different than clustering. In **natural language processing** (**NLP**), we use unsupervised (or semi-supervised, depending on who you ask) algorithms for vector representations of words. The most popular way to do this is called **word2vec**. For each word, we use its surrounding words (or its context) in the text and feed them to a simple neural network. The network produces a numerical vector, which contains a lot of information for the word (derived by the context). We then use these vectors instead of the words for various NLP tasks, such as sentiment analysis or machine translation. We'll describe word2vec in `Chapter 7`, *Recurrent Neural Networks and Language Models*.

Another interesting application of unsupervised learning is in generative models, as opposed to discriminative models. We will train a generative model with a large amount of data of a certain domain, such as images or text, and the model will try to generate new data similar to the one we used for training. For example, a generative model can colorize black and white images, change facial expressions in images, or even synthesize images based on a text description. In `Chapter 6`, *Generating Images with GANs and Variational Autoencoders,* we'll look at two of the most popular generative techniques, Variational Autoencoders and **Generative Adversarial Networks** (**GANs**).

The following depicts the techniques:


Text description
This bird is blue with white and has a very short beak
This bird has wings that are brown and has a yellow belly
A white bird with a black crown and yellow beak
This bird is white, black, and brown in color, with a brown beak
The bird has small beak, with reddish brown crown and gray belly
This is a small, black bird with a white breast and white on the wingbars.
This bird is white black and yellow in color, with a short black beak
Stage-I images
Stage-II images

In the preceding image, you can see how the authors of `StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks`, Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris Metaxas, use a combination of supervised learning and unsupervised GANs to produce photo-realistic images based on text descriptions.

K-means

K-means is a clustering algorithm that groups the elements of a dataset into k distinct clusters (hence the *k* in the name). Here is how it works:

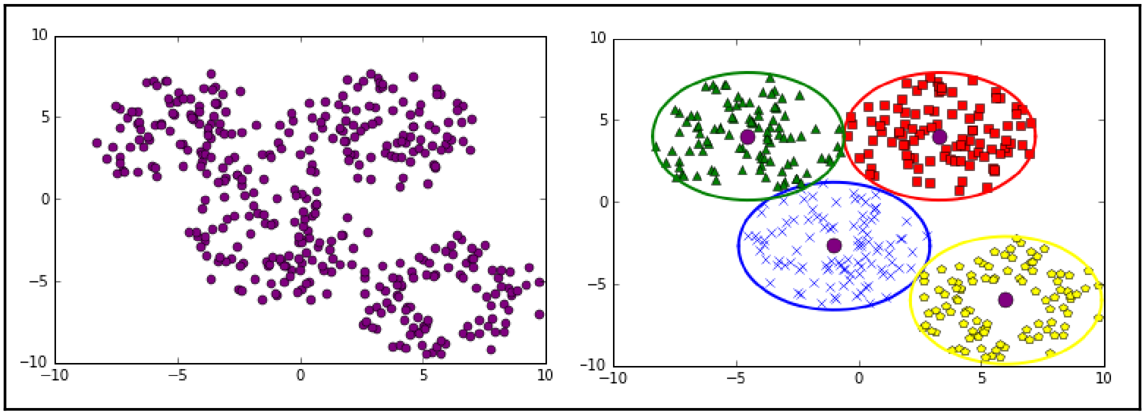
1. Choose k random points, called **centroids**, from the feature space, which will represent the center of each of the k clusters.
2. Assign each sample of the dataset (that is, each point in the feature space) to the cluster with the closest centroid.
3. For each cluster, we recomputed new centroids by taking the mean values of all the points in the cluster.
4. With the new centroids, we repeat steps 2 and 3 until the stopping criteria is met.

The preceding method is sensitive to the initial choice of random centroids and it may be a good idea to repeat it with different initial choices. It's also possible for some centroids to not be close to any of the points in the dataset, reducing the number of clusters down from k. Finally, it's worth mentioning that if we used k-means with `k=3` on the Iris dataset, we may get different distributions of the samples compared to the distribution of the decision tree that we'd introduced. Once more, this highlights how important it is to carefully choose and use the correct machine learning method for each problem.

Now let's discuss a practical example that uses k-means clustering. Let's say a pizza-delivery place wants to open four new franchises in a city, and they need to choose the locations for the sites. We can solve this problem with k-means:

1. Find the locations where pizza is ordered from most often and these will be our data points.
2. Choose four random points where the site locations will be located.

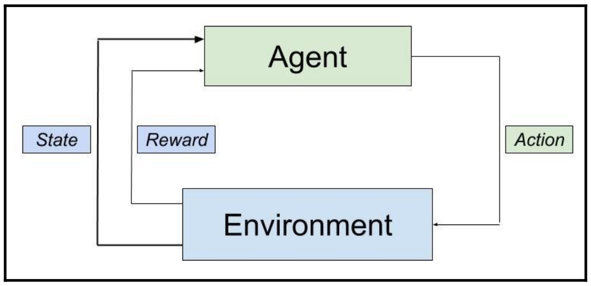
3. By using k-means clustering, we can identify the four best locations that minimize the distance to each delivery place:



In the left image, we can see the distribution of points where pizza is delivered most often. The round pints in the right image indicate where the new franchises should be located and their corresponding delivery areas

Reinforcement learning

The third class of machine learning techniques is called **reinforcement learning** (**RL**). We will illustrate this with one of the most popular applications of reinforcement learning: teaching machines how to play games. The machine (or agent) interacts with the game (or environment). The goal of the agent is to win the game. To do this, the agent takes actions that can change the environment's state. The environment provides the agent with reward signals that help the agent to decide its next action. Winning the game would provide the biggest reward. In formal terms, the goal of the agent is to maximize the total rewards it receives throughout the game:



The interaction of different elements of a reinforcement learning system

In reinforcement learning, the agent takes an action, which changes the state of the environment. The agent uses the new state and the reward to determine its next action.

Let's imagine a game of chess as an RL problem. The environment here would include the chess board along with the locations of the pieces. The goal of our agent is to beat the opponent. The agent will then receive a reward when they capture the opponent's piece, and they will win the biggest reward if they checkmate the opponent. Conversely, if the opponent captures a piece or checkmates the agent, the reward will be negative. However, as part of their larger strategies, the players will have to make moves that neither capture a piece, nor checkmate the other's king. The agent won't receive any reward then. If this was a supervised learning problem, we would have to provide a label or a reward for each move. This is not the case with reinforcement learning. In this book, we'll demonstrate how to use RL to allow the agent to use its previous experience in order to take new actions and learn from them in situations such as this.

Let's take another example, in which sometimes we have to sacrifice a pawn to achieve a more important goal (such as a better position on the chessboard). In such situations, our humble agent has to be smart enough to take a short-term loss as a long-term gain. In an even more extreme case, imagine we had the bad luck of playing against Magnus Carlsen, the current world chess champion. Surely, the agent will lose in this case. However, how would we know which moves were wrong and led to the agent's loss? Chess belongs to a class of problems where the game should be considered in its entirety in order to reach a successful solution, rather than just looking at the immediate consequences of each action. Reinforcement learning will give us the framework that will help the agent to navigate and learn in this complex environment.

An interesting problem arises from this newfound freedom to take actions. Imagine that the agent has learned one successful chess-playing strategy (or policy, in RL terms). After some games, the opponent might guess what that policy is and manage to beat us. The agent will now face a dilemma with the following decisions: either to follow the current policy and risk becoming predictable, or to experiment with new moves that will surprise the opponent, but also carry the risk of turning out even worse. In general terms, the agent uses a policy that gives them a certain reward, but their ultimate goal is to maximize the total reward. A modified policy might be more rewarding and the agent will be ineffective if they don't try to find such a policy. One of the challenges of reinforcement learning is the tradeoff between exploitation (following the current policy) and exploration (trying new moves). In this book, we'll learn the strategies to find the right balance between the two. We'll also learn how to combine deep neural networks with reinforcement learning, which made the field so popular in recent years.

So far, we've used only games as examples; however, many problems can fall into the RL domain. For example, you can think of an autonomous vehicle driving as an RL problem. The vehicle can get positive rewards if it stays within its lane and observes the traffic rules. It will gain negative rewards if it crashes. Another interesting recent application of RL is in managing stock portfolios. The goal of the agent would be to maximize the portfolio value. The reward is directly derived from the value of the stocks in the portfolio.

Q-learning

Q-learning is an off-policy temporal-difference reinforcement learning algorithm. What a mouthful! But fear not, let's not worry about what all this means, and instead just see how the algorithm works. To do this, we'll use the game of chess we introduced in the previous section. As a reminder, the board configuration (the locations of the pieces) is the current state of the environment. Here, the agents can take actions, *a*, by moving pieces, thus changing the state into a new one. We'll represent a game of chess as a graph where the different board configurations are the graph's vertices, and the possible moves from each configuration are the edges. To make a move, the agent follows the edge from the current state, *s*, to a new state, *s'*. The basic Q-learning algorithm uses Q-table to help the agent decide which moves to make. The Q-table contains one row for each board configuration, while the columns of the table are all possible actions that the agent can take (the moves). A table cell, *q(s, a)*, contains the cumulative expected reward, called **Q-value**. This is the potential total reward that the agent will receive for the remainder of the game if they take an action, *a*, from their current state, *s*. At the beginning, the Q-table is initialized with an arbitrary value. With that knowledge, let's see how Q-learning works:

```
Initialize the Q table with some arbitrary value
for each episode:
    Observe the initial state s
    for each step of the episode:
        Select new action a using a policy based on the Q-table
        Observe reward r and go to the new state s'
        Update q(s, a) in the Q table using the Bellman Equation
    until we reach a terminal state for the episode
```

An episode starts with a random initial state and finishes when we reach the terminal state. In our case, one episode would be one full game of chess.

The question that arises is this: how does the agent's policy determine what will be the next action? To do so, the policy has to take into account the Q-values of all the possible actions from the current state. The higher the Q-value, the more attractive the action is. However, the policy will sometimes ignore the Q-table (exploitation of the existing knowledge) and choose another random action to find higher potential rewards (exploration). In the beginning, the agent will take random actions because the Q-table doesn't contain much information. As time progresses and the Q-table is gradually filled, the agent will become more informed in interacting with the environment.

We update *q(s, a)* after each new action, by using **Bellman equation**. The Bellman equation is beyond the scope of this introduction, but we'll discuss it in detail in the later chapters. For now, it's enough to know that the updated value, *q(s, a)*, is based on the newly-received reward, *r* , as well as the maximum possible Q-value, $q_*(s', a')$, of the new state, *s'*.

This example was intended to help you understand the basic workings of Q-learning, but you might have noticed an issue with this. We store the combination of all possible board configurations and moves in the Q-table. This would make the table huge and impossible to fit in today's computer memory. Fortunately, there is a solution for this: we can replace the Q-table with a neural network, which will tell the agent what the optimal action is in each state. In recent years, this development has allowed reinforcement learning algorithms to achieve superhuman performance on tasks such as the game of Go, Dota 2, and Doom. In this book, we'll discuss how to apply Q-learning and other RL algorithms to some of these tasks.

Components of an ML solution

So far, we've discussed three major classes of machine learning algorithms. However, to solve an ML problem, we'll need a system in which the ML algorithm is only part of it. The most important aspects of such a system are as follows:

- **Learner**: This is algorithm is used with its learning philosophy. The choice of this algorithm is determined by the problem we're trying to solve, since different problems can be better suited for certain machine learning algorithms.
- **Training data**: This is the raw dataset that we are interested in. This can be labeled or unlabeled. It's important to have enough sample data for the learner to understand the structure of the problem.

- **Representation**: This is how we express the data in terms of the chosen features, so that we can feed it to the learner. For example, to classify handwritten images of digits, we'll represent the image as an array of values, where each cell will contain the color value of one pixel. A good choice of representation of the data is important for achieving better results.
- **Goal**: This represents the reason to learn from the data for the problem at hand. This is strictly related to the target, and helps define how and what the learner should use and what representation to use. For example, the goal may be to clean our mailbox from unwanted emails, and this goal defines what the target of our learner is. In this case, it is the detection of spam emails.
- **Target**: This represents what is being learned as well as the final output. The target can be a classification of unlabeled data, a representation of input data according to hidden patterns or characteristics, a simulator for future predictions, or a response to an outside stimulus or strategy (in the case of reinforcement learning).

It can never be emphasized enough: any machine learning algorithm can only achieve an approximation of the target and not a perfect numerical description. Machine learning algorithms are not exact mathematical solutions to problems, they are just approximations. In the previous paragraph, we defined learning as a function from the space of features (the input) into a range of classes. We'll later see how certain machine learning algorithms, such as neural networks, can approximate any function to any degree, in theory. This theorem is called the `Universal Approximation Theorem`, but it does not imply that we can get a precise solution to our problem. In addition, solutions to the problem can be better achieved by better understanding the training data.

Typically, a problem that is solvable with classic machine learning techniques may require a thorough understanding and processing of the training data before deployment. The steps to solve an ML problem are as follows:

- **Data collection**: This implies the gathering of as much data as possible. In the case of supervised learning, this also includes correct labeling.
- **Data processing**: This implies cleaning the data, such as removing redundant or highly correlated features, or even filling missing data, and understanding the features that define the training data.

- **Creation of the test case**: Usually, the data can be divided into three sets:
    - **Training set**: We use this set to train the ML algorithm.
    - **Validation set**: We use this set to evaluate the accuracy of the algorithm with unknown data during training. We'll train the algorithm for some time on the training set and then we'll use the validation set to check its performance. If we are not satisfied with the result, we can tune the hyperparameters of the algorithm and repeat the process again. The validation set can also help us to determine when to stop the training. We'll learn more about this later in this section.
    - **Test set**: When we finish tuning the algorithm with the training or validation cycle, we'll use the test set only *once* for a final evaluation. The test set is similar to the validation set in the sense that the algorithm hasn't used it during training. However, when we strive to improve the algorithm on the validation data, we may inadvertently introduce bias, which can skew the results in favor of the validation set and not reflect the actual performance. Because we use the test only once, this will provide a more objective measurement of the algorithm.

One of the reasons for the success of deep learning algorithms is that they usually require less data processing than classic methods. For a classic algorithm, you would have to apply different data processing and extract different features for each problem. With **DL**, you can apply the same data processing pipeline for most tasks. With DL, you can be more productive and you don't need as much domain knowledge for the task at hand compared to the classic ML algorithms.

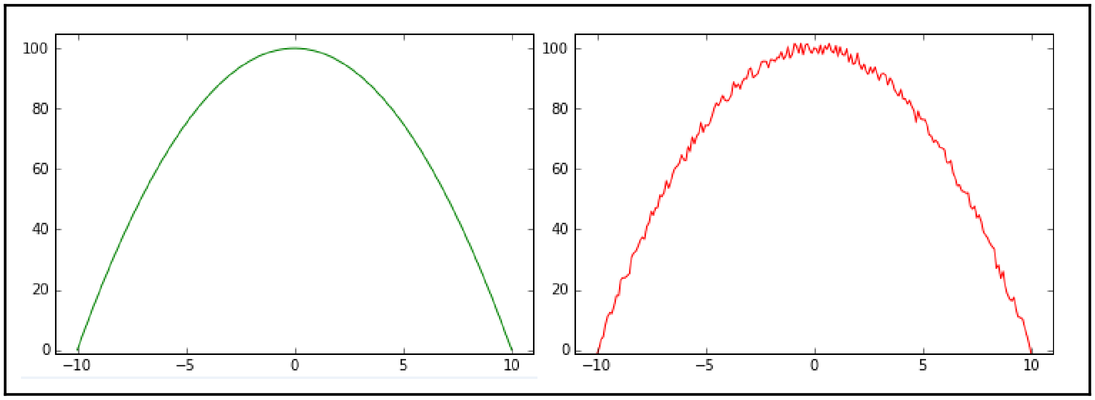
There are many valid reasons to create testing and validation datasets. As mentioned, machine learning techniques can only produce an approximation of the desired result. Often, we can only include a finite and limited number of variables, and there may be many variables that are outside of our control. If we only used a single dataset, our model may end up memorizing the data, and producing an extremely high accuracy value on the data it has memorized. However, this result may not be reproducible on other similar but unknown datasets. One of the key goals of machine learning algorithms is their ability to generalize. This is why we create both, a validation set used for tuning our model selection during training, and a final test set only used at the end of the process to confirm the validity of the selected algorithm.

To understand the importance of selecting valid features and to avoid memorizing the data, which is also referred to as overfitting in the literature-and we'll use that term from now on-let's use a joke taken from an xkcd comic as an example (`http://xkcd.com/1122`):

> *"Up until 1996, no democratic US presidential candidate who was an incumbent and with no combat experience had ever beaten anyone whose first name was worth more in Scrabble."*

It's apparent that such a rule is meaningless, but it underscores the importance of selecting valid features and the question, "how much is a name worth in Scrabble," can bear any relevance while selecting a US president? Also, this example doesn't have any predictive power over unknown data. We'll call this overfitting, which refers to making predictions that fit the data at hand perfectly, but don't generalize to larger datasets. Overfitting is the process of trying to make sense of what we'll call noise (information that does not have any real meaning) and trying to fit the model to small perturbations.

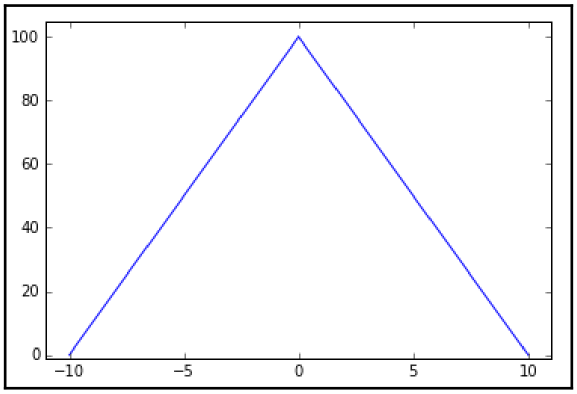
To further explain this, let's try to use machine learning to predict the trajectory of a ball thrown from the ground up into the air (not perpendicularly) until it reaches the ground again. Physics teaches us that the trajectory is shaped as a parabola. We also expect that a good machine learning algorithm observing thousands of such throws would come up with a parabola as a solution. However, if we were to zoom into the ball and observe the smallest fluctuations in the air due to turbulence, we might notice that the ball does not hold a steady trajectory but may be subject to small perturbations, which in this case is the noise. A machine learning algorithm that tries to model these small perturbations would fail to see the big picture and produce a result that is not satisfactory. In other words, overfitting is the process that makes the machine learning algorithm see the trees, but forgets about the forest:



A good prediction model versus a bad (overfitted) prediction model, with the trajectory of a ball thrown from the ground

This is why we separate the training data from the validation and test data; if the accuracy on the test data was not similar to the training data accuracy, that would be a good indication that the model overfits. We need to make sure that we don't make the opposite error either, that is, underfitting the model. In practice though, if we aim to make our prediction model as accurate as possible on our training data, underfitting is much less of a risk, and care is taken to avoid overfitting.

The following image depicts underfitting:



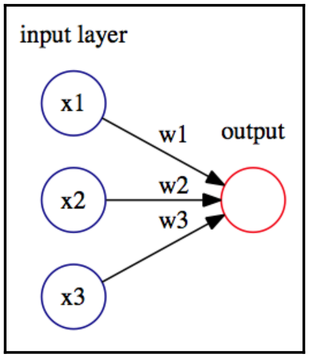
Underfitting can be a problem as well

Neural networks

In the previous sections, we introduced some of the popular classical machine learning algorithms. In this section, we'll talk about neural networks, which is the main focus of the book.

The first example of a neural network is called the **perceptron**, and this was invented by Frank Rosenblatt in 1957. The perceptron is a classification algorithm that is very similar to logistic regression. Such as logistic regression, it has weights, w, and its output is a function, $f(\vec{x} \cdot \vec{w})$, of the dot product, $\vec{x} \cdot \vec{w}$ (or $f(\sum_i w_i x_i)$ of the weights and input.

The only difference is that *f* is a simple step function, that is, if $f(\vec{x} \cdot \vec{w}) > 0$, then $f(\vec{x} \cdot \vec{w}) = 1$, or else $f(\vec{x} \cdot \vec{w}) = 0$, wherein we apply a similar logistic regression rule over the output of the logistic function. The perceptron is an example of a simple one-layer neural feedforward network:

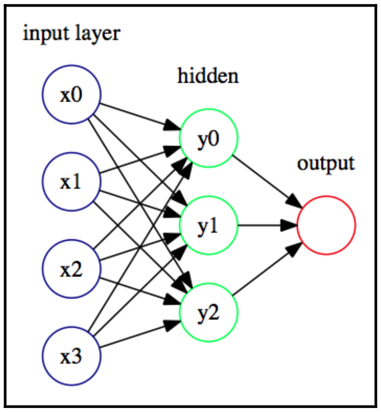


A simple perceptron with three input units (neurons) and one output unit (neuron)

The perceptron was very promising, but it was soon discovered that is has serious limitations as it only works for linearly-separable classes. In 1969, Marvin Minsky and Seymour Papert demonstrated that it could not learn even a simple logical function such as XOR. This led to a significant decline in the interest in perceptron's.

However, other neural networks can solve this problem. A classic multilayer perceptron has multiple interconnected perceptron's, such as units that are organized in different sequential layers (input layer, one or more hidden layers, and an output layer). Each unit of a layer is connected to all units of the next layer. First, the information is presented to the input layer, then we use it to compute the output (or activation), y_i, for each unit of the first hidden layer. We propagate forward, with the output as input for the next layers in the network (hence feedforward), and so on until we reach the output. The most common way to train neural networks is with a gradient descent in combination with backpropagation. We'll discuss this in detail in `chapter 2`, *Neural Networks*.

The following diagram depicts the neural network with one hidden layer:



Neural network with one hidden layer

Think of the hidden layers as an abstract representation of the input data. This is the way the neural network understands the features of the data with its own internal logic. However, neural networks are non-interpretable models. This means that if we observed the y_i activations of the hidden layer, we wouldn't be able to understand them. For us, they are just a vector of numerical values. To bridge the gap between the network's representation and the actual data we're interested in, we need the output layer. You can think of this as a translator; we use it to understand the network's logic, and at the same time, we can convert it to the actual target values that we are interested in.

The `Universal approximation theorem` tells us that a feedforward network with one hidden layer can represent any function. It's good to know that there are no theoretical limits on networks with one hidden layer, but in practice we can achieve limited success with such architectures. In `Chapter 3`, *Deep Learning Fundamentals*, we'll discuss how to achieve better performance with deep neural networks, and their advantages over the **shallow** ones. For now, let's apply our knowledge by solving a simple classification task with a neural network.

Introduction to PyTorch

In this section, we'll introduce `PyTorch`, version 1.0. PyTorch is an open source python deep learning framework, developed primarily by Facebook that has been gaining momentum recently. It provides the **Graphics Processing Unit** (**GPU**), an accelerated multidimensional array (or **tensor**) operation, and computational graphs, which we can be used to build neural networks. Throughout this book, we'll use PyTorch, TensorFlow, and Keras, and we'll talk in detail about these libraries and compare them in `Chapter 3`, *Deep Learning Fundamentals*.

The steps are as follows:

1. Let's create a simple neural network that will classify the Iris flower dataset. The following is the code block for creating a simple neural network:

```
import pandas as pd

dataset =
pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-da
tabases/iris/iris.data',
                      names=['sepal_length', 'sepal_width',
'petal_length', 'petal_width', 'species'])

dataset['species'] = pd.Categorical(dataset['species']).codes

dataset = dataset.sample(frac=1, random_state=1234)

train_input = dataset.values[:120, :4]
train_target = dataset.values[:120, 4]

test_input = dataset.values[120:, :4]
test_target = dataset.values[120:, 4]
```

2. The preceding code is boilerplate code that downloads the Iris dataset CSV file and then loads it into the pandas DataFrame. We then shuffle the DataFrame rows and split the code into numpy arrays, `train_input/train_target` (flower properties/flower class), for the training data and `test_input/test_target` for the test data.

3. We'll use 120 samples for training and 30 for testing. If you are not familiar with pandas, think of this as an advanced version of NumPy. Let's define our first neural network:

```
import torch

torch.manual_seed(1234)

hidden_units = 5

net = torch.nn.Sequential(
 torch.nn.Linear(4, hidden_units),
 torch.nn.ReLU(),
 torch.nn.Linear(hidden_units, 3)
)
```

4. We'll use a feedforward network with one hidden layer with five units, a ReLU activation function (this is just another type of activation, defined simply as `f(x) = max(0, x)`), and an output layer with three units. The output layer has three units, whereas each unit corresponds to one of the three classes of Iris flower. We'll use one-hot encoding for the target data. This means that each class of the flower will be represented as an array (`Iris Setosa = [1, 0, 0]`, `Iris Versicolour = [0, 1, 0]`, and `Iris Virginica = [0, 0, 1]`), and one element of the array will be the target for one unit of the output layer. When the network classifies a new sample, we'll determine the class by taking the unit with the highest activation value.
5. `torch.manual_seed(1234)` enables us to use the same random data every time for the reproducibility of results.
6. Choose the optimizer and loss function:

```
# choose optimizer and loss function
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1,
momentum=0.9)
```

7. With the criterion variable, we define the loss function that we'll use, in this case, this is cross-entropy loss. The loss function will measure how different the output of the network is compared to the target data.

8. We then define the **stochastic gradient descent** (**SGD**) optimizer with a learning rate of 0.1 and a momentum of 0.9. The SGD is a variation of the gradient descent algorithm. We'll discuss loss functions and SGD in detail in `Chapter 2`, *Neural Networks*. Now, let's train the network:

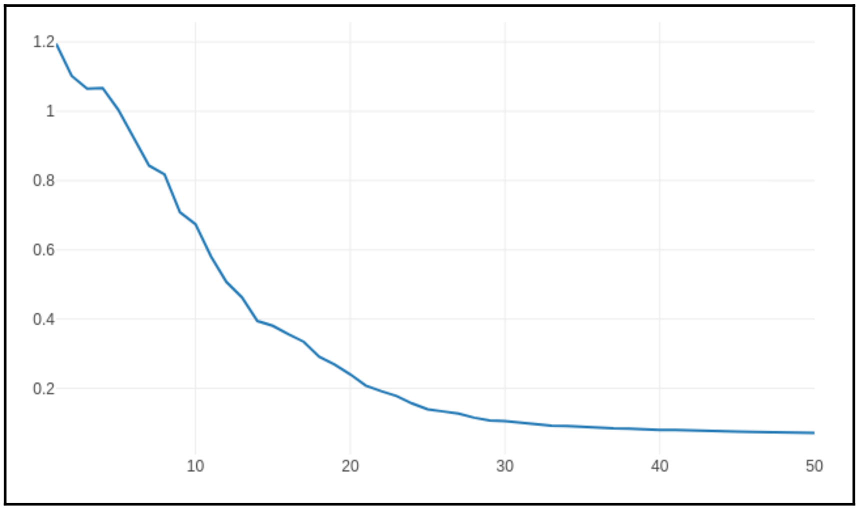
```
# train
epochs = 50

for epoch in range(epochs):
 inputs =
torch.autograd.Variable(torch.Tensor(train_input).float())
 targets =
torch.autograd.Variable(torch.Tensor(train_target).long())

 optimizer.zero_grad()
 out = net(inputs)
 loss = criterion(out, targets)
 loss.backward()
 optimizer.step()

 if epoch == 0 or (epoch + 1) % 10 == 0:
 print('Epoch %d Loss: %.4f' % (epoch + 1, loss.item()))
```

9. We'll run the training for 50 epochs, which means that we'll iterate 50 times over the training dataset:
    1. Create the torch variable that are **input** and **target** from the numpy array `train_input` and `train_target`.
    2. Zero the gradients of the optimizer to prevent accumulation from the previous iterations. We feed the training data to the neural network net (input) and we compute the loss function criterion (out, targets) between the network output and the target data.
    3. Propagate the loss value back through the network. We do this so that we can calculate how each network weight affects the loss function.
    4. The optimizer updates the weights of the network in a way that will reduce the future loss function values.

    When we run the training, the output is as follows:

```
Epoch 1 Loss: 1.2181
Epoch 10 Loss: 0.6745
Epoch 20 Loss: 0.2447
Epoch 30 Loss: 0.1397
Epoch 40 Loss: 0.1001
Epoch 50 Loss: 0.0855
```

In the following graph, you can see how the loss function decreases with each epoch. This shows how the network gradually learns the training data:



The loss function decreases with the number of epochs

10. Let's see what the final accuracy of our model is:

```
import numpy as np

inputs = torch.autograd.Variable(torch.Tensor(test_input).float())
targets = torch.autograd.Variable(torch.Tensor(test_target).long())

optimizer.zero_grad()
out = net(inputs)
_, predicted = torch.max(out.data, 1)

error_count = test_target.size - np.count_nonzero((targets ==
predicted).numpy())
print('Errors: %d; Accuracy: %d%%' % (error_count, 100 *
torch.sum(targets == predicted) / test_target.size))
```

We do this by feeding the test set to the network and computing the error manually. The output is as follows:

```
Errors: 0; Accuracy: 100%
```

We were able to classify all 30 test samples correctly.

We must also keep in mind trying different hyperparameters of the network and see how the accuracy and loss functions work. You could try changing the number of units in the hidden layer, the number of epochs we train in the network, as well as the learning rate.

Summary

In this chapter, we covered what machine learning is and why it's so important. We talked about the main classes of machine learning techniques and some of the most popular classic ML algorithms. We also introduced a particular type of machine learning algorithm, called neural networks, which is at the basis for deep learning. Then, we looked at a coding example where we used a popular machine learning library to solve a particular classification problem. In the next chapter, we'll cover neural networks in more detail and explore their theoretical justifications.

2
Neural Networks

In `Chapter 1`, *Machine Learning – an Introduction*, we introduced a number of basic **machine learning**(**ML**) concepts and techniques. We went through the main ML paradigms, as well as some popular **classic** ML algorithms, and we finished with neural networks. In this chapter, we will formally introduce what neural networks are, describe in detail how a neuron works, see how we can stack many layers to create a deep feedforward neural network, and then we'll learn how to train them.

In this chapter, we will cover the following topics:

- The need for neural networks
- An introduction to neural networks
- Training neural networks

TIP

Initially, neural networks were inspired by the biological brain (hence the name). Over time, however, we've stopped trying to emulate how the brain works and instead we focused on finding the correct configurations for specific tasks including computer vision, natural language processing, and speech recognition. You can think of it in this way: for a long time, we were inspired by the flight of birds, but, in the end, we created airplanes, which are quite different. We are still far from matching the potential of the brain. Perhaps the machine learning algorithms in the future will resemble the brain more, but that's not the case now. Hence, for the rest of this book, we won't try to create analogies between the brain and neural networks.

The need for neural networks

Neural networks have been around for many years, and they've gone through several periods during which they've fallen in and out of favor. But recently, they have steadily gained ground over many other competing machine learning algorithms. This resurgence is due to having computers that are fast, the use of **graphical processing units** (**GPUs**) versus the most traditional use of **computing processing units** (**CPUs**), better algorithms and neural net design, and increasingly larger datasets that we'll see in this book. To get an idea of their success, let's take the ImageNet Large-Scale Visual Recognition Challenge (`http://image-net.org/challenges/LSVRC/`, or just ImageNet). The participants train their algorithms using the `ImageNet` database. It contains more than one million high-resolution color images in over a thousand categories (one category may be images of cars, another of people, trees, and so on). One of the tasks in the challenge is to classify unknown images in these categories. In 2011, the winner achieved a top-five accuracy of 74.2%. In 2012, Alex Krizhevsky and his team entered the competition with a convolutional network (a special type of deep network). That year, they won with a top-five accuracy of 84.7%. Since then, the winners have always been convolutional networks and the current top-five accuracy is 97.7%. But deep learning algorithms have excelled in other areas; for example, both Google Now and Apple's Siri assistants rely on deep networks for speech recognition and Google's use of deep learning for their translation engines.

We'll talk about these exciting advances in the next chapters. But for now, we'll use simple networks with one or two layers. You can think of these as toy examples that are not deep networks, but understanding how they work is important. Here's why:

- **First:** knowing the theory of neural networks will help you understand the rest of the book, because a large majority of neural networks in use today share common principles. Understanding simple networks means that you'll understand deep networks too.
- **Second:** having some fundamental knowledge is always good. It will help you a lot when you face some new material (even material not included in this book).

I hope these arguments will convince you of the importance of this chapter. As a small consolation, we'll talk about deep learning in depth (pun intended) in `chapter 3`, *Deep Learning Fundamentals.*

An introduction to neural networks

We can describe a neural network as a mathematical model for information processing. As discussed in `Chapter 1`, *Machine Learning – an Introduction*, this is a good way to describe any ML algorithm, but, in this chapter, well give it a specific meaning in the context of neural networks. A neural net is not a fixed program, but rather a model, a system that processes information, or inputs. The characteristics of a neural network are as follows:

- Information processing occurs in its simplest form, over simple elements called **neurons.**
- Neurons are connected and they exchange signals between them through connection links.
- Connection links between neurons can be stronger or weaker, and this determines how information is processed.
- Each neuron has an internal state that is determined by all the incoming connections from other neurons.
- Each neuron has a different **activation function** that is calculated on its state, and determines its output signal.

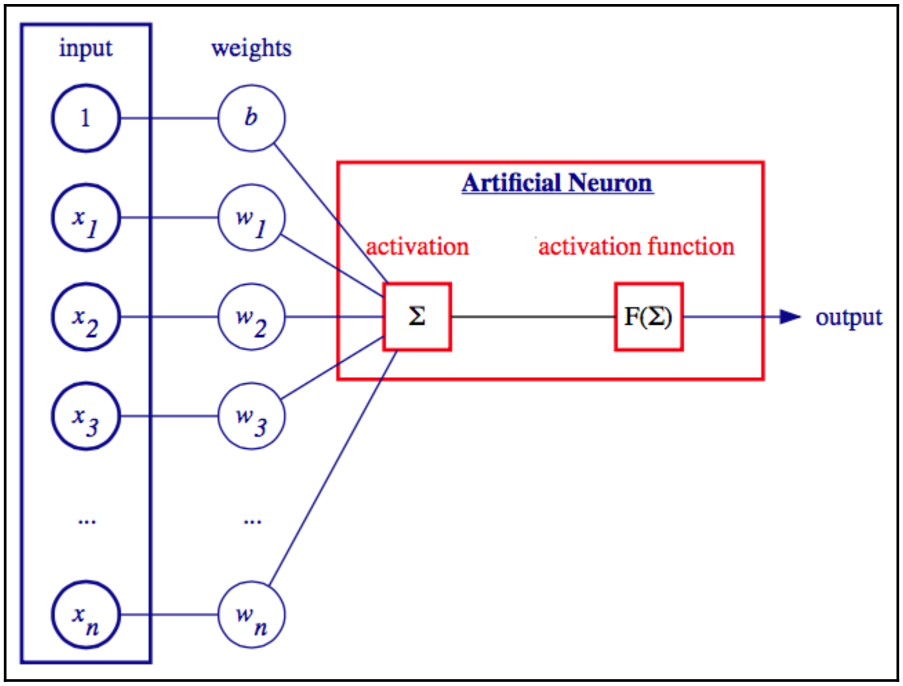
A more general description of a neural network would be as a computational graph of mathematical operations, but we will learn more about that later.

We can identify two main characteristics for a neural net:

- **The neural net architecture**: This describes the set of connections-namely, feedforward, recurrent, multi or single-layered, and so on-between the neurons, the number of layers, and the number of neurons in each layer.
- **The learning**: This describes what is commonly defined as the training. The most common but not exclusive way to train a neural network is with the gradient descent and backpropagation.

An introduction to neurons

A neuron is a mathematical function that takes one or more input values, and outputs a single numerical value:



In this diagram, we can see the different elements of the neuron

The neuron is defined as follows:

$$y = f(\sum_i x_i w_i + b)$$

1. First, we compute the weighted sum $\sum x_i w_i$ of the inputs x_i and the weights w_i (also known as an **activation value**). Here, x_i is either numerical values that represent the input data, or the outputs of other neurons (that is, if the neuron is part of a neural network):

- The weights w_i are numerical values that represent either the strength of the inputs or, alternatively, the strength of the connections between the neurons.
- The weight b is a special value called bias whose input is always 1.

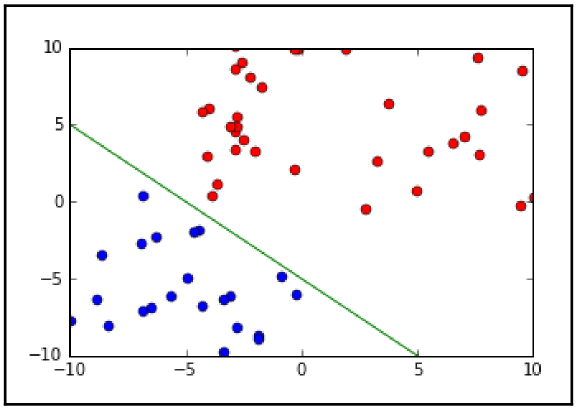
2. Then, we use the result of the weighted sum as an input to the **activation function** f, which is also known as **transfer function**. There are many types of activation functions, but they all have to satisfy the requirement to be **non-linear**, which we'll explain later in the chapter.

You might have noticed that the neuron is very similar to remove logistic regression and the perceptron, which we discussed in `Chapter 1`, *Machine Learning – an Introduction*. You can think of it as a generalized version of these two algorithms. If we use the logistic function or step function as activation functions, the neuron turns into logistic regression or perceptron respectively. Additionally, if we don't use any activation function, the neuron turns into linear regression. In this case, however, we are not limited to these cases and, as you'll see later, they are rarely used in practice.

As we mentioned in `Chapter 1`, *Machine Learning – an Introduction*, the activation value defined previously can be interpreted as the dot product between the vector w and the vector x: $y = f(\vec{x} \cdot \vec{w} + b)$. The vector x will be perpendicular to the weight vector w, if $\vec{x} \cdot \vec{w} = 0$. Therefore, all vectors x such that $\vec{x} \cdot \vec{w} = 0$ define a hyperplane in the feature space R^n , where n is the dimension of x.

That sounds complicated! To better understand it, let's consider a special case where the activation function is $f(x) = x$ and we only have a single input value, x. The output of the neuron then becomes $y = wx + b$, which is the linear equation. This shows that in one-dimensional input space, the neuron defines a line. If we visualize the same for two or more inputs, we'll see that the neuron defines a plane, or a hyperplane, for an arbitrary number of input dimensions.

In the following diagram, we can also see that the role of the bias, *b*, is to allow the hyperplane to shift away from the center of the coordinate system. If we don't use bias, the neuron will have limited representation power:



The preceding diagram displays the hyperplane

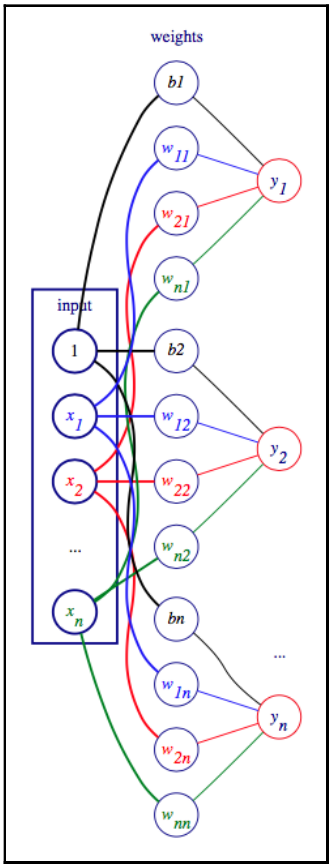
We already know from `Chapter 1`, *Machine Learning – an Introduction,* that the perceptron (hence the neuron) only works with linearly separable classes, and now we know that because it defines a hyperplane. To overcome this limitation, we'll need to organize the neurons in a neural network.

An introduction to layers

A neural network can have an indefinite number of neurons, which are organized in interconnected layers. The input layer represents the dataset and the initial conditions. For example, if the input is a grayscale image, the output of each neuron in the input layer is the intensity of one pixel of the image. For this very reason, we don't generally count the input layer as a part of the other layers. When we say 1-layer net, we actually mean that it is a simple network with just a single layer, the output, in addition to the input layer.

Unlike the examples we've seen so far, the output layer can have more than one neuron. This is especially useful in classification, where each output neuron represents one class. For example, in the case of the **Modified National Institute of Standards and Technology**(**MNIST)** dataset, we'll have 10 output neurons, where each neuron corresponds to a digit from 0-9. In this way, we can use the 1-layer net to classify the digit on each image. We'll determine the digit by taking the output neuron with the highest activation function value. If this is y_7, we'll know that the network thinks that the image shows the number 7.

In the following diagram, you can see the 1-layer feedforward network. In this case, we explicitly show the weights w for each connection between the neurons, but usually, the edges connecting neurons represent the weights implicitly. Weight w_{ij} connects the i-th input neuron with the j-th output neuron. The first input, 1, is the bias unit, and the weight, b_1, is the bias weight:



1-layer feedforward network

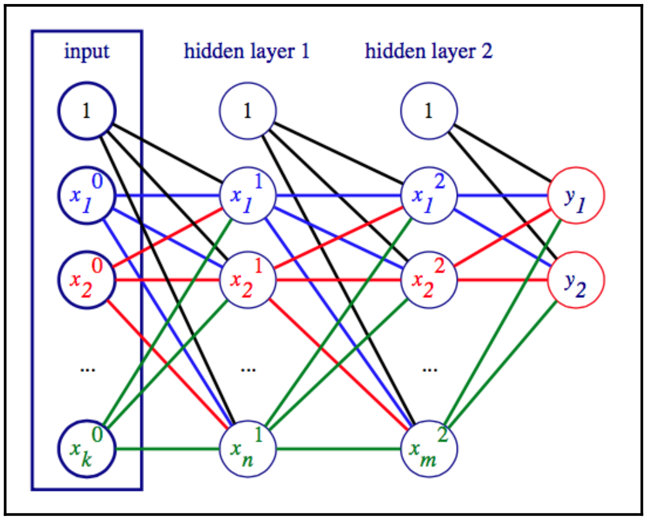
In the preceding diagram, we see the 1-layer neural network wherein the neurons on the left represent the input with bias b, the middle column represents the weights for each connection, and the neurons on the right represent the output given the weights w.

The neurons of one-layer can be connected to the neurons of other layers, but not to other neurons of the same layer. In this case, the input neurons are connected only to the output neurons.

But why do we need to organize the neurons in layers in the first place? One argument is that the neuron can convey limited information (just one value). But when we combine the neurons in layers, their outputs compose a **vector** and, instead of single activation, we can now consider the vector in its entirety. In this way, we can convey a lot more information, not only because the vector has multiple values, but also because the relative ratios between them carry additional information.

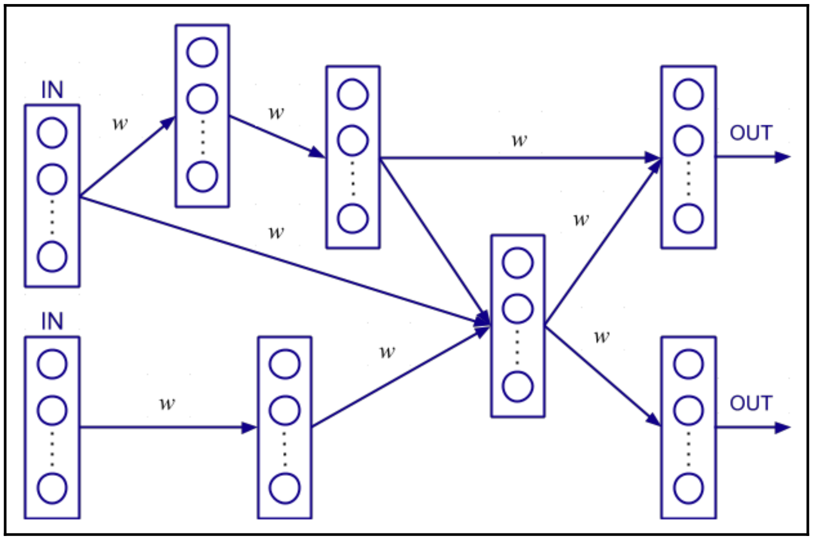
Multi-layer neural networks

As we have mentioned many times, 1-layer neural nets can only classify linearly separable classes. But there is nothing that prevents us from introducing more layers between the input and the output. These extra layers are called **hidden layers**. The following diagram demonstrates a 3-layer **fully connected** neural network with two hidden layers. The input layer has *k* input neurons, the first hidden layer has *n* hidden neurons, and the second hidden layer has *m* hidden neurons. The output, in this example, is the two classes y_1 and y_2. On top is the always-on bias neuron. A unit from one-layer is connected to all units from the previous and following layers (hence fully connected). Each connection has its own weight, *w*, that is not depicted for reasons of simplicity:



Multi-layer sequential network

But we are not limited to networks with sequential layers, as shown in the preceding diagram. The neurons and their connections form **directed cyclic graphs**. In such a graph, the information cannot pass twice from the same neuron (no loops) and it flows in only one direction, from the input to the output. We also chose to organize them in layers; therefore, the layers are also organized in the directed cyclic graph. The network in the preceding diagram is just a special case of a graph whose layers are connected sequentially. The following diagram also depicts a valid neural network with two input layers, two output layers, and randomly interconnected hidden layers. For the sake of simplicity, we've depicted the multiple weights, *w*, connecting the layers as a single line:



A neural network

There is a special class of neural networks called **recurrent networks**, which represent a directed **cyclic** graph (they can have loops). We'll discuss them in detail in `chapter 8`, *Reinforcement Learning Theory*.

In this section, we introduced the most basic type of neural network, that is, the neuron, and we gradually expanded it to a graph of neurons, organized in layers. But we can think of it in another way. Thus, we came to know that the neuron has a precise mathematical definition. Therefore, the neural network, as a composition of neurons, is also a mathematical function where the input data represents the function arguments and the network weights, *w*, are its parameters.

Different types of activation function

We now know that multi-layer networks can classify linearly inseparable classes. But to do this, they need to satisfy one more condition. If the neurons don't have activation functions, their output would be the weighted sum of the inputs, $\sum_i w_i x_i$, which is a **linear function**. Then the entire neural network, that is, a composition of neurons, becomes a composition of linear functions, which is also a linear function. This means that even if we add hidden layers, the network will still be equivalent to a simple linear regression model, with all its limitations. To turn the network into a **non-linear** function, we'll use non-linear activation functions for the neurons. Usually, all neurons in the same layer have the same activation function, but different layers may have different activation functions. The most common activation functions are as follows:

- $f(a) = a$: This function lets the activation value go through and is called the **identity function**.
- $f(a) = \begin{cases} 1 \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$: This function activates the neuron; if the activation is above a certain value, it's called the **threshold activity function**.
- $f(a) = \frac{1}{1+exp(-a)}$: This function is one of the most commonly used, as its output is bounded between 0 and 1, and it can be interpreted stochastically as the probability of the neuron activating. It's commonly called the **logistic function**, or the **logistic sigmoid**.
- $f(a) = \frac{2}{1+exp(-a)} - 1 = \frac{1-exp(-a)}{1+exp(-a)}$: This activation function is called **bipolar sigmoid**, and it's simply a logistic sigmoid rescaled and translated to have a range in (-1, 1).
- $f(a) = \frac{exp(a)-exp(-a)}{exp(a)+exp(-a)} = \frac{1-exp(-2a)}{1+exp(-2a)}$: This activation function is called the **hyperbolic tangent** (or **tanh**).
- $f(a) = \begin{cases} a \text{ if } a \geq 0 \\ 0 \text{ if } a < 0 \end{cases}$: This activation function is probably the closest to its biological counterpart. It's a mix of the identity and the threshold function, and it's called the **rectifier**, or **ReLU**, as in **Rectified Linear Unit**. There are variations on the ReLU, such as Noisy ReLU, Leaky ReLU, and ELU (Exponential Linear Unit).

The identity activation function, or the threshold function, was widely used at the inception of neural networks with implementations such as the perceptron or the Adaline (adaptive linear neuron), but subsequently lost traction in favor of the logistic sigmoid, the hyperbolic tangent, or the ReLU and its variations. The latter three activation functions differ in the following ways:

- Their range is different.
- Their derivatives behave differently during training.

The range for the logistic function is `(0,1)`, which is one reason why this is the preferred function for stochastic networks, in other words, networks with neurons that may activate based on a probability function. The hyperbolic function is very similar to the logistic function, but its range is `(-1, 1)`. In contrast, the ReLU has a range of `(0, ∞)`.

But let's look at the derivative (or the gradient) for each of the three functions, which is important for the training of the network. This is similar to how, in the linear regression example that we introduced in `Chapter 1`, *Machine Learning – an Introduction*, we were trying to minimize the function, following it along the direction opposite to its derivative.

For a logistic function *f*, the derivative is *f * (1-f)*, while if *f* is the hyperbolic tangent, its derivative is *(1+f) * (1-f)*.

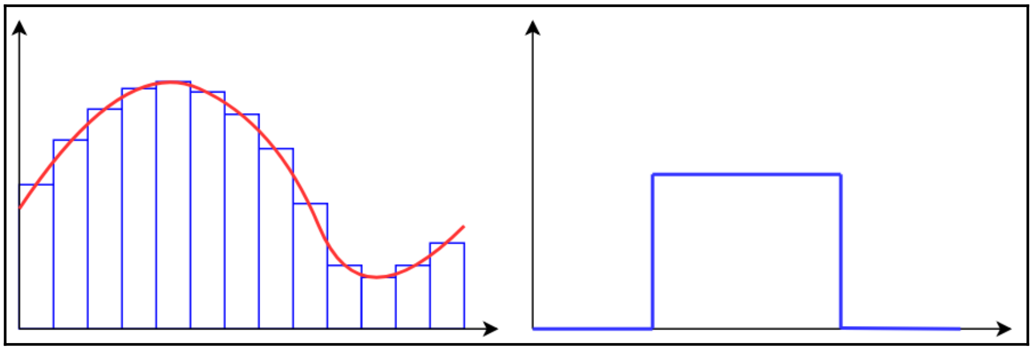
We can quickly calculate the derivative of the logistic sigmoid by simply noticing that the derivative with respect to activation *a* of the $\frac{1}{1+exp(-a)}$ function is given by the following:

$$\begin{aligned}f'(a) &= \frac{exp(-a)}{(1+exp(-a)) * (1+exp(-a))}\\ &= \frac{1}{1+exp(-a)} * \frac{(1+exp(-a))-1}{1+exp(-a)}\\ &= \frac{1}{1+exp(-a)} * (\frac{1+exp(-a)}{1+exp(-a)} - \frac{1}{1+exp(-a)})\\ &= f*(1-f)\end{aligned}$$

If *f* is the ReLU, the derivative is much simpler, that is, $f'(a) = \begin{cases} 1\; if\; a \geq 0 \\ 0\; if\; a < 0 \end{cases}$. Later in the book, we'll see the deep networks exhibit the **vanishing gradients** problem, and the advantage of the ReLU is that its derivative is constant and does not tend to zero as *a* becomes large.

Putting it all together with an example

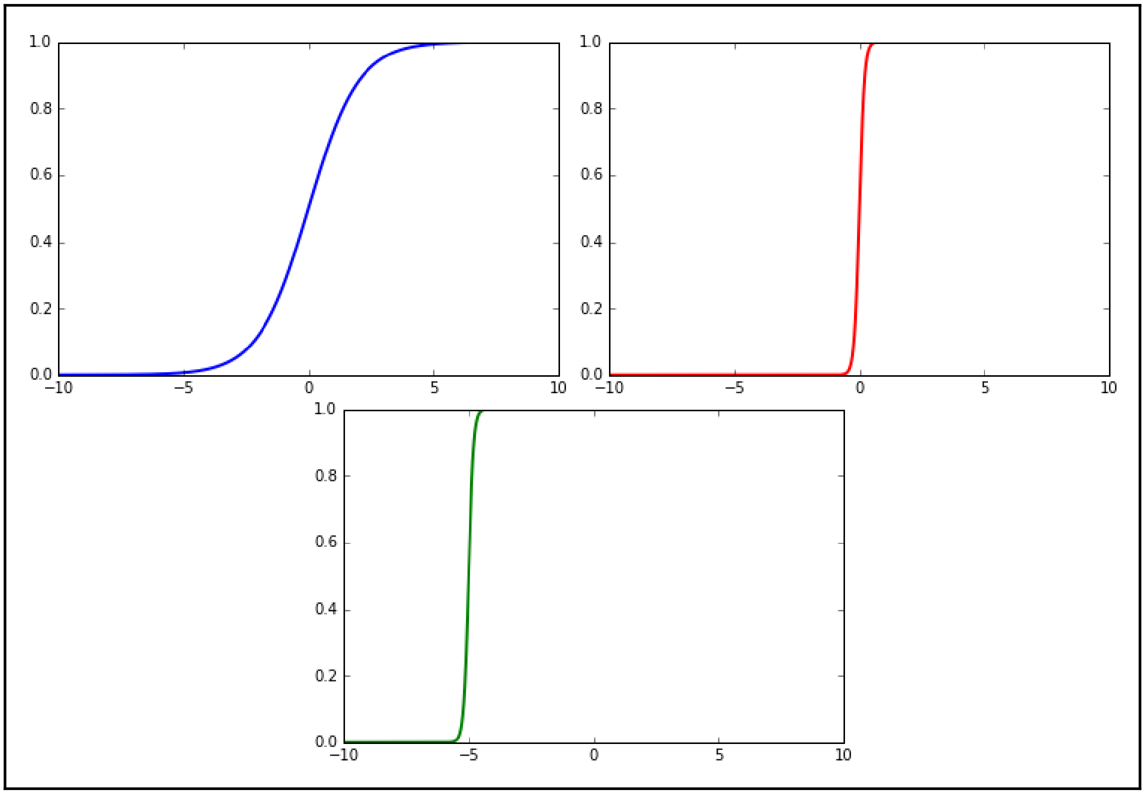
As we already mentioned, multi-layer neural networks can classify linearly separable classes. In fact, the Universal Approximation Theorem states that any continuous functions on compact subsets of R^n can be approximated by a neural network with at least one hidden layer. The formal proof of such a theorem is too complex to be explained here, but we'll attempt to give an intuitive explanation using some basic mathematics. We'll implement a neural network that approximates the boxcar function, in the following diagram on the right, which is a simple type of step function. Since a series of step functions can approximate any continuous function on a compact subset of *R*, this will give us an idea of why the Universal Approximation Theorem holds:



The diagram on the left depicts continuous function approximation with a series of step functions, while the diagram on the right illustrates a single boxcar step function

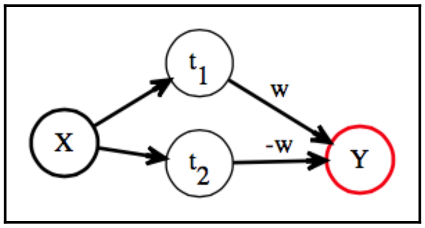
To do this, we'll use the logistic sigmoid activation function. As we know, the logistic sigmoid is defined as $1/(1+exp(-a))$, where $a(x) = \sum_i w_i x_i + b$:

1. Let's assume that we have only one input neuron, $x = x_1$
2. In the following diagrams, we can see that by making *w* very large, the sigmoid becomes close to a step function. On the other hand, *b* will simply translate the function along the *x* axis, and the translation *t* will be equal to *-b/w (t = -b/w)*:



On the left, we have a standard sigmoid with a weight of 1 and a bias of 0; in the middle, we have a sigmoid with a weight of 10; and on the right, we have a sigmoid with a weight of 10 and a bias of 50

With this in mind, let's define the architecture of our network. It will have a single input neuron, one hidden layer with two neurons, and a single output neuron:

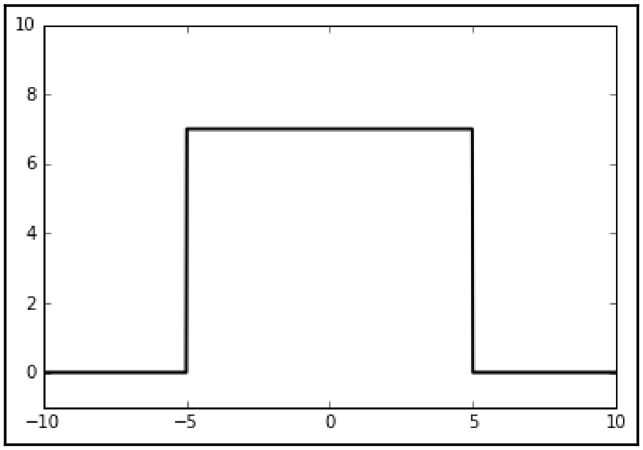


Both hidden neurons use the logistic sigmoid activation. The weights and biases of the network are organized in such a way as to take advantage of the sigmoid properties we described previously. The top neuron will initiate the first transition t_1 (0 to 1), and then, after some time has elapsed, the second neuron will initiate the opposite transition t_2. The following code implements this example:

```
# The user can modify the values of the weight w
# as well as bias_value_1 and bias_value_2 to observe
# how this plots to different step functions

import matplotlib.pyplot as plt
import numpy

weight_value = 1000

# modify to change where the step function starts
bias_value_1 = 5000

# modify to change where the step function ends
bias_value_2 = -5000

# plot the
plt.axis([-10, 10, -1, 10])

print("The step function starts at {0} and ends at {1}"
      .format(-bias_value_1 / weight_value,
              -bias_value_2 / weight_value))

inputs = numpy.arange(-10, 10, 0.01)
outputs = list()

# iterate over a range of inputs
for x in inputs:
    y1 = 1.0 / (1.0 + numpy.exp(-weight_value * x - bias_value_1))
    y2 = 1.0 / (1.0 + numpy.exp(-weight_value * x - bias_value_2))

    # modify to change the height of the step function
    w = 7

    # network output
    y = y1 * w - y2 * w

    outputs.append(y)

plt.plot(inputs, outputs, lw=2, color='black')
plt.show()
```

We set large values for `weight_value`, `bias_value_1`, and `bias_value_2`. In this way, the expressions `numpy.exp(-weight_value * x - bias_value_1)` and `numpy.exp(-weight_value * x - bias_value_2)` can switch between 0 and infinity in a very short interval of the input. In turn,`y1` and `y2` will switch between 1 and 0. This would make for a stepwise (as opposed to gradual) logistic sigmoid shape, as explained previously. Because the `numpy.exp` expressions get an infinity value, the code will produce `overflow encountered in exp warning`, but this is normal.

This code, when executed, produces the following result:



Training neural networks

We have seen how neural networks can map inputs onto determined outputs, depending on fixed weights. Once the **architecture** of the neural network has been defined and includes the feed forward network, the number of hidden layers, the number of neurons per layer, and the activation function, we'll need to set the weights, which, in turn, will define the internal states for each neuron in the network. First, we'll see how to do that for a 1-layer network using an optimization algorithm called **gradient descent**, and then we'll extend it to a deep feed forward network with the help of backpropagation.

The general concept we need to understand is the following:

Every neural network is an approximation of a function, so each neural network will not be equal to the desired function, but instead will differ by some value called **error**. During training, the aim is to minimize this error. Since the error is a function of the weights of the network, we want to minimize the error with respect to the weights. The error function is a function of many weights and, therefore, a function of many variables. Mathematically, the set of points where this function is zero represents a hypersurface, and to find a minimum on this surface, we want to pick a point and then follow a curve in the direction of the minimum.

TIP

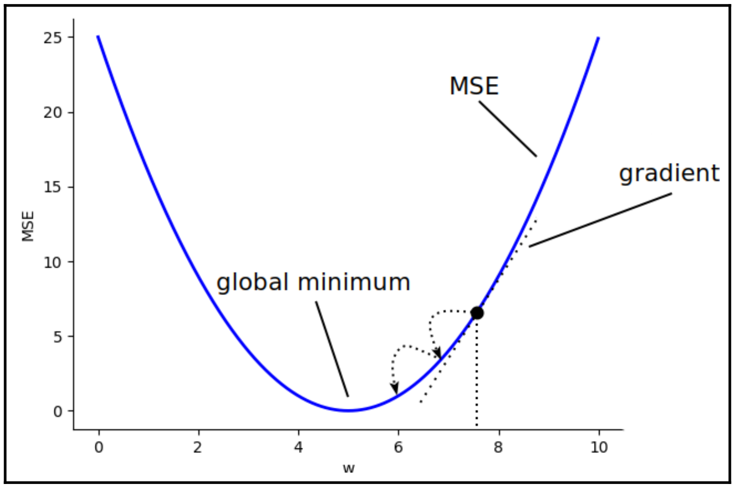
We should note that a neural network and its training are two separate things. This means we can adjust the weights of the network in some way other than gradient descent and backpropagation, but this is the most popular and efficient way to do so and is, ostensibly, the only way that is currently used in practice.

Linear regression

We have already introduced linear regression in `Chapter 1`, *Machine Learning – an Introduction*. To recap, regarding utilization of the vector notation, the output of a linear regression algorithm is a single value, y , and is equal to the dot product of the input values x and the weights w: $y = \vec{x} \cdot \vec{w}$. As we now know, linear regression is a special case of a neural network; that is, it's a single neuron with the **identity** activation function. In this section, we'll learn how to train linear regression with gradient descent and, in the following sections, we'll extend it to training more complex models. You can see how the gradient descent works in the following code block:

```
Initialize the weights w with some random values
repeat:
    # compute the mean squared error (MSE) loss function for all
samples of the training set
    # we'll denote MSE with J
    $J = MSE = \frac{1}{n}\sum_{i=0}^{n}(y^i - t^i)^2 = \frac{1}{n}\sum_{i=0}^{n}(x^i \bullet w - t^i)^2$
    # update the weights w based on the derivative of J with
respect to each weight
    $w \to w - \lambda\nabla(J(w))$
until MSE falls below threshold
```

At first, this might look scary, but fear not! Behind the scenes, it's very simple and straightforward mathematics (I know that sounds even scarier!). But let's not lose sight of our goal, which is to adjust the weights, w, in a way that will help the algorithm to predict the target values. To do this, first we need to know how the output y^i differs from the target value t^i for each sample of the training dataset (we use superscript notation to mark the i-th sample). We'll use the mean-squared error loss function (MSE), which is equal to the mean value of the squared differences y^i - t^i for all samples (the total number of samples in the training set is n). We'll denote MSE with J for ease of use and, to underscore that, we can use other loss functions. Each y^i is a function of w, and therefore, J is also a function of w. As we mentioned previously, the loss function J represents a hypersurface of dimension equal to the dimension of w (we are implicitly also considering the bias). To illustrate this, imagine that we have only one input value, x, and a single weight, w. We can see how the MSE changes with respect to w in the following diagram:



MSE diagram

Our goal is to minimize J, which means finding such w, where the value of J is at its **global minimum**. To do this, we need to know whether J increases or decreases when we modify w, or, in other words, the first **derivative** (or **gradient**) of J with respect to w:

1. In the general case, where we have multiple inputs and weights, we can calculate the partial derivative with respect to each weight w_j using the following formula:

$$\vec{d} = \frac{\partial J(w)}{\partial w_j} = \frac{\partial \frac{1}{n} \sum_i (y^i - t^i)^2}{\partial w_j}$$

2. And to move toward the minimum, we need to move in the opposite direction set by $\vec{d}$ for each w_j.
3. Let's calculate the derivative:

$$\frac{\partial J(w)}{\partial w_j} = \frac{\partial \frac{1}{n} \sum_i (y^i - t^i)^2}{\partial w_j} = \frac{1}{n} \sum_i \frac{\partial (y^i - t^i)^2}{\partial w_j} = \frac{2}{n} \sum_i \frac{\partial y^i}{\partial w_j} (y^i - t^i)$$

If $y^i = x^i \cdot w$, then $\frac{\partial y^i}{\partial w_j} = x_j^i$ and, therefore,

$$\frac{\partial J(w)}{\partial w_j} = \frac{2}{n} \sum_i x_j^i (y^i - t^i)$$

The notation can sometimes be confusing, especially the first time you encounter it. The input is given by the vectors x^i, where the superscript indicates the i-th example. Since x and w are vectors, the subscript indicates the j-th coordinate of the vector. y^i then represents the output of the neural network given the input x^i, while t^i represents the target, that is, the desired value corresponding to the input x^i.

4. Now, that we have calculated the partial derivatives, we'll update the weights with the following update rule:

$$w_j \to w_j - \eta \frac{\partial J(w)}{\partial w_j} = w_j - \eta \frac{2}{n} \sum_i x_j^i (y^i - t^i)$$

We can see that η is the **learning rate**. The learning rate determines the ratio by which the weight adjusts as new data arrives.

5. We can write the update rule in matrix form as follows:

$$w \to w - \eta \nabla (J(w)) = w - \eta \nabla (\frac{2}{n} \sum_i (y^i - t^i)^2)$$

Here, ∇, also called nabla, represents the vector of partial derivatives.

$\nabla = (\frac{\partial}{\partial w_1}, \ldots, \frac{\partial}{\partial w_n})$is a vector of partial derivatives. Instead of writing the update rule for *w* separately for each of its components, w_j, we can use the matrix form where, instead of writing the partial derivative, for each occurrence of *j*, we use ∇ to indicate each partial derivative for each *j*.

You may have noticed that in order to update the weights, we accumulate the error across all training samples. In reality, there are big datasets, and iterating over them for just one update would make training impractically slow. One solution to this problem is the **stochastic (or online) gradient descent** (**SGD**) algorithm, which works in the same way as regular gradient descent, but updates the weights after every training sample. However, SGD is prone to noise in the data. If a sample is an outlier, we risk increasing the error instead of decreasing it. A good compromise between the two is the **mini-batch gradient descent**, which accumulates the error for every *n* samples or **mini-batches** and performs one weight update. In practice, you'll almost always use mini-batch gradient descent.

Before we move to the next section, we should mention that besides the global minimum, the loss function might have multiple local minimums and minimizing its value is not as trivial, as in this example.

Logistic regression

Logistic regression uses logistic sigmoid activation, in contrast to linear regression, which uses the identity function. As we've seen before, the output of the logistic sigmoid is in the `(0,1)` range and can be interpreted as a probability function. We can use logistic regression for a 2-class (binary) classification problem, where our target, *t*, can have two values, usually 0 and 1 for the two corresponding classes. These discrete values shouldn't be confused with the values of the logistic sigmoid function, which is a continuous real-valued function between 0 and 1. The value of the sigmoid function represents the probability that the output is in class 0 or class 1:

1. Let's denote the logistic sigmoid function with *σ(a)*, where *a* is the neuron activation value *x•w*, as defined previously. For each sample *x*, the probability that the output is of class *y*, given the weights *w*, is as follows:

$$P(t|x,w) = \begin{cases} \sigma(a) & \text{if } t = 0 \\ 1 - \sigma(a) & \text{if } t = 0 \end{cases}$$

2. We can write that equation more succinctly as follows:

$$P(t|x,w) = \sigma(a)^t (1-\sigma(a))^{1-t}$$

3. And, since the probabilities $P(t^i|x^i, w)$ are independent for each sample x^i, the global probability is as follows:

$$P(t|x,w) = \prod_i P(t^i|x^i,w) = \prod_i \sigma(a^i)^{t^i}(1-\sigma(a^i))^{(1-t^i)}$$

4. If we take the natural log of the preceding equation (to turn products into sums), we get the following:

$$\begin{aligned}\log(P(t|x,w)) &= \log(\prod_i \sigma(a^i)^{t^i}(1-\sigma(a^i))^{(1-t^i)}) \\ &= \sum_i [t^i \log(\sigma(a^i)) + (1-t^i)\log(1-\sigma(a^i))]\end{aligned}$$

   Our objective now is to maximize this *log* to get the highest probability of predicting the correct results.

5. As before, we'll use gradient descent to minimize the cost function *J(w)*, defined by $J(w) = -log(P(y|x,w))$.

   As before, we calculate the derivative of the cost function with respect to the weights w_j to obtain the following:

$$\begin{aligned}\frac{\partial \log(P(t|x,w))}{\partial w_j} &= \frac{\sum_i [t^i \log(\sigma(a^i)) + (1-t^i)\log(1-\sigma(a^i))]}{\partial w_j} \\ &= \sum_i \frac{\partial[t^i \log(\sigma(a^i)) + (1-t^i)\log(1-\sigma(a^i))]}{\partial w_j} \\ &= \sum_i [t^i \frac{\partial \log(\sigma(a^i))}{\partial w_j} + (1-t^i)\frac{\partial \log(1-\sigma(a^i))}{\partial w_j}] \\ &= \sum_i [t^i(1-\sigma(a^i))x_j^i + (1-t^i)\sigma(a^i)x_j^i]\end{aligned}$$

To understand the last equation, let's recap the **chain rule** for derivatives, which states that if we have the function *F(x)=f(g(x))*, then the derivative of *F* with respect to *x* would be *F'(x)=f'(g(x))g'(x)*, or

$$\frac{\mathrm{d}F}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x}[f(g(x))] = \frac{\mathrm{d}}{\mathrm{d}g(x)}[f(g(x))]. \frac{\mathrm{d}}{\mathrm{d}x}[g(x)]$$

.

Now, back to our case:

$$\frac{\partial\sigma(a^i)}{\partial a^i} = \sigma(a^i)(1-\sigma(a^i))$$

$$\frac{\partial\sigma(a^i)}{\partial a_j} = 0$$

$$\frac{\partial a^i}{\partial w_j} = \frac{\partial \sum_k w_k x_k^i + b}{\partial w_j} = x_j^i$$

Therefore, according to the chain rule,the following is true:

$$\sum_i \frac{\partial \log(\sigma(a^i))}{\partial w_j} = \sum_i \frac{\partial \log(\sigma(a^i))}{\partial a^i}\frac{\partial a^i}{\partial w_j} = \frac{1}{\sigma(a^i)}\sigma(a^i)(1-\sigma(a^i))x_j^i$$
$$= (1-\sigma(a^i))x_j^i$$

Similarly,the following applies:

$$\sum_i \frac{\partial \log(1-\sigma(a^i))}{\partial w_j} = \sigma(a^i)x_j^i$$

This is similar to the update rule we've seen for linear regression.

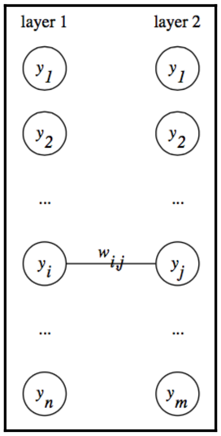
In this section, we saw a number of complicated equations, but you shouldn't feel bad if you don't fully understand them. We can recap by saying that we applied the same gradient descent algorithm to logistic regression as with linear regression. But this time, finding the partial derivatives of the error function with respect to the weights is slightly more complicated.

Backpropagation

So far, we have learned how to update the weights of 1-layer networks with gradient descent. We started by comparing the output of the network (that is, the output of the output layer) with the target value, and then we updated the weights accordingly. But, in a multi-layer network, we can only apply this technique for the weights that connect the final hidden layer to the output layer. That's because we don't have any target values for the outputs of the hidden layers. What we'll do instead is calculate the error in the final hidden layer and estimate what it would be in the previous layer. We'll propagate that error back from the last layer to the first layer; hence, we get the name backpropagation. Backpropagation is one of the most difficult algorithms to understand, but all you need is some knowledge of basic differential calculus and the chain rule.

Let's first introduce some notation:

1. We'll define w_{ij} as the weight between the i-th neuron of layer l, and the j-th neuron of layer $l+1$.
2. In other words, we use subscripts i and j, where the element with subscript i belongs to the layer preceding the layer containing the element with subscript j.
3. In a multi-layer network, l and $l+1$ can be any two consecutive layers, including input, hidden, and output layers.
4. Note that the letter y is used to denote both input and output values. y_i is the input to the next layer $l+1$, and it's also the output of the activation function of layer l:



In this example, layer 1 represents the input, layer 2 the output, and w_{ij} connects the y_i activation in layer 1 to the inputs of the j-th neuron of layer 2

5. We'll denote the cost function (error) with *J*, the activation value *x•w* with *a*, and the activation function (sigmoid, ReLU, and so on) output with *y*.
6. To recap the chain rule, for *F(x) = f(g(x))*, we have $\frac{\mathrm{d}F}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x}[f(g(x))] = \frac{\mathrm{d}}{\mathrm{d}g(x)}[f(g(x))]. \frac{\mathrm{d}}{\mathrm{d}x}[g(x)]$. In our case, a_j is a function of the weights, w_{*j}, y_j is a function of a_j, and *J* is function of y_j. Armed with this great knowledge and using the preceding notation, we can write the following for the last layer of our neural network (using partial derivatives):

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}}$$

7. Since we know that $\frac{\partial a_j}{\partial w_{i,j}} = y_i$, we have the following:

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}y_i$$

If *y* is the logistic sigmoid, we'll get the same result that we have already calculated at the end of the *Logistic regression* section. We also know the cost function and we can calculate all the partial derivatives.

8. For the previous (hidden) layers, the same formula holds:

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}}$$

TIP

Even though we have several layers, we always concentrate on pairs of successive layers and, perhaps abusing the notation somewhat, we always have a "first" (or input) layer, and a "second" (or output) layer, as in the preceding diagram.

We know that $\frac{\partial a_j}{\partial w_{i,j}} = y_i$, and we also know that $\frac{\partial y_j}{\partial a_j}$ is the derivative of the activation function, which we can calculate. Then, all we need to do is calculate the derivative $\frac{\partial J}{\partial y_j}$. Let's note that this is the derivative of the error with respect to the activation function in the "second" layer. We can now calculate all the derivatives, starting from the last layer and moving backward, because the following applies:

- We can calculate this derivative for the last layer.
- We have a formula that allows us to calculate the derivative for one layer, assuming we can calculate the derivative for the next.

9. In the following equation, y_i is the output of the first layer (and input for the second), while y_j is the output of the second layer. Applying the chain rule, we have the following:

$$\frac{\partial J}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial y_i}$$

The sum over j reflects the fact that in the feedforward part, the output y_i is fed to all neurons in the second layer; therefore, they all contribute to y_i when the error is propagated backward.

10. Once again, we can calculate both $\frac{\partial y_j}{\partial a_j}$ and $\frac{\partial a_j}{\partial y_i} = w_{i,j}$; once we know $\frac{\partial J}{\partial y_j}$, we can calculate $\frac{\partial J}{\partial y_i}$. Since we can calculate $\frac{\partial J}{\partial y_j}$ for the last layer, we can move backward and calculate $\frac{\partial J}{\partial y_i}$ for any layer, and therefore $\frac{\partial J}{\partial w_{i,j}}$ for any layer.
11. To summarize, if we have a sequence of layers where the following applies:

$$y_i \rightarrow y_j \rightarrow y_k$$

We then have these two fundamental equations:

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}}$$

$$\frac{\partial J}{\partial y_j} = \sum_k \frac{\partial J}{\partial y_k}\frac{\partial y_k}{\partial y_j}$$

By using these two equations, we can calculate the derivatives for the cost with respect to each layer. If we set $\delta_j = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}$, then δ_j represents the variation in cost with respect to the activation value, and we can think of δ_j as the error at neuron y_j.

12. We can rewrite these equations as follows:

$$\frac{\partial J}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial y_i} = \sum_j \delta_j w_{i,j}$$

This implies that $\delta_i = (\sum_j \delta_j w_{i,j})\frac{\partial y_i}{\partial a_i}$. These two equations give an alternate view of backpropagation, as there is a variation in cost with respect to the activation value.

13. It provides a formula to calculate this variation for any layer once we know the variation for the following layer:

$$\delta_j = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}$$

$$\delta_i = (\sum_j \delta_j w_{i,j})\frac{\partial y_i}{\partial a_i}$$

14. We can combine these equations to show the following:

$$\frac{\partial J}{\partial w_{i,j}} = \delta_j \frac{\partial a_j}{\partial w_{i,j}} = \delta_j y_i$$

15. The update rule for the weights of each layer is given by the following equation:

$$w_{i,j} \to w_{i,j} - \eta \delta_j y_i$$

Code example of a neural network for the XOR function

In this section, we'll create a simple network with one hidden layer, which solves the XOR function. As we mentioned at the end of the previous chapter, the XOR function is a linearly inseparable problem, hence the need for a hidden layer. The source code will allow you to easily modify the number of layers and the number of neurons per layer, so you can try a number of different scenarios. We'll not use any ML libraries. Instead, we'll implement them from scratch only with the help of `numpy`. We'll also use `matplotlib` to visualize the results:

1. With that, let's start by importing these libraries:

```
import matplotlib.pyplot as plt
import numpy
from matplotlib.colors import ListedColormap
```

2. Next, we define the activation function and its derivative (we use `tanh(x)` in this example):

```
def tanh(x): return (1.0 - numpy.exp(-2*x))/(1.0 + numpy.exp(-2*x))
def tanh_derivative(x):
    return (1 + tanh(x))*(1 - tanh(x))
```

3. Then, we start the definition of the `NeuralNetwork` class:

```
class NeuralNetwork:
```

Because of the Python syntax, anything inside the `NeuralNetwork` class will have to be indented.

4. First, we define the `__init__` initializer of `NeuralNetwork`:
    - `net_arch` is a one-dimensional array containing the number of neurons for each layer. For example [2, 4, and 1] means an input layer with two neurons, a hidden layer with four neurons, and an output layer with one neuron. Since we are studying the `XOR` function, the input layer will have two neurons, and the output layer will only have one neuron.

5. We also set the activation function to the hyperbolic tangent, and we will then define its derivative.
6. Finally, we initialize the network weights with random values in the range (-1, 1), as demonstrated in the following code block:

```
# net_arch consists of a list of integers, indicating
# the number of neurons in each layer
def __init__(self, net_arch):
    self.activation_func = tanh
    self.activation_derivative = tanh_derivative
    self.layers = len(net_arch)
    self.steps_per_epoch = 1000
    self.net_arch = net_arch

    # initialize the weights with random values in the range
(-1,1)
    self.weights = []
    for layer in range(len(net_arch) - 1):
        w = 2 * numpy.random.rand(net_arch[layer] + 1,
net_arch[layer + 1]) - 1
        self.weights.append(w)
```

7. Next, we need to define the `fit` function, which will train our network.

8. First, we add `1` to the input data (the always-on bias neuron) and set up the code to print the result at the end of each epoch to keep a track of our progress.
9. In the last line, `nn` represents the `NeuralNetwork` class and `predict` is the function in the `NeuralNetwork` class that we'll define later:

```
def fit(self, data, labels, learning_rate=0.1, epochs=10):
    """
    :param data: data is the set of all possible pairs of
booleans
                 True or False indicated by the integers 1 or 0
                 labels is the result of the logical operation
'xor'
                 on each of those input pairs
    :param labels: array of 0/1 for each datum
    """

    # Add bias units to the input layer
    ones = numpy.ones((1, data.shape[0]))
    Z = numpy.concatenate((ones.T, data), axis=1)
    training = epochs * self.steps_per_epoch
    for k in range(training):
        if k % self.steps_per_epoch == 0:
            # print ('epochs:', k/self.steps_per_epoch)
            print('epochs: {}'.format(k /
self.steps_per_epoch))
            for s in data:
                print(s, nn.predict(s))
```

10. Next, we select a random sample from the training set and propagate it forward through the network so that we can calculate the error between the network output and the target data:

```
            sample = numpy.random.randint(data.shape[0])
            y = [Z[sample]]

            for i in range(len(self.weights) - 1):
                activation = numpy.dot(y[i], self.weights[i])
                activation_f = self.activation_func(activation)
                # add the bias for the next layer
                activation_f = numpy.concatenate((numpy.ones(1),
    numpy.array(activation_f)))
                y.append(activation_f)

            # last layer
            activation = numpy.dot(y[-1], self.weights[-1])
            activation_f = self.activation_func(activation)
            y.append(activation_f)
```

11. Now that we have the error, we can propagate it backward, so we can update the weights. We'll use stochastic gradient descent to update the weights (that is, we are going to update the weights after each step):

```
            # error for the output layer
            error = labels[sample] - y[-1]
            delta_vec = [error * self.activation_derivative(y[-1])]

            # we need to begin from the back from the next to last
    layer
            for i in range(self.layers - 2, 0, -1):
                error = delta_vec[-1].dot(self.weights[i][1:].T)
                error = error * self.activation_derivative(y[i][1:])
                delta_vec.append(error)

            # reverse
            # [level3(output)→level2(hidden)] ⇒
    [level2(hidden)→level3(output)]
            delta_vec.reverse()

            # backpropagation
            # 1. Multiply its output delta and input activation
```

```
        #    to get the gradient of the weight.
        # 2. Subtract a ratio (percentage) of the gradient from the
weight
        for i in range(len(self.weights)):
            layer = y[i].reshape(1, nn.net_arch[i] + 1)

            delta = delta_vec[i].reshape(1, nn.net_arch[i + 1])
            self.weights[i] += learning_rate * layer.T.dot(delta)
```

12. This concludes the training phase of the network. We'll now write a predict function to check the results, which returns the network output:

```
    def predict(self, x):
 val = numpy.concatenate((numpy.ones(1).T, numpy.array(x)))
 for i in range(0, len(self.weights)):
 val = self.activation_func(numpy.dot(val, self.weights[i]))
 val = numpy.concatenate((numpy.ones(1).T, numpy.array(val)))

 return val[1]
```

13. Finally, we'll write a function, which plots the lines separating the classes, based on the input variables (we'll see the plots at the end of the section):

```
    def plot_decision_regions(self, X, y, points=200):
        markers = ('o', '^')
        colors = ('red', 'blue')
        cmap = ListedColormap(colors)

        x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1

        # To produce zoomed-out figures, you can replace the
preceding 2 lines with:
        # x1_min, x1_max = -10, 11
        # x2_min, x2_max = -10, 11

        resolution = max(x1_max - x1_min, x2_max - x2_min) /
float(points)

        xx1, xx2 = numpy.meshgrid(numpy.arange(x1_min,
                                               x1_max,
                                               resolution),
                                  numpy.arange(x2_min, x2_max,
resolution))
        input = numpy.array([xx1.ravel(), xx2.ravel()]).T
        Z = numpy.empty(0)
        for i in range(input.shape[0]):
            val = nn.predict(numpy.array(input[i]))
```

```
                if val < 0.5:
                    val = 0
                if val >= 0.5:
                    val = 1
                Z = numpy.append(Z, val)

            Z = Z.reshape(xx1.shape)

            plt.pcolormesh(xx1, xx2, Z, cmap=cmap)
            plt.xlim(xx1.min(), xx1.max())
            plt.ylim(xx2.min(), xx2.max())
            # plot all samples

            classes = ["False", "True"]

            for idx, cl in enumerate(numpy.unique(y)):
                plt.scatter(x=X[y == cl, 0],
                            y=X[y == cl, 1],
                            alpha=1.0,
                            c=colors[idx],
                            edgecolors='black',
                            marker=markers[idx],
                            s=80,
                            label=classes[idx])

            plt.xlabel('x-axis')
            plt.ylabel('y-axis')
            plt.legend(loc='upper left')
            plt.show()
```

14. In the following code block, we can see the code to run the entire process:

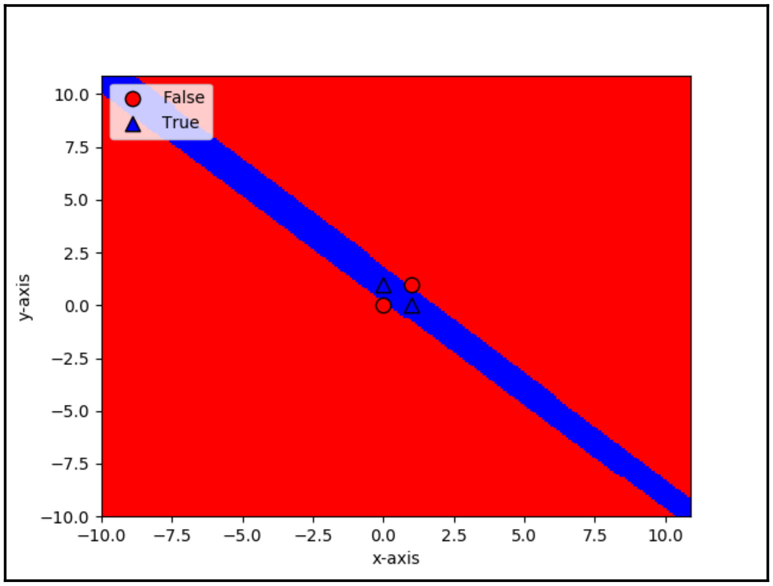
```
if __name__ == '__main__':
    numpy.random.seed(0)

    # Initialize the NeuralNetwork with 2 input, 2 hidden, and 1
output neurons
    nn = NeuralNetwork([2, 2, 1])

    X = numpy.array([[0, 0],
                     [0, 1],
                     [1, 0],
                     [1, 1]])

    y = numpy.array([0, 1, 1, 0])

    nn.fit(X, y, epochs=10)
```

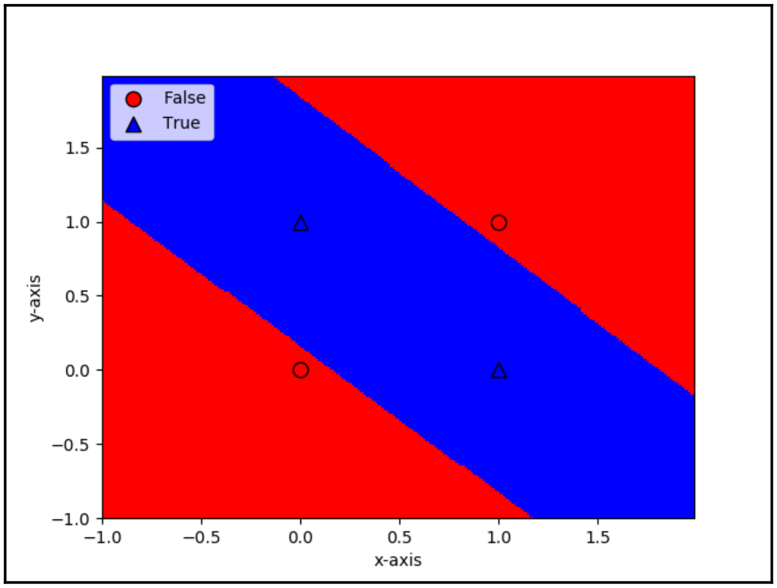
```
print("Final prediction")
for s in X:
    print(s, nn.predict(s))

nn.plot_decision_regions(X, y)
```

We use `numpy.random.seed(0)` to ensure that the weight initialization is consistent across runs, so we'll be able to compare results, but it's not necessary for the implementation of the neural net.

In the following diagrams, you can see how the `nn.plot_decision_regions function` method plots the hypersurfaces, which separate the classes. The circles represent the network output for the (**True**, **True**) and (**False**, **False**) inputs, while the triangles represent the (**True**, **False**) and (**False**, **True**) inputs for the `XOR` function:

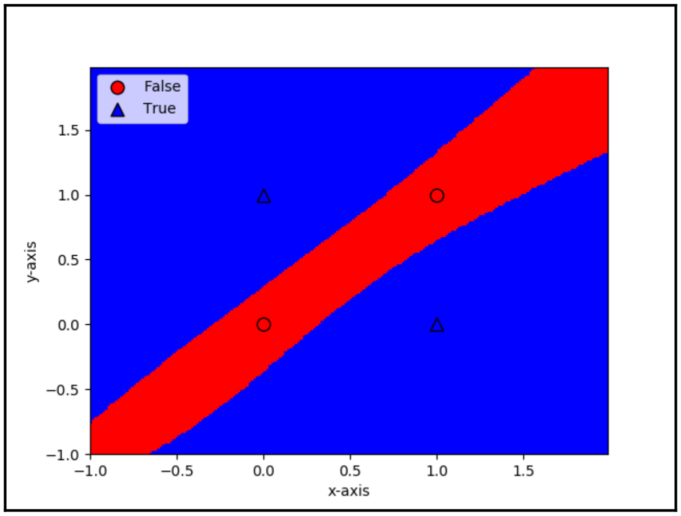
The following diagram represents the output:



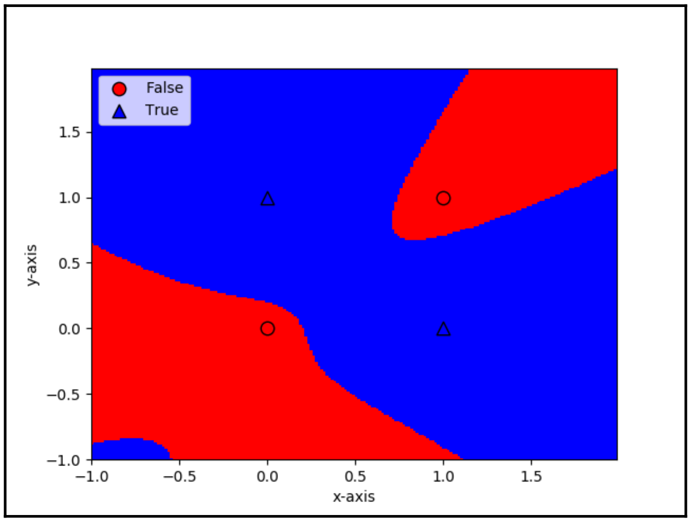


This is the same diagram, the top one zooming out, and the bottom one zooming in, on the selected inputs. The neural network learns to separate those points creating a band containing the two `True` output values. You can generate the zoomed-out image by modifying the `x1_min`, `x1_max`, `x2_min`, and `x2_max` variables in the `lot_decision_regions` function.

Networks with different architectures can produce different separating regions. We can try different combinations of hidden layers when we instantiate the network. When we build the default network, `nn = NeuralNetwork([2,2,1])`, the first and last values (2 and 1) represent the input and output layers and cannot be modified, but we can add different numbers of hidden layers with different numbers of neurons. For example, `([2,4,3,1])` will represent a 3-layer neural network, with four neurons in the first hidden layer and three neurons in the second hidden layer. You'll be able to see that while the network finds the right solution, the curves separating the regions will be different, depending on the chosen architecture.Now, `nn = NeuralNetwork([2,4,3,1])` will produce the following separation:



And here is the separation for `nn = NeuralNetwork([2,4,1])`:



The architecture of the neural network defines the way the network goes about solving the problem at hand, and different architectures provide different approaches (though they may all give the same result). We are now ready to start looking more closely at what deep neural nets are and their applications.

Summary

In this chapter, we introduced neural networks in detail and we mentioned their success vis-à-vis other competing algorithms. Neural networks are comprised of interconnected neurons (or units), where the weights of the connections characterize the strength of the communication between different neurons. We discussed different network architectures, and how a neural network can have many layers, and why inner (hidden) layers are important. We explained how the information flows from the input to the output by passing from each layer to the next based on the weights and the activation function, and finally, we showed how to train neural networks, that is, how to adjust their weights using gradient descent and backpropagation.

In the next chapter, we'll continue discussing deep neural networks, and we'll explain in particular the meaning of *deep* in deep learning, and that it not only refers to the number of hidden layers in the network, but to the quality of the learning of the network. For this purpose, we'll show how neural networks learn to recognize features and put them together as representations of larger objects. We'll also describe a few important deep learning libraries, and finally, we'll provide a concrete example where we can apply neural networks to handwritten digit recognition.

3
Deep Learning Fundamentals

In this chapter, we will introduce **deep learning**(**DL**) and **deep neural networks** (**DNNs**), that is, neural networks with multiple hidden layers. You may wonder what the point of using more than one hidden layer is, given the universal approximation theorem. This is in no way a naive question, and for a long time neural networks were used in that way. Without going into too much detail, one reason is that approximating a complex function might require a huge number of neurons in the hidden layer, making it impractical to use. There is also another, more important, reason for using deep networks, which is not directly related to the number of hidden layers, but to the level of learning. A deep network does not simply learn to predict output Y given input X; it also understands basic features of the input. It's able to learn abstractions of features of input examples, to understand the basic characteristics of the examples, and to make predictions based on those characteristics. This is a level of abstraction that is missing in other basic **machine learning**(**ML**) algorithms and in shallow neural networks.

In this chapter, we will cover the following topics:

- Introduction to deep learning
- Fundamental deep learning concepts
- Deep learning algorithms
- Applications of deep learning
- The reasons for deep learning's popularity
- Introducing popular open source libraries

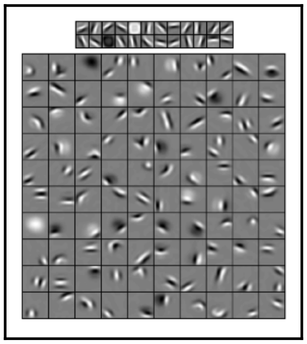
Introduction to deep learning

In 2012, Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton published a milestone paper titled *ImageNet Classification with Deep Convolutional Neural Networks* `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`. The paper describes their use of neural networks to win the ImageNet competition of the same year, which we mentioned in `Chapter 2`, *Neural Networks*. At the end of their paper, they wrote the following:

> *"It is notable that our network's performance degrades if a single convolutional layer is removed. For example, removing any of the middle layers results in a loss of about 2% for the top-1 performance of the network. So the depth really is important for achieving our results."*

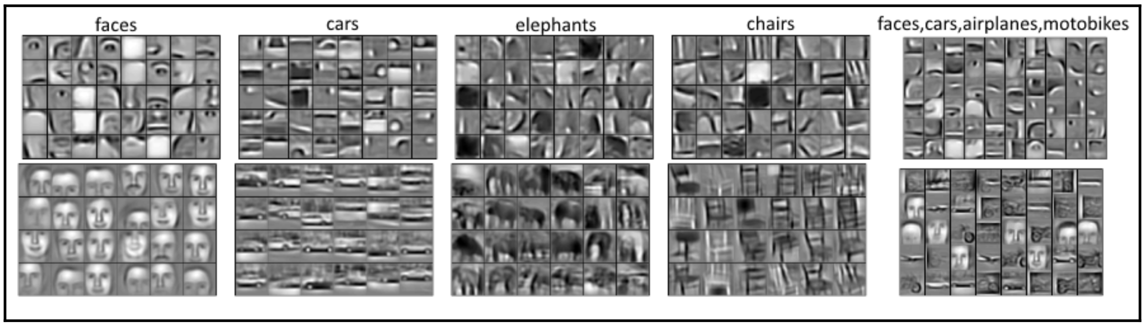
They clearly mention the importance of the number of hidden layers present in deep networks. Krizheysky, Sutskever, and Hilton talk about convolutional layers, which we will not discuss until `Chapter 4`, *Computer Vision With Convolutional Networks*, but the basic question remains: what do those hidden layers do?

A typical English saying is a **picture is worth a thousand words**. Let's use this approach to understand what deep learning is. We'll use images from the highly-cited paper *Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations* (`https://ai.stanford.edu/~ang/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf`). In *Proceedings of the International Conference on Machine Learning* (*ICML*) (2009) by H. Lee, R. Grosse, R. Ranganath, and A. Ng, the authors train a neural network with pictures of different categories of either objects or animals. In the following screenshot, we can see how the different layers of the network learn different characteristics of the input data. In the first layer, the network learns to detect some small basic features such as lines and edges, which are common for all images in all categories:



The first layer weights (top) and the second layer weights (bottom) after training

But in the next layers, which we can see in the preceding screenshot, it combines those lines and edges to compose more complex features that are specific for each category. In the first row of the bottom-left image, we can see how the network can detect different features of human faces such as eyes, noses, and mouths. In the case of cars, these would be wheels, doors, and so on, as seen in the second image from the left in the following image. These features are **abstract**, that is, the network has learned the generic shape of a feature (such as a mouth or a nose) and can detect this feature in the input data, despite the variations it might have:



Columns 1-4 represent the second layer (top) and third layer (bottom) weights learned for a specific object category (class). Column 5 represents the weights learned for a mixture of four object categories (faces, cars, airplanes, and motobikes)

In the second row of the preceding image, we can see how, in the deeper layers, the network combines these features in even more complex ones, such as faces and whole cars. A strength of deep neural networks is that they can learn these high-level abstract representations by themselves, deducting them from the training data.

Fundamental deep learning concepts

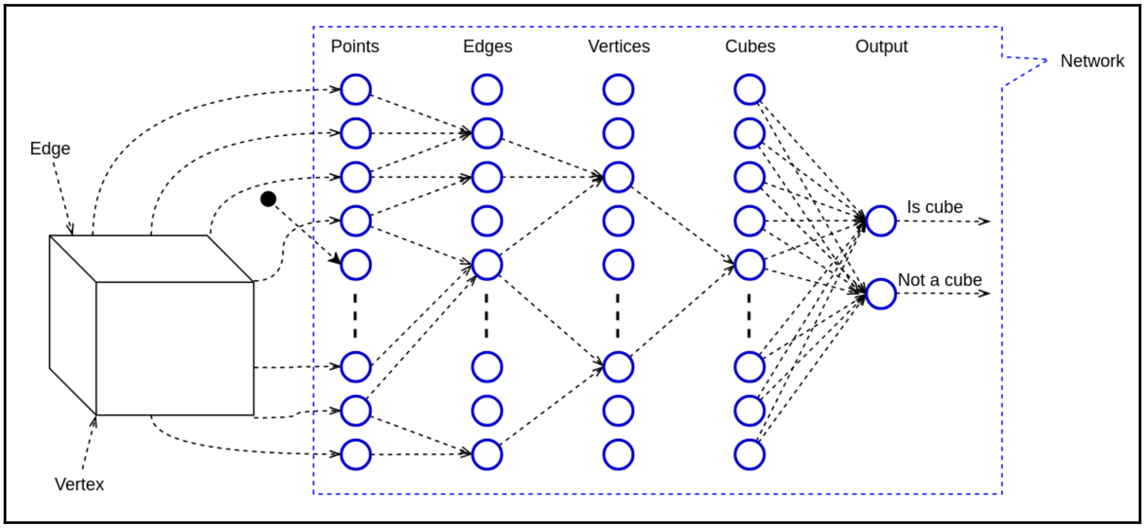
In 1801, Joseph Marie Charles invented the Jacquard loom. Charles named the Jacquard, hence the name of its invention, was not a scientist, but simply a merchant. The Jacquard loom used a set of punched cards, where each card represented a pattern to be reproduced on the loom. At the same time, each card was an abstract representation of that pattern. Punched cards have been used since, for example, in the tabulating machine invented by Herman Hollerith in 1890, or in the first computers as a means to input code. In the tabulating machine, the cards were simply abstractions of samples to be fed into the machine to calculate statistics on a population. But in the Jacquard loom, their use was subtler, and each card represented the abstraction of a pattern that could be combined with others to create more complex patterns. The punched card is an abstract representation of a feature of reality, the final weaved design.

In a way, the Jacquard loom sowed the seeds of what deep learning is today, the definition of a reality through the representations of its features. A deep neural network does not simply recognize what makes a cat a cat, or a squirrel a squirrel, but it understands what features are present in a cat and a squirrel respectively. It learns to design a cat or a squirrel using those features. If we were to design a weaving pattern in the shape of a cat using a Jacquard loom, we would need to use punched cards that have whiskers on the nose, such as those of a cat, and an elegant and slender body. Conversely, if we were to design a squirrel, we would need to use the punched card that makes a furry tail. A deep network that learns basic representations of its output can make classifications using the assumptions it has made. For example, if there is no furry tail, it will probably not be a squirrel, but rather a cat. In this way, the amount of information the network learns is much more complete and robust, and the most exciting part is that deep neural networks learn to do this automatically.

Feature learning

To illustrate how deep learning works, let's consider the task of recognizing a simple geometric figure, for example, a cube, as seen in the following diagram. The cube is composed of edges (or lines), which intersect in vertices. Let's say that each possible point in the three-dimensional space is associated with a neuron (forget for a moment that this will require an infinite number of neurons). All the points/neurons are in the first (input) layer of a multi-layer feed-forward network. An input point/neuron is active if the corresponding point lies on a line. The points/neurons that lie on a common line (edge) have strong positive connections to a single common edge/neuron in the next layer. Conversely, they have negative connections to all other neurons in the next layer. The only exception are the neurons that lie on the vertices. Each such neuron lies simultaneously on three edges, and is connected to its three corresponding neurons in the subsequent layer.

Now we have two hidden layers, with different levels of abstraction—the first for points and the second for edges. But this is not enough to encode a whole cube in the network. Let's try with another layer for vertices. Here, each three active edge/neurons of the second layer, which form a vertex, have a significant positive connection to a single common vertex/neuron of the third layer. Since an edge of the cube forms two vertices, each edge/neuron will have positive connections to two vertices/neurons and negative connections to all others. Finally, we'll introduce the last hidden layer (cube). The four vertices/neurons forming a cube will have positive connections to a single cube/neuron from the cube/layer:



An abstraction of a neural network representing a cube. Different layers encode features with different levels of abstraction

The cube representation example is oversimplified, but we can draw several conclusions from it. One of them is that deep neural networks lend themselves well to hierarchically organized data. For example, an image consists of pixels, which form lines, edges, regions, and so on. This is also true for speech, where the building blocks are called **phonemes**; as well as text, where we have characters, words, and sentences.

In the preceding example, we dedicated layers to specific cube features deliberately, but in practice, we wouldn't do that. Instead, a deep network will "discover" features automatically during training. These features might not be immediately obvious and, in general, wouldn't be interpretable by humans. Also, we wouldn't know the level of the features encoded in the different layers of the network. Our example is more akin to classic machine learning algorithms, where the user has to use his/her own experience to select what they think are the best features. This process is called **feature engineering**, and it can be labor-intensive and time-consuming. Allowing a network to automatically discover features is not only easier, but those features are highly abstract, which makes them less sensitive to noise. For example, human vision can recognize objects of different shapes, sizes, in different lighting conditions, and even when their view is partly obscured. We can recognize people with different haircuts, facial features, and even when they wear a hat or a scarf that covers their mouth. Similarly, the abstract features the network learns will help it to recognize faces better, even in more challenging conditions.

Deep learning algorithms

In `chapter 3`, *Deep Learning Fundamentals*, we gave an easy-to-understand introduction to deep learning. In the next section, *Deep networks*, we'll give a more precise definition of the key concepts that will be thoroughly introduced in the coming chapters.

Deep networks

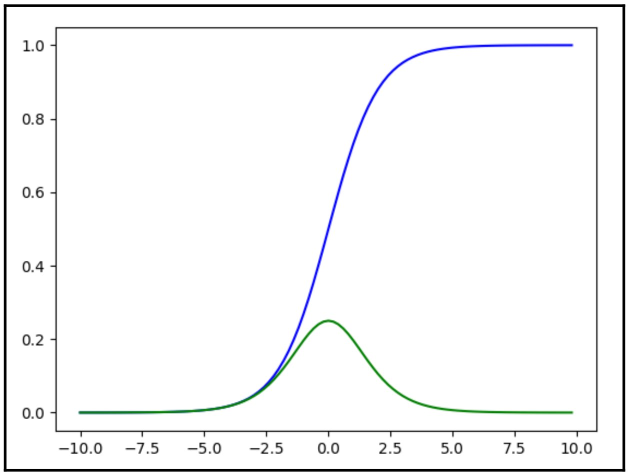
We could define deep learning as a class of machine learning techniques, where information is processed in hierarchical layers to understand representations and features from data in increasing levels of complexity. In practice, all deep learning algorithms are neural networks, which share some common basic properties. They all consist of interconnected neurons that are organized in layers. Where they differ is network architecture (or the way neurons are organized in the network), and sometimes in the way they are trained. With that in mind, let's look at the main classes of neural networks. The following list is not exhaustive, but it represents the vast majority of algorithms in use today:

- **Multi-layer perceptrons** (**MLPs**): A neural network with feed-forward propagation, fully-connected layers, and at least one hidden layer. We introduced MLPs in `Chapter 2`, *Neural Networks*.
- **Convolutional neural networks** (**CNNs**): A CNN is a feedforward neural network with several types of special layers. For example, convolutional layers apply a filter to the input image (or sound) by sliding that filter all across the incoming signal, to produce an *n*-dimensional activation map. There is some evidence that neurons in CNNs are organized similarly to how biological cells are organized in the visual cortex of the brain. We've mentioned CNNs several times up to now, and that's not a coincidence – today, they outperform all other ML algorithms on a large number of computer vision and NLP tasks.
- **Recurrent networks**: This type of network has an internal state (or memory), which is based on all or part of the input data already fed to the network. The output of a recurrent network is a combination of its internal state (memory of inputs) and the latest input sample. At the same time, the internal state changes, to incorporate newly input data. Because of these properties, recurrent networks are good candidates for tasks that work on sequential data, such as text or time-series data. We'll discuss recurrent networks in `Chapter 7`, *Recurrent Neural Networks and Language Models.*

- **Autoencoders**: A class of unsupervised learning algorithms, in which the output shape is the same as the input that allows the network to better learn basic representations. We'll discuss autoencoders when we talk about generative deep learning, in `Chapter 6`, *Generating Images with GANs and VAEs.*

A brief history of contemporary deep learning

In addition to the aforementioned models, the first edition of this book included networks such as **Restricted Boltzmann Machines** (**RBMs**) and DBNs. They were popularized by Geoffrey Hinton, a Canadian scientist, and one of the most prominent deep learning researchers. Back in 1986, he was also one of the inventors of backpropagation. RBMs are a special type of generative neural network, where the neurons are organized into two layers, namely, visible and hidden. Unlike feed-forward networks, the data in an RBM can flow in both directions – from visible to hidden units, and vice versa. In 2002, Prof. Hinton introduced contrastive divergence, which is an unsupervised algorithm for training RBMs. And in 2006, he introduced Deep Belief Nets, which are deep neural networks that are formed by stacking multiple RBMs. Thanks to their novel training algorithm, it was possible to create a DBN with more hidden layers than had previously been possible. To understand this, we should explain why it was so difficult to train deep neural networks prior to that. In the past, the activation function of choice was the logistic sigmoid, shown in the following chart:



Logistic sigmoid (blue) and its derivative (green)

We now know that, to train a neural network, we need to compute the derivative of the activation function (along with all the other derivatives). The sigmoid derivative has significant value in a very narrow interval, centered around 0 and converges towards 0 in all other cases. In networks with many layers, it's highly likely that the derivative would converge to 0, when propagated to the first layers of the network. Effectively, this means we cannot update the weights in these layers. This is a famous problem called **vanishing gradients** and (along with a few other issues), which prevents the training of deep networks. By stacking pre-trained RBMs, DBNs were able to alleviate (but not solve) this problem.

But training a DBN is not easy. Let's look at the following steps:

- First, we have to train each RBM with contrastive divergence, and gradually stack them on top of each other. This phase is called **pre-training**.
- In effect, pre-training serves as a sophisticated weight initialization algorithm for the next phase, called **fine-tuning**. With fine-tuning, we transform the DBN in a regular multi-layer perceptron and continue training it using supervised backpropagation, in the same way we saw in `Chapter 2`, *Neural Networks*.

However, thanks to some algorithmic advances, it's now possible to train deep networks using plain old backpropagation, thus effectively eliminating the pre-training phase. We will discuss these improvements in the coming sections, but for now, let's just say that they rendered DBNs and RBMs obsolete. DBNs and RBMs are, without a doubt, interesting from a research perspective, but are rarely used in practice anymore. Because of this, we will omit them from this edition.

Training deep networks

As we mentioned in `chapter 2`, *Neural Networks,* we can use different algorithms to train a neural network. But in practice, we almost always use **Stochastic Gradient Descent** (**SGD**) and backpropagation, which we introduced in `Chapter 2`, *Neural Networks*. In a way, this combination has withstood the test of time, outliving other algorithms, such as DBNs. With that said, gradient descent has some extensions worth discussing.

In the following section, we'll introduce **momentum**, which is an effective improvement over the vanilla gradient descent. You may recall the weight update rule that we introduced in `Chapter 2`, *Neural Networks*:

1. $w \to w - \lambda\nabla(J(w))$, where λ is the learning rate.

   To include momentum, we'll add another parameter to this equation.

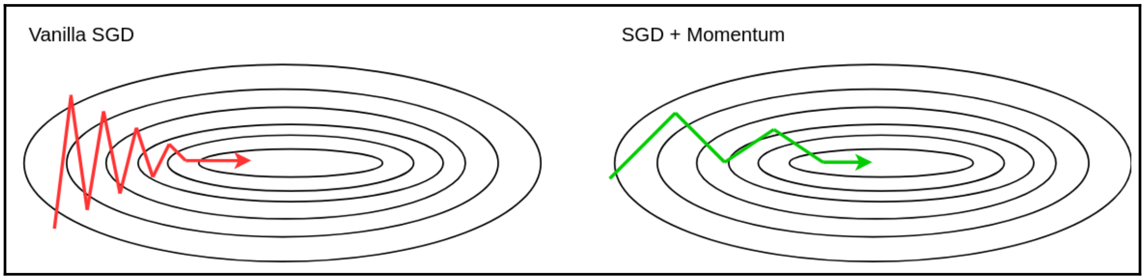
2. First, we'll calculate the weight update value:

$$\triangle w \to \mu \triangle w - \lambda(\nabla J(w))$$

3. Then, we'll update the weight:

$$w \to w + \triangle w$$

From the preceding equation, we see that the first component, $\mu \triangle w$, is the momentum. The $\triangle w$ represents the previous value of the weight update and μ is the coefficient, which will determine how much the new value depends on the previous ones. To explain this, let's look at the following diagram, where you will see a comparison between vanilla SGD and SGD with momentum. The concentric ellipses represent the surface of the error function, where the innermost ellipse is the minimum and the outermost the maximum. Think of the loss function surface as the surface of a hill. Now, imagine that we are holding a ball at the top of the hill (maximum). If we drop the ball, thanks to Earth's gravity, it will start rolling toward the bottom of the hill (minimum). The more distance it travels, the more its speed will increase. In other words, it will gain momentum (hence the name of the optimization). As a result, it will reach the bottom of the hill faster. If, for some reason, gravity didn't exist, the ball would roll at its initial speed and it would reach the bottom more slowly:



A comparison between vanilla SGD and SGD + momentum

In your practice, you may encounter other gradient descent optimizations, such as Nesterov momentum, ADADELTA `https://arxiv.org/abs/1212.5701`, RMSProp `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, and Adam `https://arxiv.org/abs/1412.6980`. Some of these will be discussed in later chapters of the book.

Applications of deep learning

Machine learning in general, and deep learning in particular, are producing more and more astonishing results in terms of the quality of predictions, feature detection, and classification. Many of these recent results have made the news. Such is the pace of progress, that some experts are worrying that machines will soon be more intelligent than humans. But I hope that any such fears you might have will be alleviated after you have read this book. For better or worse, we're still far from human-level intelligence.

In `Chapter 2`, *Neural Networks*, we mentioned how deep learning algorithms have occupied the leaderboard of the ImageNet competition. They are successful enough to make the jump from academia to industry. Let's now talk about some real-world use cases of deep learning:

- Nowadays, new cars have a suite of safety and convenience features that aim to make the driving experience safer and less stressful. One such feature is automated emergency braking if the car sees an obstacle. Another one is lane-keeping assist, which allows the vehicle to stay in its current lane without the driver needing to make corrections with the steering wheel. To recognize lane markings, other vehicles, pedestrians, and cyclists, these systems use a forward-facing camera. One of the most prominent suppliers of such systems, Mobileye `https://www.mobileye.com/`, has produced custom chips that use CNNs to detect these objects on the road ahead. To give you an idea of the importance of this sector, in 2017, Intel acquired Mobileye for $15.3 billion. This is not an outlier, and Tesla's famous Autopilot system also relies on CNNs to achieve the same results. In fact, the director of AI at Tesla, Andrej Karpathy `https://cs.stanford.edu/people/karpathy/`, is a well-known researcher in the field of deep learning. We can speculate that future autonomous vehicles will also use deep networks for computer vision.
- Both Google's Vision API `https://cloud.google.com/solutions/image-search-app-with-cloud-vision` and Amazon's Rekognition `https://aws.amazon.com/rekognition/faqs/` service use deep learning models to provide various computer vision capabilities. These include recognizing and detecting objects and scenes in images, text recognition, face recognition, and so on.

- If these APIs are not enough, you can run your own models in the cloud. For example, you can use Amazon's AWS Deep Learning AMIs (Amazon Machine Images), `https://aws.amazon.com/machine-learning/amis/` , virtual machines that come configured with some of the most popular DL libraries. Google offers a similar service with their Cloud AI, `https://cloud.google.com/products/ai/`, but they've gone one step further. They created Tensor Processing Units TPUs,(`https://cloud.google.com/tpu/`)– microprocessors, optimized for fast neural network operations such as matrix multiplication and activation functions.
- Deep learning has a lot of potential for medical applications. However, strict regulatory requirements, as well as patient data confidentiality have slowed down its adoption. Nevertheless, we'll identify two areas in which deep learning could have a high impact:
    - Medical imaging is an umbrella term for various non-invasive methods of creating visual representations of the inside of the body. Some of these include **Magnetic resonance images** (**MRIs**), ultrasound, **Computed Axial Tomography** (**CAT**) scans, X-rays, and histology images. Typically, such an image is analyzed by a medical professional to determine the patient's condition. Computer-aided diagnosis, and computer vision in particular, can help specialists by detecting and highlighting important features of images. For example, to determine the degree of malignancy of colon cancer, a pathologist would have to analyze the morphology of the glands, using histology imaging. This is a challenging task, because morphology can vary greatly. A deep neural network could segment the glands from the image automatically, leaving the pathologist to verify the results. This would reduce the time needed for analysis, making it cheaper and more accessible.
    - Another medical area that could benefit from deep learning is the analysis of medical history records. When a doctor diagnoses a patient's condition and prescribes treatment, they consult the patient's medical history first. A deep learning algorithm could extract the most relevant and important information from those records, even if they are handwritten. In this way, the doctor's job would be made easier, and the risk of errors would also be reduced.
- Google's Neural Machine Translation API `https://arxiv.org/abs/1611.04558` uses – you guessed it – deep neural networks for machine translation.

- Google Duplex is another impressive real-world demonstration of deep learning. It's a new system that can carry out natural conversations over the phone. For example, it can make restaurant reservations on a user's behalf. It uses deep neural networks, both to understand the conversation, and also to generate realistic, human-such as replies.
- Siri (`https://machinelearning.apple.com/2017/10/01/hey-siri.html`), Google Assistant, and Amazon Alexa (`https://aws.amazon.com/deep-learning/`)rely on deep networks for speech recognition.
- Finally, AlphaGo is an **artificial intelligence** (**AI**) machine based on deep learning, which made the news in 2016 by beating the world Go champion, Lee Sedol. AlphaGo had already made the news, in January 2016, when it beat the European champion, Fan Hui. Although, at the time, it seemed unlikely that it could go on to beat the world champion. Fast-forward a couple of months and AlphaGo was able to achieve this remarkable feat by sweeping its opponent in a 4-1 victory series. This was an important milestone, because Go has many more possible game variations than other games, such as chess, and it's impossible to consider every possible move in advance. Also, unlike chess, in Go it's very difficult to even judge the current position or value of a single stone on the board. In 2017, DeepMind released an updated version of AlphaGo called AlphaZero(`https://arxiv.org/abs/1712.01815`).

TIP

Neural networks are representatives of **pattern recognition**, one possible approach to artificial intelligence. Pattern recognition is the process of automating the recognition of patterns and regularities in data. In other words, in pattern recognition, the computer uses machine learning to learn features of data by itself. The opposite approach would be to use hand-crafted rules (hardcoded by a human).

With this short list, we aimed to cover the main areas in which deep learning is applied, such as computer vision, natural language processing, speech recognition, and reinforcement learning. This list is not exhaustive, as there are many other uses for deep learning algorithms. Still, I hope this has been enough to spark your interest.

The reasons for deep learning's popularity

If you've followed machine learning for some time, you may have noticed that many DL algorithms are not new. We dropped some hints about this in the *A brief history of contemporary deep learning* section, but let's see some more examples now. Multilayer perceptrons have been around for nearly 50 years. Backpropagation has been discovered a couple of times, but finally gained recognition in 1986. Yann LeCun, a famous computer scientist, perfected his work on convolutional networks in the 1990s. In 1997, Sepp Hochreiter and Jürgen Schmidhuber invented long short-term memory, a type of recurrent neural network still in use today. In this section, we'll try to understand why we have AI summer now, and why we only had AI winters(`https://en.wikipedia.org/wiki/AI_winter`) before.

The first reason is, today, we have a lot more data than in the past. The rise of the internet and software in different industries has generated a lot of computer-accessible data. We also have more benchmark datasets, such as ImageNet. With this comes the desire to extract value from that data by analyzing it. And, as we'll see later, deep learning algorithms work better when they are trained with a lot of data.

The second reason is the increased computing power. This is most visible in the drastically increased processing capacity of **Graphical Processing Units** (**GPUs**). Architecturally, **Central Processing Units** (**CPUs**) are composed of a few cores that can handle a few threads at a time, while GPUs are composed of hundreds of cores that can handle thousands of threads in parallel. A GPU is a highly parallelizable unit, compared to a CPU, which is mainly a serial unit. Neural networks are organized in such a way as to take advantage of this parallel architecture. Let's see why.

As we now know, neurons from a network layer are not connected to neurons from the same layer. Therefore, we can compute the activation of each neuron in that layer independently from the others. This means that we can compute their activation in parallel. To better understand this, let's use two sequential fully-connected layers, where the input layer has *n* neurons and the second layer has *m* neurons. The activation value for each neuron is $a_j = \sum_i w_{ij} x_i$. If we express it in vector form, we have $a_j = f(\vec{x} \cdot \vec{w}_j)$, where *x* and *w* are n-dimensional vectors (because the input size is *n*). We can combine the weight vectors for all neurons in the second layer in an *n* by *m* dimensional matrix, *W*. Now, let's recall that we train the network using mini batches of inputs with an arbitrary size, *k*. We can represent one mini batch of input vectors as a *k* by *n* dimensional matrix, *X*. We'll optimize the execution by propagating the whole mini batch through the network as a single input. Putting it all together, we can compute all of the neuron activations of the second layer, *Y*, for all input vectors in the mini batch, as a matrix multiplication - *Y* = *XW*. This highly parallelizable operation can fully utilize the advantages of the GPU.

Furthermore, CPUs are optimized for latency and GPUs are optimized for bandwidth. This means that a CPU can fetch small chunks of memory very quickly, but will be slow to fetch large chunks. The GPU does the opposite. For matrix multiplication in a deep network with a lot of wide layers, bandwidth becomes the bottleneck, not latency. In addition, the L1 cache of the GPU is much faster than the L1 cache for the CPU and is also larger. The L1 cache represents the memory of the information that the program is likely to use next, and storing this data can speed up the process. Much of the memory gets reused in deep neural networks, which is why L1 cache memory is important.

But even under these favorable conditions, we still haven't addressed the issue of training deep neural networks, such as vanishing gradients. Thanks to a combination of algorithmic advances, it's now possible to the train neural networks with almost arbitrary depth with the help of the combination. These include better activation functions, **Rectified Linear Unit** (**ReLU**), better initialization of the network weights before training, new network architectures, as well as new types of regularization techniques such as **Batch normalization**.

Introducing popular open source libraries

There are many open-source libraries that allow the creation of deep neural nets in Python, without having to explicitly write the code from scratch. In this book, we'll use three of the most popular: - TensorFlow, Keras, and PyTorch. They all share some common features, as follows:

- The basic unit for data storage is the **tensor**. Consider the tensor as a generalization of a matrix to higher dimensions. Mathematically, the definition of a tensor is more complex, but in the context of deep learning libraries, they are multi-dimensional arrays of base values. A tensor is similar to a NumPy array and is made up of the following:
    - A basic data type of tensor elements. These can vary between libraries, but typically include 16-, 32-, and 64-bit float and 8-, 16-, 32-, and 64-bit integers.

    - An arbitrary number of axes (also known as the rank, order, or degree of the tensor). An 0D tensor is just a scalar value, 1D is a vector, 2D is a matrix, and so on. In deep networks, the data is propagated in batches of *n* samples. This is done for performance reasons, but it also suits the notion of stochastic gradient descent. For example, if the input data is one-dimensional, such as [0, 1], [1, 0], [0, 0], and [1, 1] for XOR values, we'll actually work with a 2D tensor `[[0, 1], [1, 0], [0, 0], [1, 1]]` to represent all of the samples in a single batch. Alternatively, two-dimensional grayscale images will be represented as a three-dimensional tensor. In the context of deep learning libraries, the first axis of the tensor represents the different samples.
    - A shape that is the size (the number of values) of each axis of the tensor. For example, the XOR tensor from the preceding example will have a shape of (4, 2). A tensor representing a batch of 32 128x128 images will have a shape of (32, 128, 128).
- Neural networks are represented as a **computational graph** of operations. The nodes of the graph represent the operations (weighted sum, activation function, and so on). The edges represent the flow of data, which is how the output of one operation serves as an input for the next one. The inputs and outputs of the operations (including the network inputs and outputs) are tensors.
- All libraries include **automatic differentiation**. This means, that all you need to do is define the network architecture and activation functions, and the library will automatically figure out all of the derivatives required for training with backpropagation.
- All libraries use Python.
- Until now, we've referred to GPUs in general, but in reality, the vast majority of deep learning projects work exclusively with NVIDIA GPUs. This is so because of the better software support NVIDIA provides. These libraries are no exception – to implement GPU operations, they rely on the `CUDA` toolkit in combination with the `cuDNN` library. cuDNN is an extension of CUDA, built specifically for deep learning applications. As was previously mentioned in the *Applications of deep learning* section, you can also run your deep learning experiments in the cloud.

For these libraries, we will quickly describe how to switch between a GPU and a CPU. Much of the code in this book can then be run on a CPU or a GPU, depending on the hardware available to the reader.

At the time of writing, the latest versions of the libraries are the following:

- `TensorFlow` 1.12.0
- `PyTorch` 1.0
- `Keras` 2.2.4

We'll use them throughout the book.

TensorFlow

TensorFlow (**TF**) (`https://www.tensorflow.org`), is the most popular deep learning library. It's developed and maintained by Google. You don't need to explicitly require the use of a GPU; rather TensorFlow will automatically try to use it if you have one. If you have more than one GPU, you must assign operations to each GPU explicitly, or only the first one will be used. To do this, you simply need to type the line that is show in the following code block:

```
with tensorflow.device("/gpu:1"):
   # model definition here
```

Here's an example:

- `"/cpu:0"`: the main CPU of your machine
- `"/gpu:0"`: the first GPU of your machine, if one exists
- `"/gpu:1"`: the second GPU of your machine, if a second exists
- `"/gpu:2"`: the third GPU of your machine, if a third exists, and so on

TensorFlow has a steeper learning curve, compared to the other libraries. You can refer to the TensorFlow documentation to learn how to use it.

Keras

Keras is a high-level neural net Python library that runs on top of **TensorFlow**, **CNTK** (`https://github.com/Microsoft/CNTK`), or **Theano**. For the purposes of this book, we'll assume that it uses TensorFlow on the backend. With **Keras**, you can perform rapid experimentation and it's relatively easy to use compared to TF. It will automatically detect an available GPU and attempt to use it. Otherwise, it will revert to the CPU. If you wish to specify the device manually, you can import **TensorFlow** and use the same code as in the previous section, *TensorFlow*:

```
with tensorflow.device("/gpu:1"):
    # Keras model definition here
```

Once again, you can refer to the online documentation for further information at `http://keras.io.`

PyTorch

PyTorch (`https://pytorch.org/`) is a deep learning library based on Torch and developed by Facebook. It is relatively easy to use, and has recently gained a lot of popularity. It will automatically select a GPU, if one is available, reverting to the CPU otherwise. If you wish to select the device explicitly, you could use the following code sample:

```
# at beginning of the script
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
...
# then whenever you get a new Tensor or Module
# this won't copy if they are already on the desired device
input = data.to(device)
model = MyModule(...).to(device)
```

Using Keras to classify handwritten digits

In this section, we'll use Keras to classify the images of the MNIST dataset. It's comprised of 70,000 examples of handwritten digits by different people. The first 60,000 are typically used for training and the remaining 10,000 for testing:

Sample of digits taken from the MNIST dataset

One of the advantages of Keras is that it can import this dataset for you without needing to explicitly download it from the web (it will download it for you):

1. Our first step will be to download the datasets using Keras:

```
from keras.datasets import mnist
```

2. Then, we need to import a few classes to use a feed-forward network:

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.utils import np_utils
```

3. Next, we'll load the training and testing data. `(X_train, Y_train)` are the training images and labels, and `(X_test, Y_test)` are the test images and labels:

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

4. We need to modify the data to be able to use it. `X_train` contains 60,000 28 x 28 pixel images, and `X_test` contains 10,000. To feed them to the network as inputs, we want to reshape each sample as a 784-pixel long array, rather than a (28,28) two-dimensional matrix. We can accomplish this with these two lines:

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

5. The labels indicate the value of the digit depicted in the images. We want to convert this into a 10-entry **one-hot encoded** vector comprised of zeroes and just one 1 in the entry corresponding to the digit. For example, 4 is mapped to [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]. Conversely, our network will have 10 output neurons:

```
classes = 10
Y_train = np_utils.to_categorical(Y_train, classes)
Y_test = np_utils.to_categorical(Y_test, classes)
```

6. Before calling our main function, we need to set the size of the input layer (the size of the `MNIST` images), the number of hidden neurons, the number of epochs to train the network, and the mini batch size:

```
input_size = 784
batch_size = 100
hidden_neurons = 100
epochs = 100
```

7. We are ready to define our network. In this case, we'll use the `Sequential` model, where each layer serves as an input to the next. In Keras, `Dense` means fully-connected layer. We'll use a network with one hidden layer, sigmoid activation, and softmax output:

```
model = Sequential([
    Dense(hidden_neurons, input_dim=input_size),
    Activation('sigmoid'),
    Dense(classes),
    Activation('softmax')
])
```

8. Keras now provides a simple way to specify the cost function (the `loss`) and its optimization, in this case, **cross-entropy** and stochastic gradient descent. We'll use the default values for learning rate, momentum, and so on:

```
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='sgd')
```

Softmax and cross-entropy

In the *Logistic regression* section of `Chapter 2`, *Neural Networks,* we learned how to apply regression to binary classification (two classes) problems. The softmax function is a generalization of this concept for multiple classes. Let's look at the following formula:

$$F(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Here, *i, j = 0, 1, 2, ... n* and x_i represent each of *n* arbitrary real values, corresponding to *n* mutually exclusive classes. The softmax "squashes" the input values in the (0, 1) interval, similar to the logistic function. But it has the additional property that the sum of all the squashed outputs adds up to 1. We can interpret the softmax outputs as a normalized probability distribution of the classes. Then, it makes sense to use a loss function, which compares the difference between the estimated class probabilities and the actual class distribution (the difference is known as cross-entropy). As we mentioned in step 5 of this section, the actual distribution is usually a one-hot-encoded vector, where the real class has a probability of 1, and all others have a probability of 0. The loss function that does this is called **cross-entropy loss**:

$$H(p,q) = -\sum_{i=1}^{n} p_i(x)\log(q_i(x))$$

Here, $q_i(x)$ is the estimated probability of the output to belong to class *i* (out of *n* total classes) and $p_i(x)$ is the actual probability. When we use one-hot-encoded target values for $p_i(x)$, only the target class has a non-zero value (1) and all others are zeros. In this case, cross entropy loss will only capture the error on the target class and will discard all other errors. For the sake of simplicity, we'll assume that we apply the formula over a single training sample.

9. We are ready to train the network. In Keras, we can do this in a simple way, with the `fit` method:

```
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=epochs,
verbose=1)
```

10. All that's left to do is to add code to evaluate the network accuracy on the test data:

    ```
    score = model.evaluate(X_test, Y_test, verbose=1) print('Test
    accuracy:', score[1])
    ```

    And that's it. The test accuracy will be about 96%, which is not a great result, but this example runs in less than 30 seconds on a CPU. We can make some simple improvements, such as a larger number of hidden neurons, or a higher number of epochs. We'll leave those experiments to you, to familiarize yourself with the code.

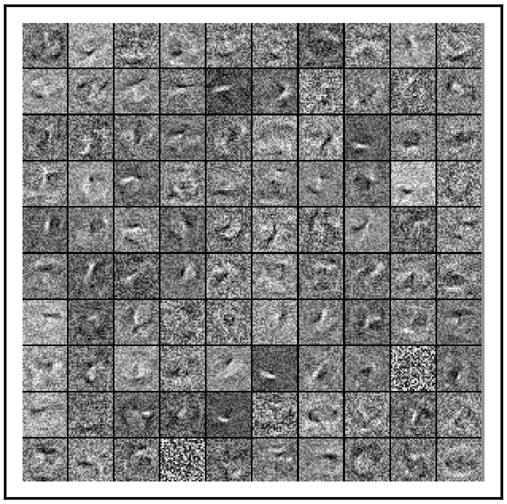
11. To see what the network has learned, we can visualize the weights of the hidden layer. The following code allows us to obtain them:

    ```
    weights = model.layers[0].get_weights()
    ```

12. To do this, we'll reshape the weights for each neuron back to a 28x28 two-dimensional array:

    ```
    import matplotlib.pyplot as plt
    import matplotlib.cm as cm
    import numpy

    fig = plt.figure()

    w = weights[0].T
    for neuron in range(hidden_neurons):
     ax = fig.add_subplot(10, 10, neuron + 1)
     ax.axis("off")
     ax.imshow(numpy.reshape(w[neuron], (28, 28)), cmap=cm.Greys_r)

    plt.savefig("neuron_images.png", dpi=300)
    plt.show()
    ```

13. And we can see the result in the following image:



Composite figure of what was learned by all the hidden neurons

For simplicity, we've aggregated the images of all neurons in a single figure that represents a composite of all neurons. Clearly, since the initial images are very small and do not have lots of details (they are just digits), the features learned by the hidden neurons are not all that interesting. But it's already clear that each neuron is learning a different shape.

Using Keras to classify images of objects

With Keras, it's easy to create neural nets, but it's also easy to download test datasets. Let's try to use the **CIFAR-10** (Canadian Institute For Advanced Research, `https://www.cs.toronto.edu/~kriz/cifar.html`) dataset instead of MNIST. It consists of 60,000 32x32 RGB images, divided into 10 classes of objects, namely: airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships, and trucks:

1. We'll import CIFAR-10 in the same way as we did MNIST:

```
from keras.datasets import cifar10
from keras.layers.core import Dense, Activation
from keras.models import Sequential
from keras.utils import np_utils
```

2. Then, we'll split the data into 50,000 training images and 10,000 testing images. Once again, we need to reshape the image to a one-dimensional array. In this case, each image has 3 color channels (red, green, and blue) of 32x32 pixels, hence 3 x32x3 = 3072:

```
(X_train, Y_train), (X_test, Y_test) = cifar10.load_data()

X_train = X_train.reshape(50000, 3072)
X_test = X_test.reshape(10000, 3072)

classes = 10
Y_train = np_utils.to_categorical(Y_train, classes)
Y_test = np_utils.to_categorical(Y_test, classes)

input_size = 3072
batch_size = 100
epochs = 100
```

3. This dataset is more complex than MNIST and the network has to reflect that. Let's try to use a network with three hidden layers and more hidden neurons than the previous example:

```
model = Sequential([
    Dense(1024, input_dim=input_size),
    Activation('relu'),
    Dense(512),
    Activation('relu'),
    Dense(512),
    Activation('sigmoid'),
    Dense(classes),
    Activation('softmax')
])
```

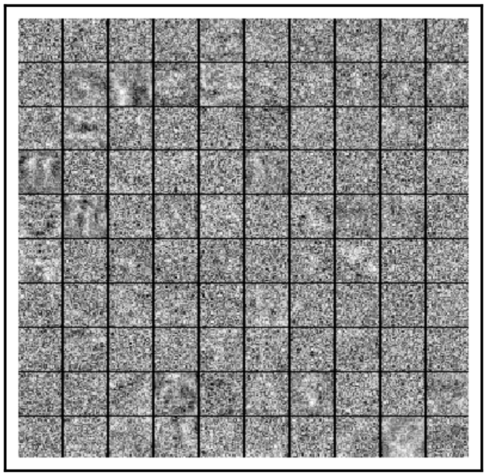
4. We'll run the training with one additional parameter, `validation_data=(X_test, Y_test)`, which will use the test data as a validation set:

```
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='sgd')
model.fit(X_train, Y_train, batch_size=batch_size, epochs=epochs,
validation_data=(X_test, Y_test), verbose=1)
```

5. Next, we'll visualize the weights of 100 random neurons from the first layer. We'll reshape the weights to 32x32 arrays and we'll compute the mean value of the 3 color channels to produce a grayscale image:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.gridspec as gridspec
import numpy
import random

fig = plt.figure()
outer_grid = gridspec.GridSpec(10, 10, wspace=0.0, hspace=0.0)

weights = model.layers[0].get_weights()

w = weights[0].T

for i, neuron in enumerate(random.sample(range(0, 1023), 100)):
 ax = plt.Subplot(fig, outer_grid[i])
 ax.imshow(numpy.mean(numpy.reshape(w[i], (32, 32, 3)), axis=2),
cmap=cm.Greys_r)
 ax.set_xticks([])
 ax.set_yticks([])
 fig.add_subplot(ax)

plt.show()
```

If everything goes as planned, we'll see the result in the following image:



Composite figure with the weights of 100 random neurons from the first layer. Unlike MNIST, there is no clear indication of what the neurons might have learned

Compared to the MNIST example, training takes much longer. But by the end, we'll have about 60% training accuracy and only about 51% test accuracy, despite the larger network. This is due to the higher complexity of the data. The accuracy of the training keeps increasing, but the validation accuracy plateaus at some point, showing that the network starts to overfit and to saturate some parameters.

Summary

In this chapter, we explained what deep learning is and how it's related to deep neural networks. We discussed the different types of networks and how to train them. We also mentioned many real-world applications of deep learning and tried to analyze the reasons for its efficiency. Finally, we introduced three of the most popular deep learning libraries, namely, TensorFlow, Keras and PyTorch. We also implemented a couple of examples with Keras, but we hit a low accuracy ceiling when we tried to classify the **CIFAR-10** dataset.

In the next chapter, we'll discuss how to improve these results with the help of convolutional networks – one of the most popular and effective deep network models. We'll talk about their structure, building blocks, and what makes them uniquely suited to computer vision tasks. To spark your interest, let's recall that convolutional networks have consistently won the popular ImageNet challenge since 2012, delivering top-five accuracy from 74.2% to 97.7%.

4
Computer Vision with Convolutional Networks

In `Chapter 2`, *Neural Networks*, and `Chapter 3`, *Deep Learning Fundamentals*, we set high expectations of deep learning and computer vision. First, we mentioned the ImageNet competition, and then we talked about some of its exciting real-world applications, such as semi-autonomous cars. In this chapter, and the next two chapters, it's time to deliver on those expectations.

Vision is arguably the most important human sense. We rely on it for almost any action we take. But image recognition has (and in some ways still is), for the longest time, been one of the most difficult problems in computer science. Historically, it's been very difficult to explain to a machine what features make up a specified object, and how to detect them. But, as we've seen, in deep learning the neural network can learn those features by itself.

In this chapter, we will cover the following topics:

- Intuition and justification for **convolutional neural networks** (**CNNs**)
- Convolutional layers
- Stride and padding in convolutional layers
- Pooling layers
- The structure of a convolutional network
- Improving the performance of CNNs
- A CNN example with Keras and CIFAR-10

Intuition and justification for CNN

The information we extract from sensory inputs is often determined by their context. With images, we can assume that nearby pixels are closely related and their collective information is more relevant when taken as a unit. Conversely, we can assume that individual pixels don't convey information related to each other. For example, to recognize letters or digits, we need to analyze the dependency of pixels close by, because they determine the shape of the element. In this way, we could figure the difference between, say, a 0 or a 1. The pixels in an image are organized in a two-dimensional grid, and if the image isn't grayscale, we'll have a third dimension for the color maps.

Alternatively, a **magnetic resonance image** (**MRI**) also uses three-dimensional space. You might recall that, until now, if we wanted to feed an image to a neural network, we had to reshape it from two-dimensional to a one-dimensional array. CNNs are built to address this issue: how to make information pertaining to neurons that are closer more relevant than information coming from neurons that are further apart. In visual problems, this translates into making neurons process information coming from pixels that are near to one another. With CNNs, we'll be able to feed one-, two-, or three-dimensional inputs and the network will produce an output of the same dimensionality. As we'll see later, this will give us several advantages.

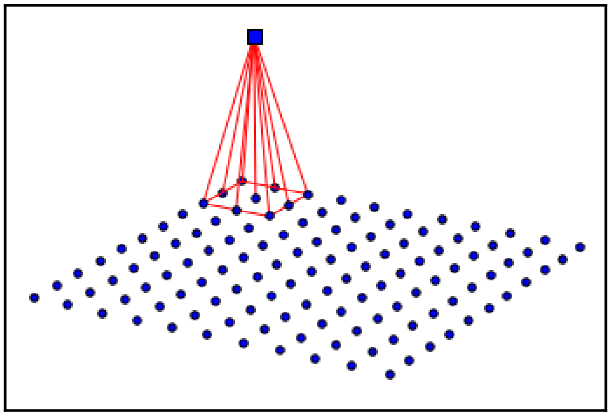
If you recall, at the end of the previous chapter, we tried to classify the CIFAR-10 images using network of fully-connected layers with little success. One of the reasons is that they overfit. Let's analyze the first hidden layer of that network, which has 1,024 neurons. The input size of the image is 32x32x3 = 3,072. Therefore, the first hidden layer had a total of 2072 * 1024 = 314, 5728 weights. That's no small number! Not only is it easy to overfit such a large network, but it's also memory inefficient. Additionally, each input neuron (or pixel) is connected to every neuron in the hidden layer. Because of this, the network cannot take advantage of the spatial proximity of the pixels, since it doesn't have a way of knowing which pixels are close to each other. By contrast, CNNs have properties that provide an effective solution to these problems:

- They connect neurons, which only correspond to neighboring pixels of the image. In this way, the neurons are "forced" to only take input from other neurons which are spatially close. This also reduces the number of weights, since not all neurons are interconnected.
- A CNN uses **parameter sharing**. In other words, a limited number of weights are shared among all neurons in a layer. This further reduces the number of weights and helps fight overfitting. It might sound confusing, but it will become clear in the next section.

In this chapter, we'll discuss CNNs in the context of computer vision and all explanations and examples will be related to that. But they are also successfully applied in areas such as speech recognition and **natural language processing** (**NLP**). Many of the explanations we'll describe here are also valid for those areas. That is, the principles of CNNs are the same regardless of the field of use.

Convolutional layers

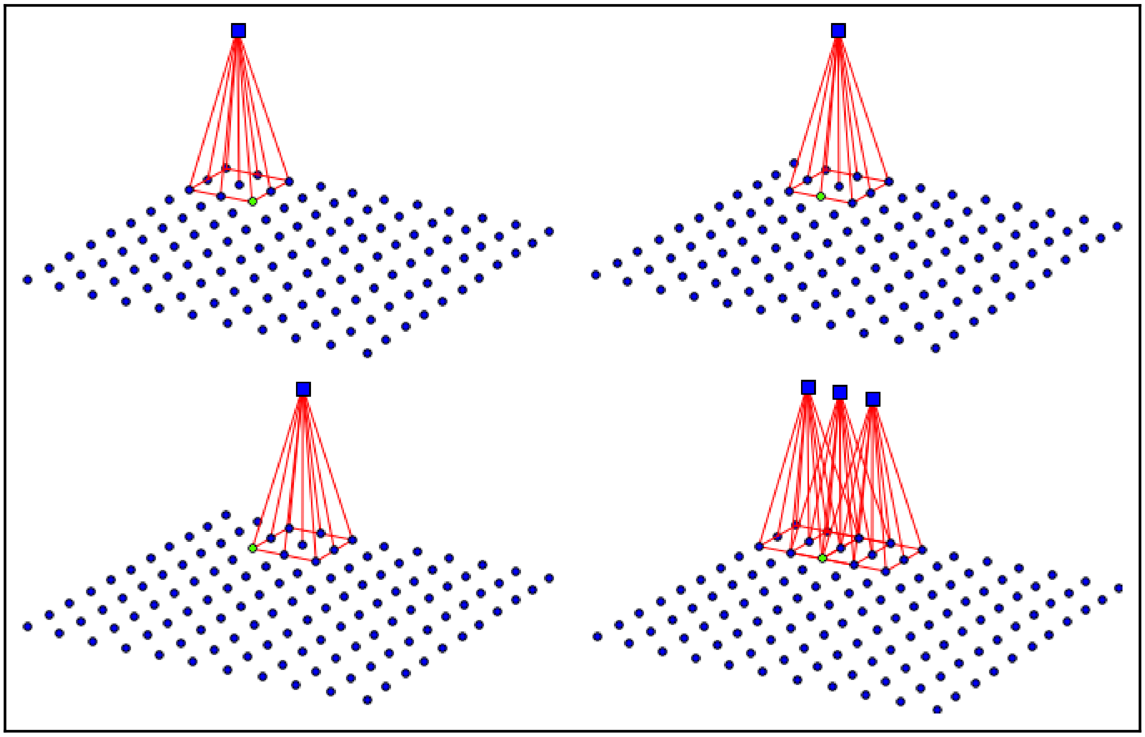
The convolutional layer is the most important building block of a CNN. It consists of a set of filters (also known as kernels or feature detectors), where each filter is applied across all areas of the input data. A filter is defined by a **set of learnable weights**. As a nod to the topic at hand, the following image illustrates this very well:



Displayed is a two-dimensional input layer of a neural network. For the sake of simplicity, we'll assume that this is the input layer, but it can be any layer of the network. As we've seen in the previous chapters, each input neuron represents the color intensity of a pixel (we'll assume it's a grayscale image for simplicity). First, we'll apply a 3x3 filter in the top-right corner of the image. Each input neuron is associated with a single weight of the filter. It has nine weights, because of the nine input neurons, but, in general, the size is arbitrary (2x2, 4x4, 5x5, and so on). The output of the filter is a **weighted sum** of its inputs (the activations of the input neurons). Its purpose is to highlight a specific feature in the input, for example, an edge or a line. The group of nearby neurons, which participate in the input are called the **receptive field**. In the context of the network, the filter output represents the activation value of a neuron in the next layer. The neuron will be active, if the feature is present at this spatial location.

In a convolutional layer, the neuron activation value is defined in the same way as the activation value of the neuron, we defined in `Chapter 2`, *Neural Networks*. But here, the neuron takes input only from a limited number of input neurons in its immediate surroundings. This is opposed to a fully-connected layer, where the input comes from all neurons.

So far, we've calculated the activation of a single neuron. What about the others? It's simple! For each new neuron, we'll slide the filter across the input image and we'll compute its output (the weighted sum) with each new set of input neurons. In the following diagram, you can see how to compute the activations of the next two positions (one pixel to the right):

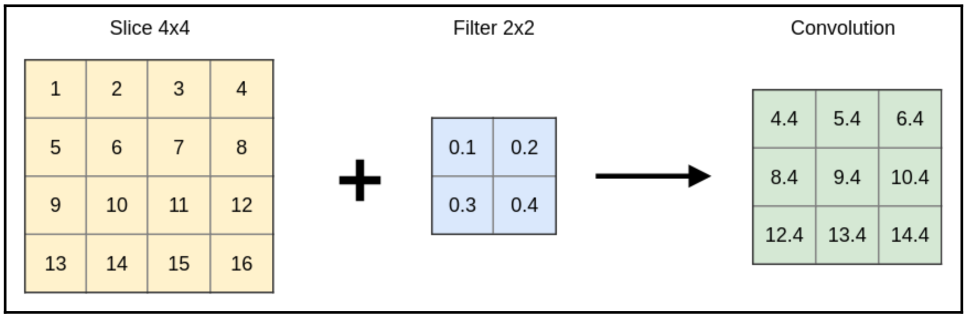


As the filter moves across the image, we compute the new activation values for the neurons in the output slice

By saying "slide", we mean that the weights of the filter don't change across the image. In effect, we'll use the same nine filter weights to compute the activations of all output neurons, each time with a different set of input neurons. We call this parameter sharing, and we do it for two reasons:

- By reducing the number of weights, we reduce the memory footprint and prevent overfitting.
- The filter highlights specific features. We can assume that this feature is useful, regardless of its position on the image. By sharing weights, we guarantee that the filter will be able to locate the feature throughout the image.

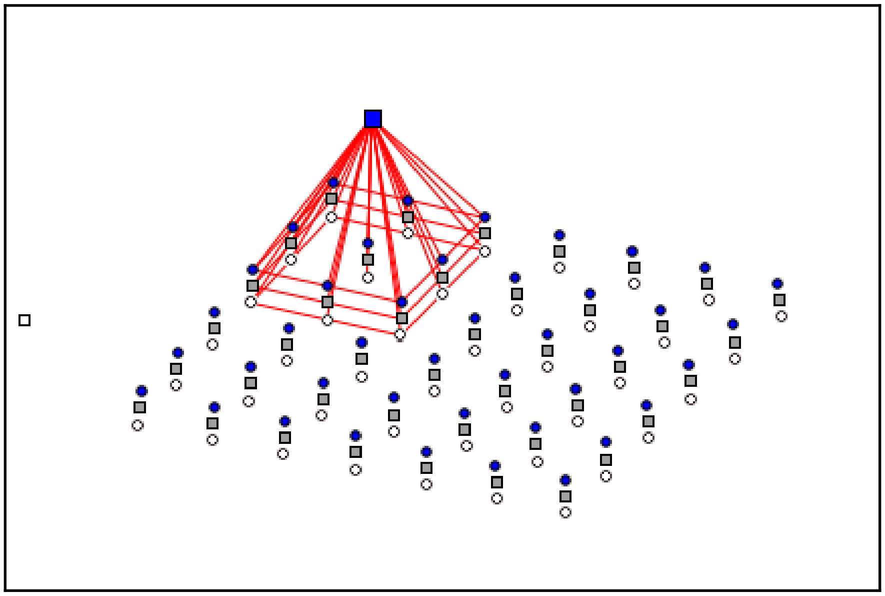
To compute all output activations, we'll repeat the process until we've moved across the whole input. The spatially arranged neurons are called **depth slices** (or a feature map), implying that there is more than one slice. The slice can serve as an input to other layers in the network. It's interesting to note that each input neuron is part of the input of multiple output neurons. For example, as we slide the filter, the green neuron in the preceding diagram will form the input of nine output neurons. Finally, just as with regular layers, we can use activation function after each neuron. As we mentioned in `Chapter 2`, *Neural Networks*, the most common activation function is the ReLU. An example of this is shown in the following illustration:



An example of convolution with a 2x2 filter over a 4x4 slice. The output is a 3x3 slice

In convolutional layers, the bias weight is also shared across all neurons. We'll have a **single** bias weight for the whole slice.

So far, we have described the one-to-one slice relation, where the output is a single slice, which takes input from another slice (or an image). This works well in grayscale, but how do we adapt it for color images (n to 1 relation)? Once again, it's simple! First, we'll split the image in color channels. In the case of RGB, that would be three. We can think of each color channel as a depth slice, where the values are the pixel intensities for the given color (R, G, or B), as shown in the following example:



An example of an input slice with depth 3

The combination of slices is called **input volume** with a **depth** of 3. A unique 3x3 filter is applied to each slice. The activation of one output neuron is just the weighted sum of the filters applied across all slices. In other words, we'll combine the three filters in one big 3 x 3 x 3 + 1 filter with 28 weights (we added depth and a single bias). Then, we'll compute the weighted sum by applying the relevant weights to each slice.

The input and output features maps have different dimensions. Let's say we have an input layer with size `(width, height)` and a filter with dimensions `(filter_w, filter_h)`. After applying the convolution, the dimensions of the output layer are `(width - filter_w + 1, height - filter_h + 1)`.

As we mentioned, a filter highlights a specific feature, such as edges or lines. But, in general, many features are important and we'll be interested in all of them. How do we highlight them all? As usual, it's simple (if Grumpy Cat read this book, she would surely be tired of so much optimism). We'll apply multiple filters across the set of input slices. Each filter will generate a unique output slice, which highlights the feature, detected by the filter (n to m relation). An output slice can receive input from:

- All input slices, which is the standard for convolutional layers. In this scenario, a single output slice is a case of the n-to-1 relationship, we described before. With multiple output slices, the relation becomes n-to-m. In other words, each input slice contributes to the output of each output slice.
- A single input slice. This operation is known as **depthwise convolution**. It's a kind of reversal of the previous case. In its most simple form, we apply a filter over a single input slice to produce a single output slice. This is a case of the one-to-one relation, we described in the previous section. But we can also specify a **channel multiplier** (an integer *m*), where we apply *m* filters over a single output slice to produce *m* output slices. This is a case of 1-to-m relation. The total number of output slices is *n * m*.

Let's denote the width and height of the filter with F_w and F_h, the depth of the input volume with *D*, and the depth of the output volume with *M*. Then, we can compute the total number of weights *W* in a convolutional layer with the following equation:

$$W = (D * F_w * F_h + 1) * M$$

Let's say we have three slices and want to apply four 5x5 filters to them. Then, the convolutional layer will have a total of *(3x5x5 + 1) * 4 = 304* weights, and four output slices (output volume with depth of 4), one bias per slice. The filter for each output slice will have three 5x5 filter patches for each of the three input slices and one bias for a total of 3x5x5 + 1 = 76 weights. The combination of the output maps is called **output volume** with a **depth** of four.

We can think of the fully-connected layer as a special case of convolutional layer, with input volume of depth 1, filters with the same size as the size of the input, and a total number of filters, equal to the number of output neurons.

A coding example of convolution operation

We've now described how convolutional layers work, but we'll gain better intuition with a visual example. Let's implement a convolution operation by applying a couple of filters across an image. For the sake of clarity, we'll implement the sliding of the filters across the image manually and we won't use any DL libraries. Let's start.

1. First, we'll import `numpy`, as shown in the following example:

   ```
   import numpy as np
   ```

2. Then, we'll define the function `conv`, which applies the convolution across the image. `conv` takes two parameters: `image` for the image itself and `filter`, for the filter:
    - First, we'll compute the output image size, which depends on the input image and filter sizes. We'll use it to instantiate the output image `im_c`.
    - Then, we'll iterate over all pixels of the image, applying the filter at each location.
    - We'll check if any value is out of the `[0, 255]` interval and fix it, if necessary.
    - Finally, we'll display the input and output images for comparison.

This is shown in the following example:

```
def conv(image, im_filter):
    """
    :param image: grayscale image as a 2-dimensional numpy array
    :param im_filter: 2-dimensional numpy array
    """

    # input dimensions
    height = image.shape[0]
    width = image.shape[1]

    # output image with reduced dimensions
    im_c = np.zeros((height - len(im_filter) + 1,
                     width - len(im_filter) + 1))

    # iterate over all rows and columns
    for row in range(len(im_c)):
        for col in range(len(im_c[0])):
            # apply the filter
            for i in range(len(im_filter)):
                for j in range(len(im_filter[0])):
                    im_c[row, col] += image[row + i, col + j] *
```

```
im_filter[i][j]

    # fix out-of-bounds values
    im_c[im_c > 255] = 255
    im_c[im_c < 0] = 0

    # plot images for comparison
    import matplotlib.pyplot as plt
    import matplotlib.cm as cm

    plt.figure()
    plt.imshow(image, cmap=cm.Greys_r)
    plt.show()

    plt.imshow(im_c, cmap=cm.Greys_r)
    plt.show()
```

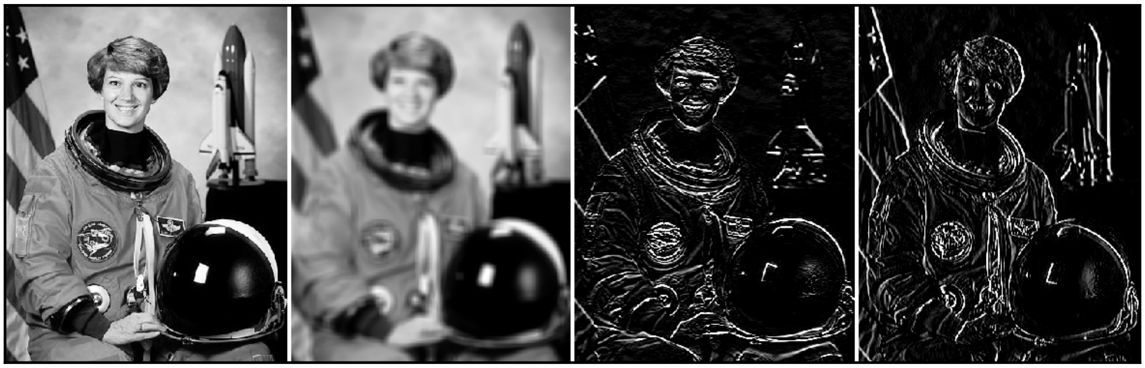
3. Next, we'll download the image. The following boilerplate code is in the global scope of the module. It loads the requested `RGB` image into a `numpy` array and converts it to grayscale, as shown:

```
import requests
from PIL import Image
from io import BytesIO

# load the image
url =
"https://upload.wikimedia.org/wikipedia/commons/thumb/8/88/Commande
r_Eileen_Collins_-_GPN-2000-001177.jpg/382px-
Commander_Eileen_Collins_-_GPN-2000-001177.jpg?download"
resp = requests.get(url)
image_rgb =
np.asarray(Image.open(BytesIO(resp.content)).convert("RGB"))

# convert to grayscale
image_grayscale = np.mean(image_rgb, axis=2, dtype=np.uint)
```

4. Finally, we'll apply different filters across the image. To better illustrate our point, we'll use 10 x 10 `blur` filter, as well as Sobel edge detectors, as shown in the following example:

```
# blur filter
blur = np.full([10, 10], 1. / 100)
conv(image_grayscale, blur)

# Sobel edge detectors
sobel_x = [[-1, -2, -1],
             [0, 0, 0],
             [1, 2, 1]]
conv(image_grayscale, sobel_x)

sobel_y = [[-1, 0, 1],
             [-2, 0, 2],
             [-1, 0, 1]]
conv(image_grayscale, sobel_y)
```

This code generates the following images:

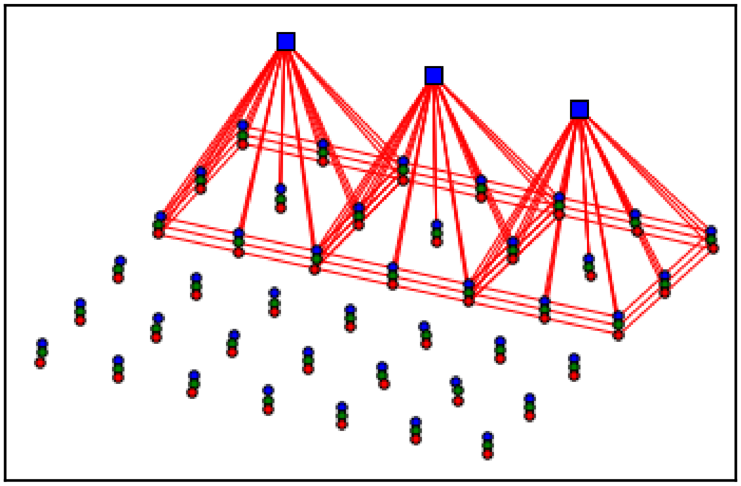


The first image is the grayscale input. The second image is the result of a 10 x 10 blur filter. The third and fourth images use detectors and vertical Sobel edge detector

In this example, we used filters with hard-coded weights to visualize how the convolution operation works in neural networks. In reality, the weights of the filter will be set during the network training. All we'll need to do is define the network architecture, such as the number of convolutional layers, depth of the output volume, and the size of the filters. The network will figure out the features, highlighted by each filter during training.

Stride and padding in convolutional layers

Until now, we assumed that sliding of the filter happens one pixel at a time, but that's not always the case. We can slide the filter multiple positions. This parameter of the convolutional layers is called **stride**. Usually, the stride is the same across all dimensions of the input. In the following diagram, we can see a convolutional layer with a stride of 2:



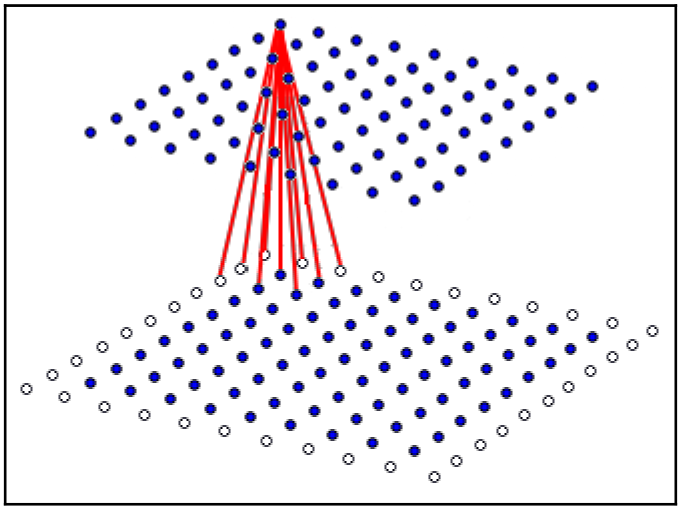
With stride 2, the filter is translated by two pixels at a time

By using a stride larger than 1, we reduce the size of the output slice. In the previous section, we introduced a simple formula for the output size, which included the sizes of the input and the kernel. Now, we'll extend it to also include the stride: `((width - filter_w) / stride_w + 1, ((height - filter_h) / stride_h + 1)`. For example, the output size of a square slice generated by a 28x28 input image, convolved with a 3x3 filter with stride 1, would be 28 - 3 + 1 = 26. But with stride 2, we get (28 - 3) / 2 + 1 = 13.

The main effect of the larger stride is an increase in the receptive field of the output neurons. Let's explain this with an example. If we use stride 2, the size of the output slice will be roughly four times smaller than the input. In other words, one output neuron will "cover" area, which is four times larger, compared to the input neurons. The neurons in the following layers will gradually capture input from larger regions from the input image. This is important, because it would allow them to detect larger and more complex features of the input.

A convolution operation with stride larger than 1 is usually called **stride convolution**.

The convolution operations we have discussed until now have produced smaller output than the input. But, in practice, it's often desirable to control the size of the output. We can solve this by **padding** the edges of the input slice with rows and columns of zeros before the convolution operation. The most common way to use padding is to produce output with the same dimensions as the input. In the following diagram, we can see a convolutional layer with padding of 1:



Convolutional layer with padding 1

The white neurons represent the padding. The input and the output slices have the same dimensions (dark neurons). This is the most common way to use padding. The newly padded zeros will participate in the convolution operation with the slice, but they won't affect the result. The reason is that, even though the padded areas are connected with weights to the following layer, we'll always multiply those weights by the padded value, which is 0.

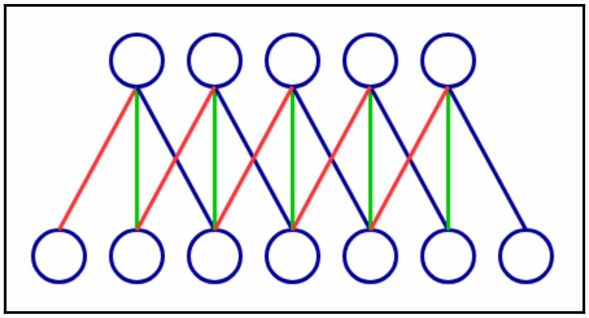
We'll now add padding to the formula of the output size. Let the size of the input slice be $I=(I_w, I_h)$, the size of the filter $F=(F_w,F_h)$, the stride $S=(S_w,S_h)$, and the padding $P=(P_w,P_h)$. Then the size $O=(O_w,O_h)$ of the output slice is given by the following equations:

$$O_w = \frac{I_w + 2P_w - F_w}{S_w} + 1$$

$$O_h = \frac{I_h + 2P_h - F_h}{S_h} + 1$$

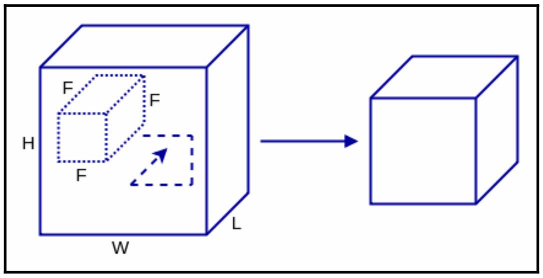
1D, 2D, and 3D convolutions

Until now, we've used 2D convolutions, where the input and output neurons were arranged in a two-dimensional grid. This works very well for images. But we can also have 1D and 3D convolutions, where the neurons are arrange in one-dimensional or three-dimensional space respectively. In all cases, the filter has the same number of dimensions as the input and the weights are shared across the input. For example, we would use 1D convolution with time-series data, because the values are arranged across a single time axis. In the following diagram, we can see an example of 1D convolution:



1D convolution

The weights with the same color (red, green, or blue) share the same value. The output of 1D convolution is also 1D. If the input is 3D, such as a 3D MRI, we could use 3D convolution, which will also produce 3D output. In this way, we'll maintain the spatial arrangement of the input data. We can see an example of 3D convolution in the following diagram:



3D convolution

The input has dimensions H/W/L and the filter has a single size F for all dimensions. The output is also 3D.

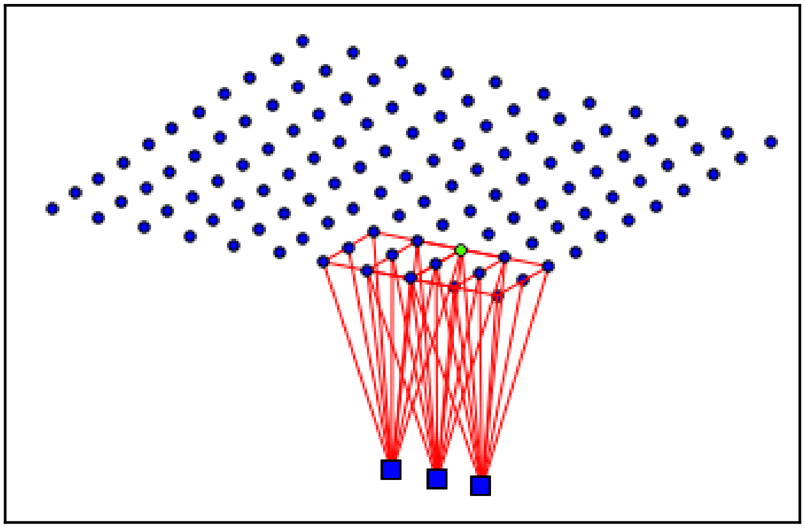
In the previous sections, we used 2D convolutions to work with RGB images. But we might consider the three colors as an additional dimension, making the RGB image 3D. Why didn't we use 3D convolutions then? The reason is that, even though we can think of the input as 3D, the output is still a two-dimensional grid. Had we used 3D convolution, the output would also be 3D, which doesn't carry any meaning in the case of images.

1x1 convolutions

1x1 (pointwise) convolution is a special case of convolution, where each dimension of the convolution filter is of size 1 (1x1 in 2D convolutions and 1x1x1 in 3D). At first this doesn't make sense—a 1x1 filter doesn't increase the receptive field size of the output neurons. The result of such convolution would be just pointwise scaling. But it can be useful in another way - we can use them to change the depth between the input and output volumes. To understand this, let's recall that in the general case we have input volume with a depth of *D* slices and *M* filters for *M* output slices. Each output slice is generated by applying a unique filter over all input slices. If we use a 1x1 filter and *D != M*, we'll have output slices of the same size, but with different volume depth. At the same time, we won't change the receptive field size between input and output. The most common use case is to reduce the output volume, or *D* > *M* (dimension reduction), nicknamed the "bottleneck" layer.

Backpropagation in convolutional layers

In `Chapter 2`, *Neural Networks*, we talked about backpropagation in general, and for fully-connected layers in particular. In a fully-connected layer, an input neuron contributes to all output neurons. Because of this, when the gradient is routed back, all output neurons contribute back to the original neuron. In effect, we used the same operation of weighted sum in the forward and backward passes. The same rule applies for convolutional layers, where the neurons are locally-connected. In the *Convolutional layers* section, we observed how a neuron participates in the inputs of several output neurons. This is illustrated in the following diagram, where we can see a convolution operation with 3x3 filter. The green neuron will participate in the inputs of 9 output neurons, arranged in a 3x3 pattern. Conversely, the same neurons will route the gradient in the backward pass. This example shows that the backward pass of a convolution operation is another convolution operation with the same parameters, but with spatially-flipped filter. This operation is known as transposed convolution (or deconvolution). As we'll see later in the book, it has other applications besides backpropagation:



The backward pass of a convolution operation is also a convolution

As we mentioned in the previous chapter, all modern deep learning libraries have automatic differentiation. This is true for all the layers, we'll talk about in this chapter. You'll probably never have to implement the derivatives of a convolution operation, except as an exercise.

Convolutional layers in deep learning libraries

PyTorch, Keras, and TensorFlow have out of the gate support for 1D, 2D, and 3D standard, and depthwise convolutions. The inputs and outputs of the convolution operation are tensors. A 1D convolution would have 3D input and output tensors. Their axes can be in either NCW or NWC order, where:

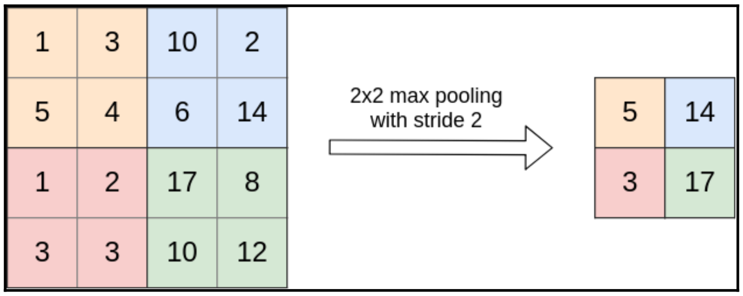
- N stands for the index of the sample in the mini-batch
- C stands for the index of the depth slice in the volume
- W stands for the content of the slice

In the same way, a 2D convolution will be represented by NCHW or NHWC ordered tensors, where H and W are the height and width of the slices. A 3D convolution will have NCDHW or NDHWC order, where D stands for the depth of the slice.

Pooling layers

In the previous section, we explained how to increase the receptive field of the neurons by using a stride larger than 1. But we can also do this with the help of pooling layers. A pooling layer splits the input slice into a grid, where each grid cell represents a receptive field of a number of neurons (just as a convolutional layer does). Then, a pooling operation is applied over each cell of the grid. Different types of pooling layers exist. Pooling layers don't change the volume depth, because the pooling operation is performed independently on each slice.

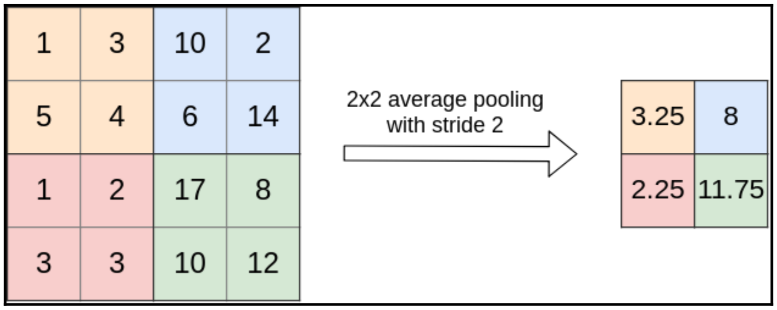
Max pooling: is the most popular way of pooling. The max pooling operation takes the neuron with the highest activation value in each local receptive field (grid cell), and propagates only that value forward. In the following figure, we can see an example of max pooling with a receptive field of 2x2:



An example of the input and output of a max pooling operation with stride 2 and 2x2 receptive field. This operation discards 3/4 of the input neurons

Pooling layers don't have any weights. In the backward pass of max pooling, the gradient is routed only to the neuron, which had the highest activation during the forward pass. The other neurons in the receptive field propagate zeros.

Average pooling: is another type of pooling, where the output of each receptive field is the mean value of all activations within the field. The following is an example of average pooling:



An example of the input and output of a max pooling operation with stride 2 and 2x2 receptive field

Pooling layers are defined by two parameters:

- Stride, which is the same as with convolutional layers
- Receptive field size, which is the equivalent of the filter size in convolutional layers

In practice, only two combinations are used. The first is a 2x2 receptive field with stride 2, and the second is a 3x3 receptive field with stride 2 (overlapping). If we use a larger value for either parameters, the network loses too much information. Alternatively, if the stride is 1, the size of the layer wouldn't be smaller, neither will the receptive field increase.

Based on these parameters, we can compute the output size of a pooling layer. Let's denote the size of the input slice with *I*, the size of the receptive field with *F*, the size of the stride with *S*, and the size of the output with *O*. Pooling layers typically don't have padding. Then the formulas are as follows:

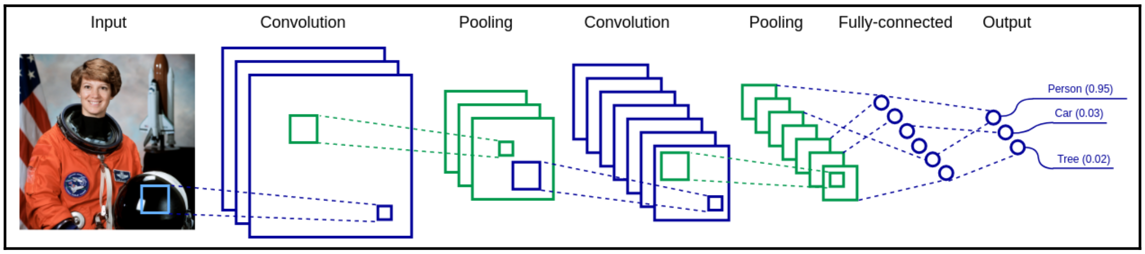
$$O_w = \frac{I_w - F_w}{S_w} + 1$$

$$O_h = \frac{I_h - F_h}{S_h} + 1$$

Pooling layers are still very much used, but sometimes we can achieve similar or better results by simply using convolutional layers with larger strides. (See, for example, J. Springerberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, *Striving for Simplicity: The All Convolutional Net*, (2015), `https://arxiv.org/abs/1412.6806`).

The structure of a convolutional network

Before going further, let's put together everything we have learned so far. In the figure following we can see the structure of a basic CNN:



A basic convolutional network with convolutional and fully-connected layers in blue and pooling layers in green

Most CNNs share basic properties. Here are some of them:

- We would typically alternate one or more convolutional layers with one pooling layer. In this way, the convolutional layers can detect features at every level of the receptive field size. The aggregated receptive field size of deeper layers is larger than the ones at the beginning of the network. This allows them to capture more complex features from larger input regions. Let's illustrate this with an example. Imagine that the network uses 3x3 convolutions with stride 1 and 2x2 pooling with stride 2:
    - The neurons of the first convolutional layer will receive input from 3x3 pixels of the image.
    - A group of 2x2 output neurons of the first layer will have a combined receptive field size of 4x4 (because of the stride).
    - After the first pooling operation, this group will be combined in a single neuron of the pooling layer.
    - The second convolution operation takes input from 3x3 pooling neurons. Therefore, it will receive input from a square with side 3x4 = 12 (or a total of 12x12 = 144) pixels from the input image.

- We use the convolutional layers to extract features from the input. The features detected by the deepest layers are highly abstract, but they are also not readable by humans. To solve this problem, we usually add one or more fully-connected layers after the last convolutional/pooling layer. In this example, the last fully-connected layer (output) will use softmax to estimate the class probabilities of the input. You can think of the fully-connected layers as translators between the network's language (which we don't understand) and ours.
- The deeper convolutional layers usually have more filters (hence higher volume depth), compared to the initial ones. A feature detector in the beginning of the network works on a small receptive field. It can only detect a limited number of features, such as edges or lines, shared among all classes. On the other hand, a deeper layer would detect more complex and numerous features. For example, if we have multiple classes such as cars, trees, or people, each would have its own set of features such as tires, doors, leaves and faces, and so on. This would require more feature detectors.

Classifying handwritten digits with a convolutional network

In the third chapter, we introduced a simple neural network to classify digits using Keras and we got 96% accuracy. We could improve this with some tricks (more hidden neurons, for example), but let's try with a simple CNN instead:

1. First, we'll do the imports. We'll also set the random seed:

```
# for reproducibility
from numpy.random import seed

seed(1)
from tensorflow import set_random_seed

set_random_seed(1)
```

2. Then, we'll do the imports, including convolutional and max pooling layers, as shown in the following example:

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Flatten

from keras.utils import np_utils
```

3. Next, we'll import the MNIST dataset. We already did a similar step in the previous chapter in the section *Using Keras to classify handwritten digits*. Since we'll be using convolutional layers, we can reshape the input in 28x28 patches, as shown in the following example:

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)

Y_train = np_utils.to_categorical(Y_train, 10)
Y_test = np_utils.to_categorical(Y_test, 10)
```

4. Then, we'll define the model - a network with two convolutional layers, one max pooling layer, and two fully-connected layers. Besides this, we need to use a `Flatten` between the max pooling and the fully-connected layer. We have to do this, because the fully-connected layer expects one-dimensional input, but the output of the convolutional layer is three-dimensional. This is shown in the following example:

```
model = Sequential([
    Convolution2D(filters=32,
                  kernel_size=(3, 3),
                  input_shape=(28, 28, 1)),  # first conv layer
    Activation('relu'),
    Convolution2D(filters=32,
                  kernel_size=(3, 3)),  # second conv layer
    Activation('relu'),
    MaxPooling2D(pool_size=(2, 2)),  # max pooling layer
    Flatten(),  # flatten the output tensor
    Dense(64),  # fully-connected hidden layer
    Activation('relu'),
    Dense(10),  # output layer
    Activation('softmax')])

print(model.summary())
```

5. We can use the `model.summary()` method of Keras to better explain the network architecture. The output is shown in the following example:

```
Layer (type)                     Output Shape         Param #
=================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)   320
_________________________________________________________________
activation_1 (Activation)        (None, 26, 26, 32)   0
_________________________________________________________________
conv2d_2 (Conv2D)                (None, 24, 24, 32)   9248
_________________________________________________________________
activation_2 (Activation)        (None, 24, 24, 32)   0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D)   (None, 12, 12, 32)   0
_________________________________________________________________
flatten_1 (Flatten)              (None, 4608)         0
_________________________________________________________________
dense_1 (Dense)                  (None, 64)           294976
_________________________________________________________________
activation_3 (Activation)        (None, 64)           0
_________________________________________________________________
dense_2 (Dense)                  (None, 10)           650
_________________________________________________________________
activation_4 (Activation)        (None, 10)           0
=================================================================
Total params: 305,194
Trainable params: 305,194
Non-trainable params: 0
```

6. Next, we'll define the `optimizer`. Instead of a **stochastic gradient descent** (**SGD**), we'll use ADADELTA. It automatically makes the learning rate larger or smaller in an inversely proportional way to the gradient. In this way, the network doesn't learn too slowly and it doesn't skip minima by taking too large a step. By using ADADELTA, we dynamically adjust the parameters with time (see also: Matthew D. Zeiler, *ADADELTA: An Adaptive Learning Rate Method,* arXiv:1212.5701v1 (`https://arxiv.org/abs/1212.5701`). This is shown in the following example:

```
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='adadelta')
```

7. Then, we'll train the network for `5 epochs`, as shown in the following example:

```
model.fit(X_train, Y_train, batch_size=100, epochs=5,
validation_split=0.1, verbose=1)
```

8. Finally, we'll test, as shown in the following code:

```
score = model.evaluate(X_test, Y_test, verbose=1)
print('Test accuracy:', score[1])
```

The accuracy of this model is 98.5%.

Improving the performance of CNNs

We now know the basics of CNNs. With this foundation, we'll discuss different techniques to improve their performance.

Data pre-processing

Until now, we've fed the network with unmodified inputs. In the case of images, these are pixel intensities in the range [0:255]. But that's not optimal. Imagine that we have an RGB image, where the intensities in one of the color channels is very high compared to the other two. When we feed the image to the network, the values of this channel will become dominant, diminishing the others. This could skew the results, because in reality every channel has equal importance. To solve this, we need to prepare (or **normalize**) the data, before we feed it to the network. In practice, we'll use two types of normalization:

- **Feature scaling:** where $x = \frac{x - x_{min}}{x_{max} - x_{min}}$. This operation scales all inputs in the [0, 1] range. For example, a pixel with intensity 125, would have a scaled value of $\frac{125-0}{250-0} = 0.5$. Feature scaling is fast and easy to implement.
- **Standard score:** where $x = \frac{x - \mu}{\sigma}$. Here μ and σ are the mean and standard deviation of all training data. They are usually computed separately for each input dimension. For example, in an RGB image, we would compute mean μ and σ for each channel. We should note that μ and σ have to be computed only on the training data and then applied to the test data.

Regularization

We already know that overfitting is a central problem in machine learning (and even more so in deep networks). In this section, we'll discuss several ways to prevent it. Such techniques are collectively known as **regularization**. To quote Ian Goodfellow's Deep Learning book:

> *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

Weight decay

The first technique we are going to discuss is weight decay (also known as L2 regularization). It works by adding additional terms to the value of the loss function. Without going into too much detail, we'll say that this term is a function of all the weights of the network. This means that, if the network weights have large values, the loss function increases. In effect, weight decay penalizes large network weights (hence the name). This prevents the network from relying too heavily on a few features associated with these weights. There is less chance of overfitting, when the network is forced to work with multiple features. In practical terms, we can add weight decay by changing the weight update rule, we introduced in `Chapter 2`, *Neural networks*, as shown in the following equations:

$$w \to w - \eta\nabla(J(w))$$

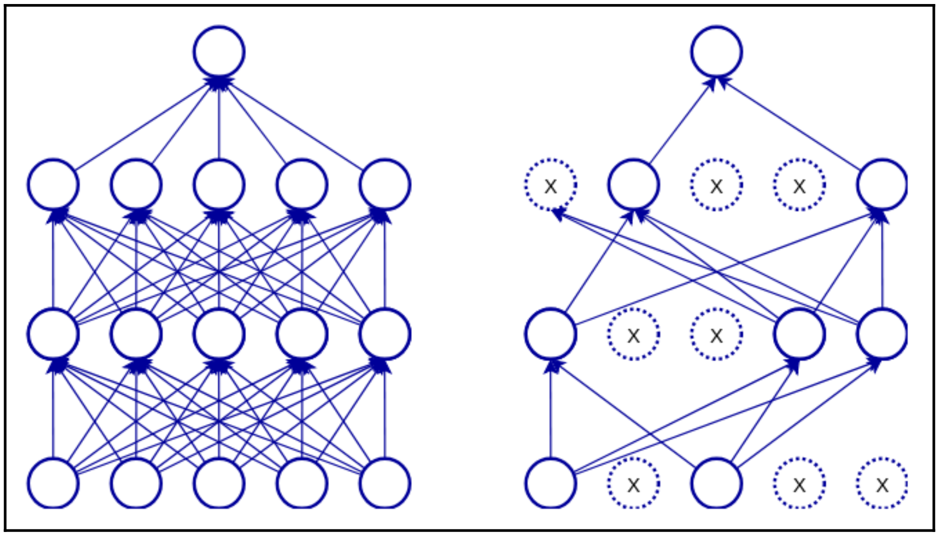
becomes

$$w \to w - \eta(\nabla(J(w)) - \lambda w)$$

where λ is the weight decay coefficient.

Dropout

Dropout is a regularization technique, which can be applied to the output of some of the network layers. Dropout randomly and periodically removes some of the neurons (along with their input and output connections) from the network. During a training mini-batch, each neuron has a probability *p* to be stochastically dropped. This is to ensure that no neuron ends up relying too much on other neurons and "learns" something useful for the network instead. Dropout can be applied after convolutional, pooling, or fully-connected layers. In the following illustration, we can see a dropout for fully-connected layers:



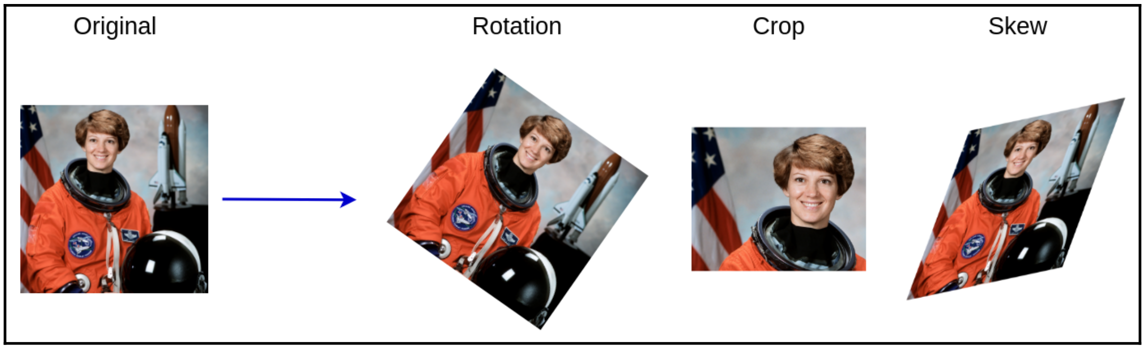
An example of dropout on fully-connected layers

Data augmentation

One of the most efficient regularization techniques is data augmentation. If the training data is too small, the network might start to overfit. Data augmentation helps counter this by artificially increasing the size of the training set. Let's use an example. In the MNIST and CIFAR-10 examples, we've trained the network over multiple epochs. The network will "see" every sample of the dataset once per epoch. To prevent this, we can apply random augmentations to the images, before using them for training. The labels will stay the same. Some of the most popular image augmentations are:

- **Rotation**
- Horizontal and vertical flip
- Zoom in/out
- **Crop**
- **Skew**
- Contrast and brightness adjustment

The emboldened augmentations are shown in the following example:



Examples of different image augmentations

Batch normalization

In *Data pre-processing*, we explained why data normalization is important. Batch normalization provides a way to apply data processing, similar to the standard score, for the hidden layers of the network. It normalizes the outputs of the hidden layer for each mini-batch (hence the name) in a way, which maintains its mean activation value close to 0, and its standard deviation close to 1. We can use it with both convolutional and fully-connected layers. Networks with batch normalization train faster and can use higher learning rates. For more information about batch normalization, see the original paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by Sergey Ioffe and Christian Szegedy, which can be seen at the following link: `Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.`

A CNN example with Keras and CIFAR-10

In `Chapter 3`, *Deep Learning Fundamentals*, we tried to classify the CIFAR-10 images with a fully-connected network, but we only managed 51% test accuracy. Let's see if we can do better with all the new things we've learned. This time we'll use CNN with data augmentation.

1. We'll start with the imports. We'll use all the layers we introduced in this chapter, as shown in the following example:

   ```
   import keras
   from keras.datasets import cifar10
   from keras.layers import Conv2D, MaxPooling2D
   from keras.layers import Dense, Dropout, Activation, Flatten,
   BatchNormalization
   from keras.models import Sequential
   from keras.preprocessing.image import ImageDataGenerator
   ```

2. We'll define the mini `batch_size` for convenience, as shown in the following code:

   ```
   batch_size = 50
   ```

3. Next, we'll import the CIFAR-10 dataset and we'll normalize the data by dividing it to 255 (maximum pixel intensity), as shown in the following code:

   ```
   (X_train, Y_train), (X_test, Y_test) = cifar10.load_data()

   X_train = X_train.astype('float32')
   X_test = X_test.astype('float32')
   X_train /= 255
   X_test /= 255

   Y_train = keras.utils.to_categorical(Y_train, 10)
   Y_test = keras.utils.to_categorical(Y_test, 10)
   ```

4. Then, we'll implement loading of the data and we'll define the augmentation types we'll use:
    - To do this, we need the `ImageDataGenerator` class
    - We'll allow rotation of up to 90 degrees, horizontal flip, horizontal and vertical shift of the data

- We'll standardize the training data (`featurewise_center` and `featurewise_std_normalization`). Because the mean and standard deviation are computed over the whole data set, we need to call the `data_generator.fit(X_train)` method
- Finally, we need to apply the training standardization over the test set. `ImageDataGenerator` will generate a stream of augmented images during training

We can see the implementation in the following code block:

```
data_generator = ImageDataGenerator(rotation_range=90,
 width_shift_range=0.1,
 height_shift_range=0.1,
 featurewise_center=True,
 featurewise_std_normalization=True,
 horizontal_flip=True)

data_generator.fit(X_train)

# standardize the test set
for i in range(len(X_test)):
    X_test[i] = data_generator.standardize(X_test[i])
```

5. Next, we'll define the network:
    - It will have three blocks of two convolutional layers (3x3 filters) and one max pooling layer
    - Perform batch normalization after each convolutional layer
    - We will define **Exponential Linear Unit** (**ELU**) activation functions
    - A single fully-connected layer after the last max pooling. Please note the `padding='same'` parameter. This simply means that the output volume slices will have the same dimensions as the input ones.

The following code demonstrates the model:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Conv2D(128, (3, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(128, (3, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

6. Next, we'll define the `optimizer`, in this case, Adam, as shown in the following code:

```
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

7. Then, we'll train the network. Because of the data augmentation, we'll now use the `model.fit_generator` method. Our generator is the `ImageDataGenerator`, we defined earlier. We'll use the test set as `validation_data`. In this way we'll know our actual performance after each epoch.

TIP

In this example, we'll train the network for 100 epochs. However, if your PC is older (or doesn't have a dedicated GPU), we would advise you to run the training for 3 or 5 epochs. The results won't be as good, but you'll still see improvement in the accuracy.

The following code demonstrates the model:

```
model.fit_generator(
 generator=data_generator.flow(x=X_train,
 y=Y_train,
 batch_size=batch_size),
 steps_per_epoch=len(X_train) // batch_size,
 epochs=100,
 validation_data=(X_test, Y_test),
 workers=4)
```

Depending on the number of epochs, this model will produce the following results:

- 47% accuracy in `3 epochs`
- 59% accuracy in `5 epochs`
- 80% accuracy in about `100 epochs`—significantly better than before, but still not perfect

Summary

In this chapter, we introduced convolutional neural networks. We talked about their main building blocks – convolutional and pooling layers – and we discussed their architecture and features. We discussed data pre-processing and various regularization techniques such as weight decay, dropout, and data augmentation. We also demonstrated how to use CNNs to classify MNIST and CIFAR-10.

In the next chapter, we'll build upon our new-found computer vision knowledge with some exciting additions. We'll discuss how to train networks faster by transferring knowledge from one problem to another, as well as the best performing advanced CNN architectures. We'll also go beyond simple classification with object detection, or how to find the object's location on the image. And for dessert, we'll talk about a fun CNN application called neural style transfer.

5
Advanced Computer Vision

In the `Chapter 4`, *Computer Vision with Convolutional Networks*, we introduced convolutional networks for computer vision. In this chapter, we'll continue with more of the same, but at a more advanced level. Our modus operandi so far has been to provide simple examples as a support to the theoretical knowledge of neural networks. We can now elevate our knowledge to the point where we'll be able to successfully solve real-world computer vision tasks with **Convolutional Neural Networks** (**CNNs**).

This chapter will cover the following topics:

- Transfer learning
- Advanced network architectures
- Capsule networks
- Object detection
- Semantic segmentation
- Artistic style transfer

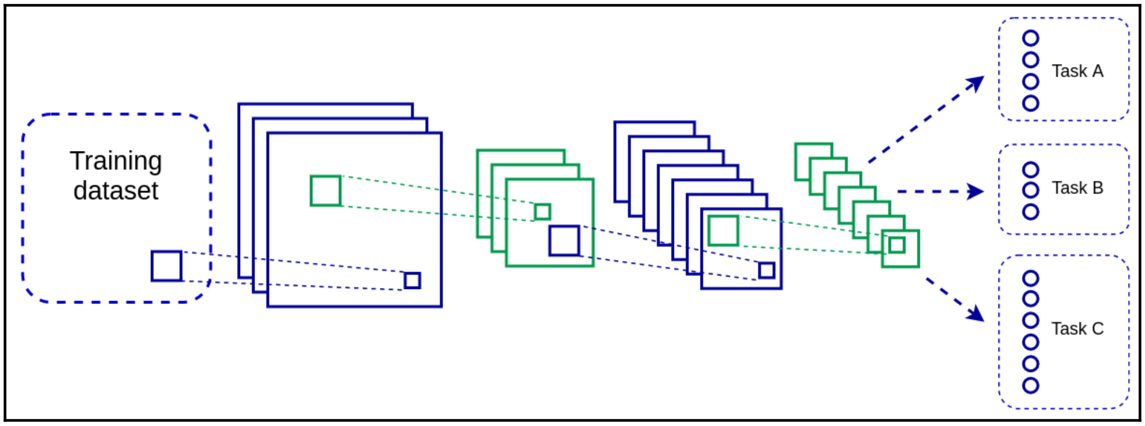
Transfer learning

So far, we've trained small models on toy datasets, where the training took no more than an hour. But if we want to work with large datasets, such as ImageNet, we would need a much bigger network that trains for a lot longer. More importantly, large datasets are not always available for the tasks we're interested in. Keep in mind that besides obtaining the images, they have to be labeled, and this could be expensive and time-consuming. So, what does a humble engineer do when they want to solve a real ML problem with limited resources? Enter transfer learning.

Transfer learning is the process of applying an existing trained ML model to a new, but related, problem. For example, we can take a network trained on ImageNet and repurpose it to classify grocery store items. Alternatively, we could use a driving simulator game to train a neural network to drive a simulated car, and then use the network to drive a real car (but don't try this at home!). Transfer learning is a general ML concept, applicable to all ML algorithms, but in this context we'll talk about CNNs. Here's how it works.

We start with an existing pre-trained net. The most common scenario is to take a net pre-trained with ImageNet, but it could be any dataset. TensorFlow/Keras/PyTorch all have popular ImageNet pre-trained neural architectures that we can use. Alternatively, we can train our own network with a dataset of our choice.

In `Chapter 4`, *Computer Vision with Convolutional Networks*, we mentioned how the fully-connected layers at the end of a CNN act as translators between the network's language (the abstract feature representations learned during training) and our language, which is the class of each sample. You can think of transfer learning as a translation to another language. We start with the network's features, which is the output of the last convolutional or pooling layer. Then, we translate them to a different set of classes of the new task. We can do this by removing the last fully-connected layer (or all fully-connected layers) of an existing pre-trained network and replacing it with another layer, which represents the classes of the new problem. Here is a diagram of the transfer learning scenario:



In transfer learning, we can replace the last layer of a pre-trained net and repurpose it for a new problem

But we cannot do this mechanically and expect the new network to work, because we still have to train the new layer with data related to the new task. Here, we have two options:

- **Use the original part of the network as feature extractor and only train the new layer(s)**: In this scenario, we feed the network a training batch of the new data and propagate it forward to see the network output. This part works just such as regular training would. But in the backward pass, we lock the weights of the original net and only update the weights of the new layers. This is the recommended way, when we have limited training data on the new problem. By locking most of the network weights, we prevent overfitting on the new data.
- **Fine-tuning the whole network**: In this scenario, we'll train the whole network, and not just the newly added layers at the end. It is possible to update all network weights, but we can also lock some of the weights in the first layers. The idea here is that the initial layers detect general features – not related to a specific task – and it makes sense to reuse them. On the other hand, the deeper layers might detect task-specific features and it would be better to update them. We can use this method when we have more training data and don't need to worry about overfitting.

Transfer learning example with PyTorch

Now that we know what transfer learning is, let's see whether it works in practice. In this section, we'll apply an advanced ImageNet pre-trained network on the CIFAR-10 images. We'll use both types of transfer learning. It's preferable to run this example on GPU:

1. Do the following imports:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import models, transforms
```

2. Define `batch_size` for convenience:

```
batch_size = 50
```

3. Define the training dataset. We have to consider a few things:
    - The CIFAR-10 images are 32 x 32, while the ImageNet network expects 224 x 224 input. As we are using ImageNet based network, we'll upsample the 32x32 CIFAR images to 224x224.
    - Standardize the CIFAR-10 data using the ImageNet mean and standard deviation, because this is what the network expects.
    - Add minor data augmentation (flip):

```
# training data
train_data_transform = transforms.Compose([
    transforms.Resize(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
])

train_set = torchvision.datasets.CIFAR10(root='./data',
                                         train=True,
                                         download=True,
transform=train_data_transform)

train_loader = torch.utils.data.DataLoader(train_set,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=2)
```

4. Follow the same steps with the validation/test data:

```
val_data_transform = transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
])

val_set = torchvision.datasets.CIFAR10(root='./data',
                                       train=False,
                                       download=True,
transform=val_data_transform)

val_order = torch.utils.data.DataLoader(val_set,
                                        batch_size=batch_size,
                                        shuffle=False,
                                        num_workers=2)
```

5. Choose a `device` – preferably GPU with a fallback on CPU:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
```

6. Define the training of the model. Unlike Keras, in PyTorch we have to iterate over the training data manually. This method iterates once over the whole training set (one epoch) and applies the optimizer after each forward pass:

```
def train_model(model, loss_function, optimizer, data_loader):
    # set model to training mode
    model.train()

    current_loss = 0.0
    current_acc = 0

    # iterate over the training data
    for i, (inputs, labels) in enumerate(data_loader):
        # send the input/labels to the GPU
        inputs = inputs.to(device)
        labels = labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        with torch.set_grad_enabled(True):
            # forward
            outputs = model(inputs)
            _, predictions = torch.max(outputs, 1)
            loss = loss_function(outputs, labels)

            # backward
            loss.backward()
            optimizer.step()

        # statistics
        current_loss += loss.item() * inputs.size(0)
        current_acc += torch.sum(predictions == labels.data)

    total_loss = current_loss / len(data_loader.dataset)
    total_acc = current_acc.double() / len(data_loader.dataset)

    print('Train Loss: {:.4f}; Accuracy: {:.4f}'.format(total_loss,
total_acc))
```

7. Define the testing/validation of the model. It's very similar to the training phase, but we will skip the backpropagation part:

```
def test_model(model, loss_function, data_loader):
    # set model in evaluation mode
    model.eval()

    current_loss = 0.0
    current_acc = 0

    # iterate over the validation data
    for i, (inputs, labels) in enumerate(data_loader):
        # send the input/labels to the GPU
        inputs = inputs.to(device)
        labels = labels.to(device)

        # forward
        with torch.set_grad_enabled(False):
            outputs = model(inputs)
            _, predictions = torch.max(outputs, 1)
            loss = loss_function(outputs, labels)

        # statistics
        current_loss += loss.item() * inputs.size(0)
        current_acc += torch.sum(predictions == labels.data)

    total_loss = current_loss / len(data_loader.dataset)
    total_acc = current_acc.double() / len(data_loader.dataset)

    print('Test Loss: {:.4f}; Accuracy: {:.4f}'.format(total_loss,
total_acc))
```

8. Define the first transfer learning scenario, where we use the pre-trained net as a feature extractor:
    - We'll use a popular network, ResNet-18. We'll talk about it in detail in the *Advanced network architectures* section. PyTorch will automatically download the pre-trained weights.
    - Replace the last network layer with a new layer with 10 outputs (one for each CIFAR-10 class).
    - Exclude the existing network layers from the backward pass, and only pass the newly-added fully-connected layer to the Adam optimizer.
    - Run the training for `epochs` and we'll evaluate the network accuracy after each epoch.

The following is the `tl_feature_extractor` function, which implements all this:

```
def tl_feature_extractor(epochs=3):
    # load the pre-trained model
    model = torchvision.models.resnet18(pretrained=True)

    # exclude existing parameters from backward pass
    # for performance
    for param in model.parameters():
        param.requires_grad = False

    # newly constructed layers have requires_grad=True by default
    num_features = model.fc.in_features
    model.fc = nn.Linear(num_features, 10)

    # transfer to GPU (if available)
    model = model.to(device)

    loss_function = nn.CrossEntropyLoss()

    # only parameters of the final layer are being optimized
    optimizer = optim.Adam(model.fc.parameters())

    # train
    for epoch in range(epochs):
        print('Epoch {}/{}'.format(epoch + 1, epochs))

        train_model(model, loss_function, optimizer, train_loader)
        test_model(model, loss_function, val_order)
```

9. Implement the fine-tuning approach. This function is similar to `tl_feature_extractor`, but now we'll train the whole network:

```
def tl_fine_tuning(epochs=3):
    # load the pre-trained model
    model = models.resnet18(pretrained=True)

    # replace the last layer
    num_features = model.fc.in_features
    model.fc = nn.Linear(num_features, 10)

    # transfer the model to the GPU
    model = model.to(device)

    # loss function
    loss_function = nn.CrossEntropyLoss()
    # We'll optimize all parameters
    optimizer = optim.Adam(model.parameters())

    # train
    for epoch in range(epochs):
        print('Epoch {}/{}'.format(epoch + 1, epochs))

        train_model(model, loss_function, optimizer, train_loader)
        test_model(model, loss_function, val_order)
```

10. Finally, we can run the whole thing in one of two ways:
    1. Call `tl_fine_tuning(epochs=5)` to use the fine tuning transfer learning approach for five epochs.
    2. Call `tl_feature_extractor(epochs=5)` to train the network with the feature extractor approach for five epochs.

With network as a feature extractor, we'll get about 76% accuracy, while with fine-tuning we'll get 87%. But if we run the fine-tuning for more epochs, the network starts to overfit.

Advanced network architectures

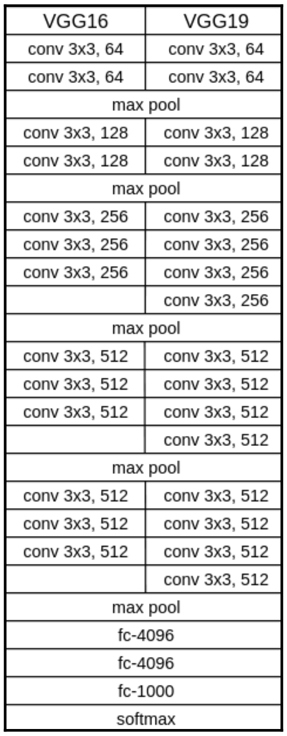
We are now familiar with the powerful technique of transfer learning. In this section, we'll discuss some recent and popular network architectures that go beyond the ones we've seen so far. You'll be able to use these networks as pre-trained models in a transfer learning scenario, or if you are brave enough, train them from scratch to solve your tasks.

VGG

The first architecture we're going to discuss is VGG (from Oxford's Visual Geometry Group, `https://arxiv.org/abs/1409.1556)`. It was introduced in 2014, when it became a runner-up in the ImageNet challenge of that year. The VGG family of networks remains popular today and is often used as a benchmark against newer architectures. Prior to VGG (for example, LeNet-5: `http://yann.lecun.com/exdb/lenet/`) and AlexNet (`https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`), the initial convolutional layers of a network used filters with large receptive fields, such as 7 x 7. Additionally, the networks usually had alternating single convolutional and pooling layers. The authors of the paper observed that a convolutional layer with a large filter size can be replaced with a stack of two or more convolutional layers with smaller filters (factorized convolution). For example, we can replace one 5 x 5 layer with a stack of two 3 x 3 layers, or a 7 x 7 layer with a stack of three 3 x 3 layers. This structure has several advantages:

- The neurons of the last of the stacked layers have the equivalent receptive field size of a single layer with a large filter.
- The number of weights and operations of stacked layers is smaller, compared to a single layer with large filter size. Let's assume we want to replace one 5 x 5 layer with two 3 x 3 layers. Let's also assume that all layers have an equal number of input and output channels (slices), `M`. The total number of weights (excluding biases) of the 5 x 5 layer is $5x5xMxM = 25xM^2$. On the other hand, the total weights of a single 3 x 3 layer is $3x3xMxM = 9xM^2$, and simply $2x(3x3xMxM) = 18xM^2$ for two layers, which makes this arrangement 28% more efficient (18/25 = 0.72). The efficiency will increase further with larger filters.
- Stacking multiple layers makes the decision function more discriminative.

The VGG networks consist of multiple blocks of two, three, or four stacked convolutional layers combined with a max-pooling layer. We can see the two most popular variants, VGG16 and VGG19, in the following table:



VGG16	VGG19
conv 3x3, 64	conv 3x3, 64
conv 3x3, 64	conv 3x3, 64
max pool	
conv 3x3, 128	conv 3x3, 128
conv 3x3, 128	conv 3x3, 128
max pool	
conv 3x3, 256	conv 3x3, 256
conv 3x3, 256	conv 3x3, 256
conv 3x3, 256	conv 3x3, 256
	conv 3x3, 256
max pool	
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
	conv 3x3, 512
max pool	
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
	conv 3x3, 512
max pool	
fc-4096	
fc-4096	
fc-1000	
softmax	

Architecture of VGG16 and VGG19 networks, named so after the number of weighted layers in each network

As the depth of the VGG network increases, so does the width (number of filters) in the convolutional layers. We have multiple pairs of convolutional layers with a volume depth of 128/256/512 connected to other layers with the same depth. In addition, we also have two 4,096-neuron fully-connected layers. Because of this, the VGG networks have large number of parameters (weights), which makes them memory-inefficient, as well computationally expensive. Still, this is a popular and straightforward network architecture, which has been further improved by the addition of batch normalization.

VGG with Keras, PyTorch, and TensorFlow

All three libraries have pre-trained VGG models. Let's see how to use them. We'll start with Keras, where it's easy to use this model in a transfer learning scenario. You can set `include_top` to `False`, which will exclude the fully-connected layers. The following are the steps:

1. Preload the weights by setting the `weights` parameter and they will be downloaded automatically:

```
# VGG16
from keras.applications.vgg16 import VGG16
vgg16_model = VGG16(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)

# VGG19
from keras.applications.vgg19 import VGG19
vgg19_model = VGG19(include_top=True, weights='imagenet',
input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

2. We'll continue with PyTorch, where you can choose whether you want to use a pre-trained model (again with automatic download):

```
import torchvision.models as models
model = models.vgg16(pretrained=True)
```

Finally, the process of using pre-trained models with TensorFlow isn't as straightforward; therefore we suggest the reader consult the official documentation on how to do this.

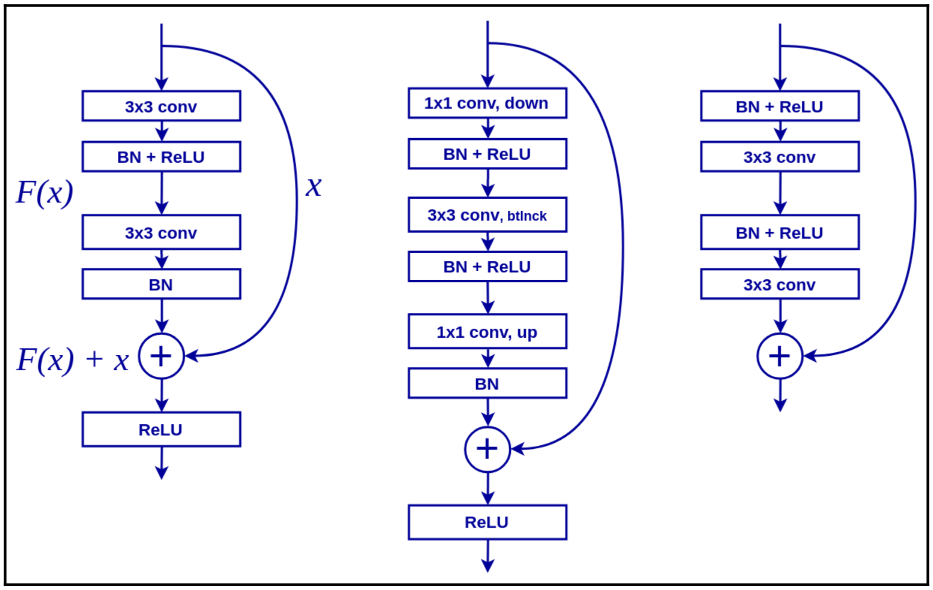
You can try other pre-trained models, using the same procedures we described. To avoid repetition, we won't include the same code examples for the other architectures in the section.

Residual networks

Residual networks (ResNets, `https://arxiv.org/abs/1512.03385`) were released in 2015, when they won all five categories of the ImageNet challenge that year. In `Chapter 2`, *Neural Networks*, we mentioned that the layers of a neural network are not restricted to sequential order, but form a graph instead. This is the first architecture we'll learn, which takes advantage of this flexibility. This is also the first network architecture that has successfully trained a network with the depth of more than 100 layers.

Thanks to better weight initializations, new activation functions, as well as normalization layers, it's now possible to train deep networks. But the authors of the paper conducted some experiments and observed that a network with 56 layers had higher training and testing errors compared to a network with 20 layers. They argue that this should not be the case. In theory, we can take a shallow network and stack identity layers (these are layers whose output just repeats the input) on top of it to produce a deeper network that behaves in exactly the same way as the shallow one. Yet, their experiments have been unable to match the performance of the shallow network.

To solve this problem, they proposed a network constructed of residual blocks. A residual block consists of two or three sequential convolutional layers and a separate parallel identity (repeater) shortcut connection, which connects the input of the first layer and the output of the last one. We can see three types of residual blocks in the following diagram:



Left: Original residual block. Middle: Original bottleneck residual block. Right: Residual block v2

Each block has two parallel paths. The left path is similar to the other networks we've seen, and consists of sequential convolutional layers + batch normalization. The right path contains the identity shortcut connection (also known as skip connection). The two paths are merged via an element-wise sum. That is, the left and right tensors have the same shape and an element of the first tensor is added to the element of the same position of the second tensor. The output is a single tensor with the same shape as the input. In effect, we propagate forward the features learned by the block, but also the original unmodified signal. In this way, we can get closer to the original scenario, as described by the authors. The network can decide to skip some of the convolutional layers thanks to the skip connections, in effect reducing its own depth. The residual blocks use padding in such a way that the input and the output of the block have the same dimensions. Thanks to this, we can stack any number of blocks for a network with arbitrary depth.

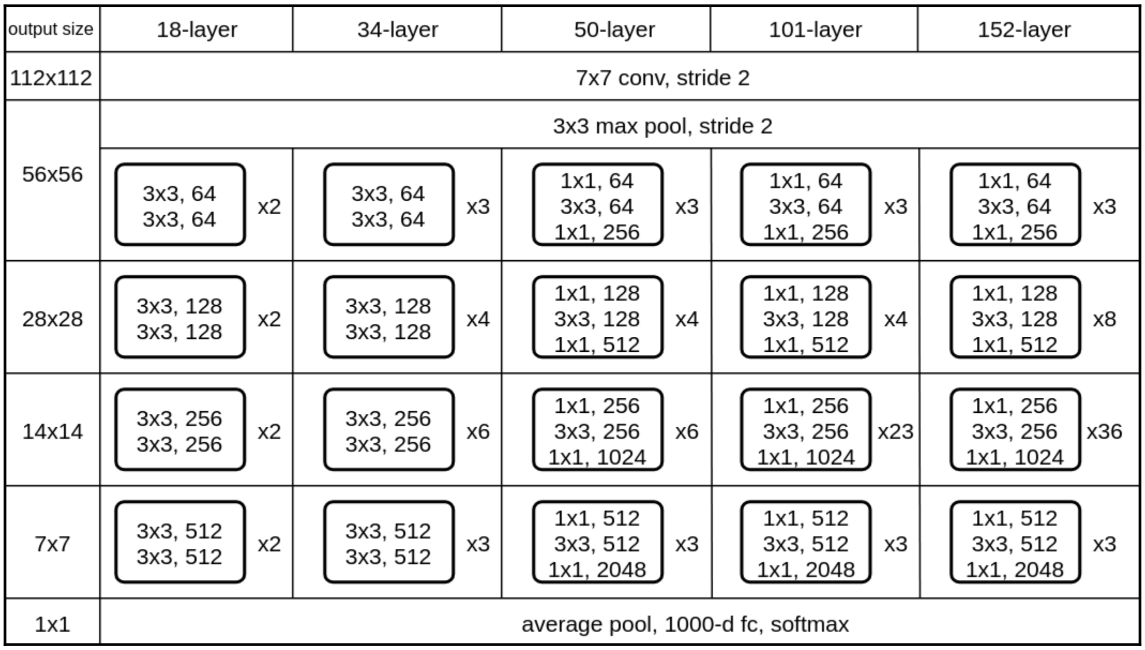
And now, let's see how the blocks in the diagram differ:

- The first block contains two 3 x 3 convolutional layers. This is the original residual block, but if the layers are wide, stacking multiple blocks becomes computationally expensive.
- The second block is equivalent to the first, but it uses the so-called bottleneck layer. First, we use a 1 x 1 convolution to downsample the input volume depth (we discussed this in `Chapter 4`, *Computer Vision with Convolutional Layers*). Then, we apply a 3 x 3 (bottleneck) convolution on the reduced input. Finally, we extend the output back to the desired depth with another 1 x 1 convolution. This layer is less computationally expensive than the first.
- The third block is the latest revision of the idea, published in 2016 by the same authors. It uses pre-activations –; the batch normalization and the activation function come before the convolutional layer. This may seem strange at first, but thanks to this design, the skip connection path can run uninterrupted throughout the whole network. This is contrary to the other residual blocks, where at least one activation function is on the path of the skip connection. A combination of stacked residual blocks still has the layers in the right order.

In the following table, we can see the family of networks proposed by the authors. Some of their properties are as follows:

- They start with a 7 x 7 convolutional layer with stride 2, followed by 3 x 3 max-pooling.

- Downsampling is implemented with a modified residual block with stride 2.
- Average pooling downsamples the output after all residual blocks and before the fully-connected layer:



output size	18-layer	34-layer	50-layer	101-layer	152-layer
112x112	7x7 conv, stride 2				
56x56	3x3 max pool, stride 2				
	3x3, 64 3x3, 64 x2	3x3, 64 3x3, 64 x3	1x1, 64 3x3, 64 1x1, 256 x3	1x1, 64 3x3, 64 1x1, 256 x3	1x1, 64 3x3, 64 1x1, 256 x3
28x28	3x3, 128 3x3, 128 x2	3x3, 128 3x3, 128 x4	1x1, 128 3x3, 128 1x1, 512 x4	1x1, 128 3x3, 128 1x1, 512 x4	1x1, 128 3x3, 128 1x1, 512 x8
14x14	3x3, 256 3x3, 256 x2	3x3, 256 3x3, 256 x6	1x1, 256 3x3, 256 1x1, 1024 x6	1x1, 256 3x3, 256 1x1, 1024 x23	1x1, 256 3x3, 256 1x1, 1024 x36
7x7	3x3, 512 3x3, 512 x2	3x3, 512 3x3, 512 x3	1x1, 512 3x3, 512 1x1, 2048 x3	1x1, 512 3x3, 512 1x1, 2048 x3	1x1, 512 3x3, 512 1x1, 2048 x3
1x1	average pool, 1000-d fc, softmax				

The family of the most popular residual networks. The residual blocks are represented by rounded rectangles

For more information about ResNets, check out the original paper, *Deep Residual Learning for Image Recognition* (`https://arxiv.org/abs/1512.03385`), by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, as well as the latest version, *Identity Mappings in Deep Residual Networks* (`https://arxiv.org/abs/1603.05027`), by the same authors.

Inception networks

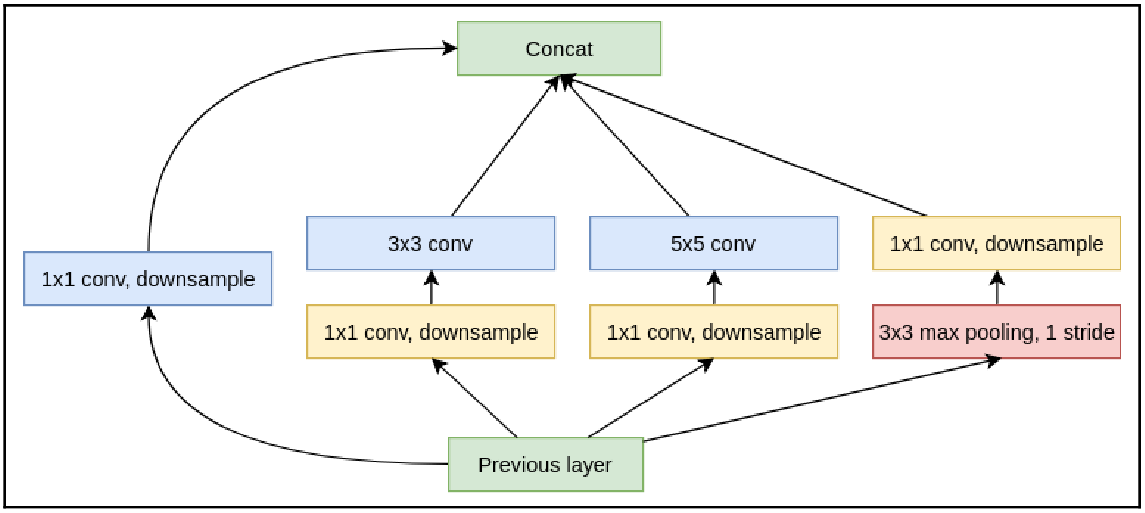
Inception networks (`https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf` by Szegedy et al.) were introduced in 2014, when they won the ImageNet challenge of that year (there seems to be a pattern here). Since then, the authors have released multiple improvements (versions) of the architecture.

Fun fact: the name inception comes in part from the “We need to go deeper” internet meme, related to the movie *Inception*.

The idea behind inception networks starts from the basic premise that the objects in an image have different scales. A distant object might take up a small region of the image, but the same object, once nearer, might take up the majority of the image. This presents a difficulty for standard CNNs, where the neurons in the different layers have a fixed receptive field size as imposed on the input image. A regular network might be a good detector of objects in a certain scale, but could miss them otherwise. To solve this problem, Szegedy et al proposed a novel architecture: one composed of inception blocks. An inception block starts with a common input, and then splits it into different parallel paths (or towers). Each path contains either convolutional layers with a different-sized filter, or a pooling layer. In this way, we apply different receptive fields on the same input data. At the end of the inception block, the outputs of the different paths are concatenated.

Inception v1

Here, we can see the first version of the inception block, part of the GoogLeNet network architecture. GoogLeNet contains nine such inception blocks; we can see them in the following diagram:



Inception v1 block

The v1 block has four paths:

- 1 x 1 convolution, which acts as a kind of repeater to the input
- 1 x 1 convolution, followed by a 3 x 3 convolution
- 1 x 1 convolution, followed by a 5 x 5 convolution
- 3 x 3 max pooling with a stride of 1

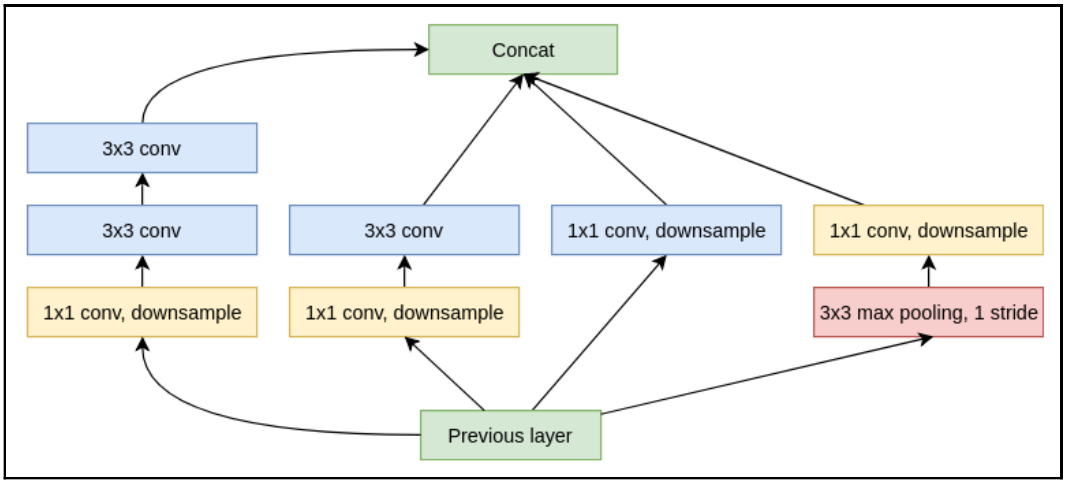
The layers in the block use padding in such a way that the input and the output have the same shape (but different depths). The padding is also necessary, because each path would produce output with a different shape, depending on the filter size. This is valid for all versions of the inception blocks.

The other major innovation of this inception block is the use of downsampling 1 x 1 convolutions. They are needed because the output of all paths is concatenated to produce the final output of the block. The result of the concatenation is an output with a quadrupled depth. If another inception blocks followed the current, its output depth would quadruple again. To avoid such exponential growth, the block uses 1 x 1 convolutions to reduce the depth for each path, which in turn reduces the output depth of the block. This makes it possible to create deeper networks, without running out of resources.

GoogLeNet also utilizes auxiliary classifiers – that is, it has two additional classification output (with the same groundtruth labels) at various intermediate layers. During training, the total value of the loss is a weighted sum of the auxiliary losses and the real loss. For more details about the architecture of GoogLeNet, we point the reader to the original paper, *Going Deeper with Convolutions* (`https://arxiv.org/abs/1409.4842`), by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich.

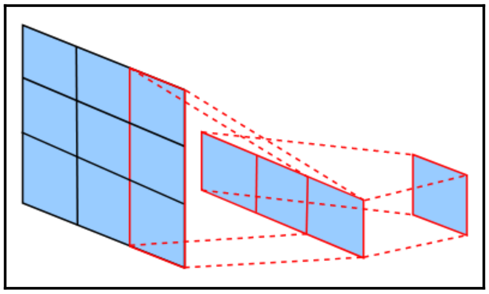
Inception v2 and v3

Inception v2 and v3 were released together and propose several improvements over the original inception block. The first is the factorization of the 5 x 5 convolution in two stacked 3 x 3 convolutions. We discussed the advantages of this in the *VGG* section. We can see the new inception block in the following diagram:



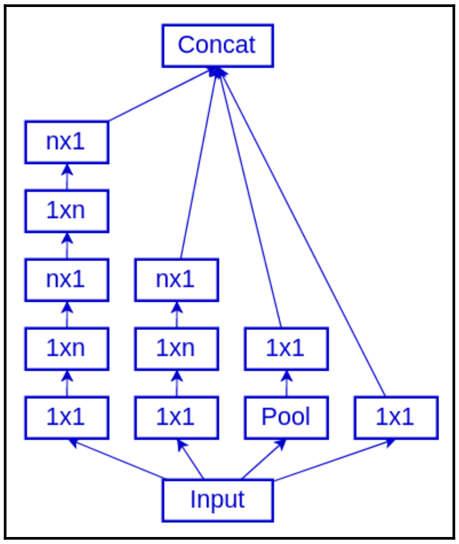
Inception block A

The next improvement is the factorization of an `nxn` convolution in two stacked asymmetrical `1xn` and `nx1` convolutions. For example, we can split a single 3 x 3 convolution into two 1 x 3 and 3 x 1 convolutions, where the 3 x 1 convolution is applied over the output of the 1 x 3 convolution. In the first case, the filter size would be 3 x 3 = 9, while in the second case we would have a combined size of (3 x 1) + (1 x 3) = 3 + 3 = 6, resulting in 33% efficiency:



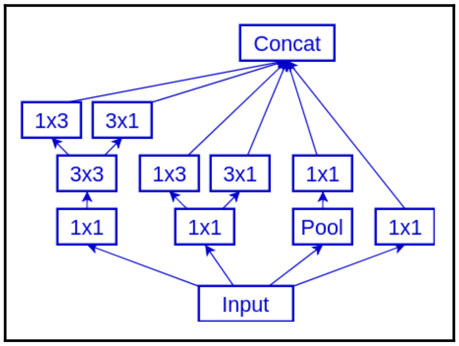
Factorization of a 3 x 3 convolution in 1 x 3 and 3 x 1 convolutions

The authors introduced two new blocks, which utilize factorized convolutions. The first (second in total) is the equivalent of block A, we introduced preceding. The following represents the image:



Inception block B. When n=3, it is equivalent to block A

The second (third in total) block is similar, but the asymmetrical convolutions are parallel, resulting in a higher output depth (more concatenated paths). The hypothesis here is that the more features (different filters) the network has, the faster it learns (we also discussed the need for more filters in `Chapter 4`, *Computer Vision with Convolutional Networks*). On the other hand, the wider layers take more memory and computation time. As a compromise, this block is only used in the deeper part of the network, after the other blocks:

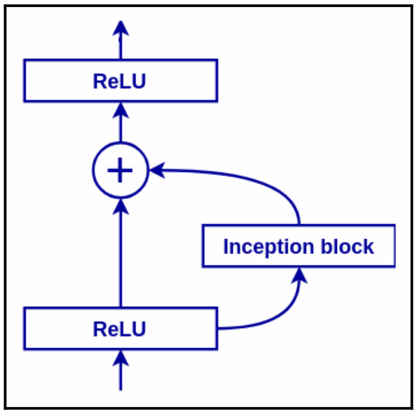


Inception block C

Using these new blocks, the authors proposed two new inception networks: v1 and v2. Another major improvement in this version is the use of batch normalization, which was introduced by the same authors. For more information about Inception v2 and v3, check out the original paper, *Rethinking the Inception Architecture for Computer Vision* (`https://arxiv.org/abs/1512.00567`), by Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna, as well as *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* (`https://arxiv.org/abs/1512.00567`), by Sergey Ioffe and Christian Szegedy.

Inception v4 and Inception-ResNet

In the latest revision of inception networks, the authors introduce three new streamlined inception blocks that build upon the idea of the previous versions. They introduce 7 x 7 asymmetric factorized convolutions, and average pooling instead of max pooling. More importantly, they create a residual/inception hybrid network known as Inception-ResNet, where the inception blocks also include residual connections. We can see the schematic of one such block in the following diagram:



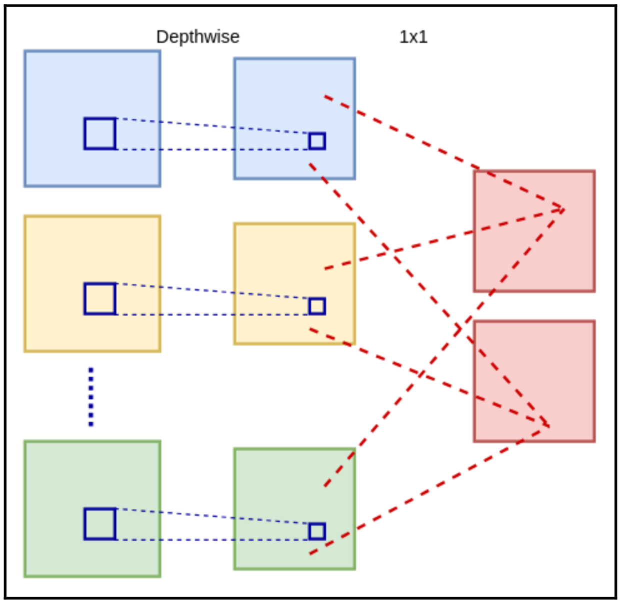
An inception block with residual skip connection

For more information about the new inception blocks and the network architectures, check out the original paper, *Inception-v4, Inception-ResNet, and the Impact of Residual Connections on Learning* (`https://arxiv.org/abs/1602.07261`), by Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi.

Xception and MobileNets

The last inception network we'll discuss is Xception (from Extreme Inception). To understand its hypothesis, let's recall that in `Chapter 3`, *Deep Learning Fundamentals, Computer Vision, and Convolutional Layers*, we introduced standard and depthwise convolutions. An output slice in standard convolution receives input from all input slices using a single filter. The filter tries to learn features in a 3D space, where two of the dimensions are spatial (the height and width of the slice) and the third is the channel. Therefore, the filter maps both spatial and cross-channel correlations.

All inception blocks so far have started with a dimensionality-reduction 1 x 1 convolution. From our new point of view, this connection maps cross-channel correlations, but not spatial ones (because of the 1 x 1 filter size). On the other hand, the subsequent operations in an inception block are standard convolutions, therefore mapping both types of correlations. The author of Xception argues that, in fact, we can completely decouple cross-channel and spatial correlations. We can do this with the so-called depthwise separable convolutions. A depthwise separable convolution combines two operations: a depthwise convolution and a 1 x 1 convolution. In a depthwise convolution, a single input slice produces a single output slice, therefore it only maps spatial (and not cross-channel) correlations. With 1 x 1 convolutions, we have the opposite. The following image represents the depthwise convolution:



A depthwise separable convolution

Let's compare the standard and depthwise separable convolutions. Imagine that we have 32 input and output channels, and a filter with a size of 3 x 3. In a standard convolution, one output slice is the result of applying one filter for each of the 32 input slices for a total of 32 x 3 x 3 = 288 weights (excluding bias). In a comparable depthwise convolution, the filter has only 3 x 3 = 9 weights and the filter for the 1 x 1 convolution has 32 x 1 x 1 = 32 weights. The total number of weights is 32 + 9 = 41. Therefore, the depthwise separable convolution is both faster and more memory-efficient compared to the standard one.

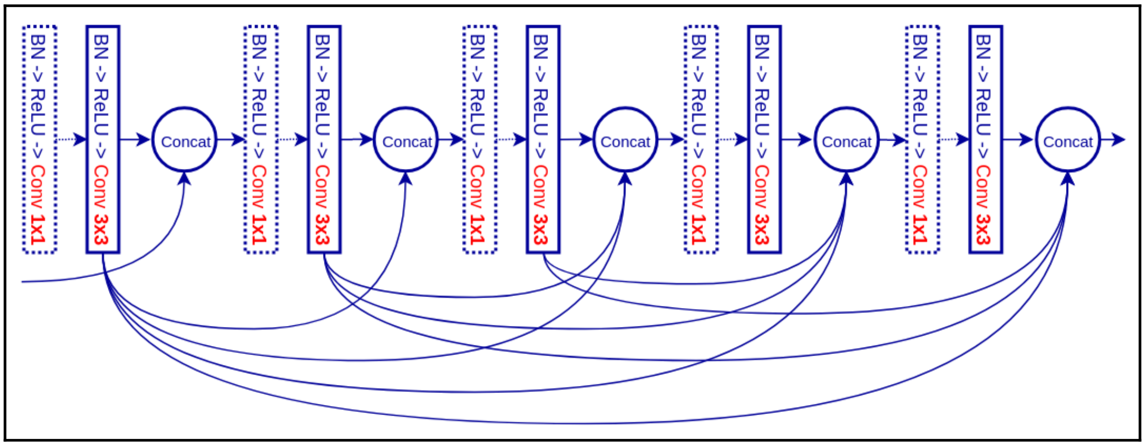
We can think of the depthwise separable convolution as an extreme (hence the name) version of an inception block, where each depthwise input/output slice pair represents one parallel path. We have as many parallel paths as the number of input slices. One difference with the other inception blocks is that the 1 x 1 convolution comes last, instead of first. But these operations are meant to be stacked anyway, and we can assume that the order is of no significance. Another difference is the absence of non-linear activation (ReLU or ELU) between the two operations. According to the author's experiments, networks with absent, non-linearity depthwise convolution converged faster and were more accurate.

The Xception network is built entirely of depthwise separable convolutions and it also includes residual connections. For more information, check out the original paper, *Xception: Deep Learning with Depthwise Separable Convolutions* (`https://arxiv.org/abs/1610.02357`), by François Chollet.

MobileNets are another class of models, built with depthwise separable convolutions. These networks are lightweight and specifically optimized for mobile and embedded applications. You can read more about them in the original paper, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications* (`https://arxiv.org/abs/1704.04861`), by Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, as well as the new version, *MobileNetV2: Inverted Residuals and Linear Bottlenecks* (`https://arxiv.org/abs/1801.04381`), by Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen.

DenseNets

DenseNet stands for Densely-Connected Convolutional Networks. It tries to alleviate the vanishing gradient problem and improve feature propagation, while reducing the number of network parameters. We've already seen how ResNets introduce residual blocks with skip connections to solve this. DenseNets take some inspiration from this idea and introduce dense blocks. A dense block consists of sequential convolutional layers, where any layer has a direct connection to all subsequent layers:



A dense block: The dimensionality-reduction layers (dashed lines) are part of the DenseNet-B architecture, while the original DenseNet doesn't have them

Here are some properties of the dense block:

- The different inputs are merged via concatenation, unlike ResNets, which use sum.
- A batch normalization and ReLU are applied over each concatenation, and then the result is fed to the following convolutional layer.
- A dense block is specified by its number of convolutional layers and the output volume depth of each layer, which is called **growth rate** in this context. Let's assume that the input of the dense block has a volume depth of k_0 and the output volume depth of each convolutional layer is `k`. Then, because of the concatenation, the input volume depth for the l-th layer will be $k_0 + k \times (l - 1)$. The authors also introduced a second type of dense net, DenseNet-B, which applies a dimensionality-reduction 1 x 1 convolution after each concatenation.
- Although the later layers of a dense block have a large input volume depth (because of the many concatenations), DenseNets can work with growth rate values as low as 12, which reduces the total number of parameters.

- To make concatenation possible, dense blocks use padding in such a way that the height and width of all output slices are the same throughout the block. The network uses average pooling between the dense blocks for downsampling.

For more information about DenseNets, check out the original paper, *Densely Connected Convolutional Networks* (`https://arxiv.org/abs/1608.06993`) by Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger.

Capsule networks

Capsule networks were introduced by Geoffrey Hinton as a way to overcome some of the limitations of standard CNNs. To understand the idea behind capsule networks, we need to understand these limitations first.

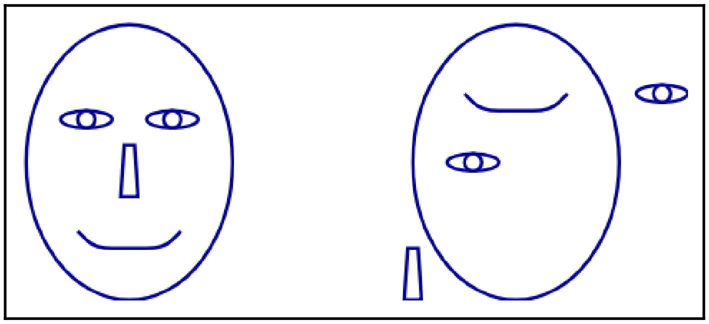
Limitations of convolutional networks

Let's start with a quote from professor Hinton himself:

> *"The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster."*

What he means is that the CNNs are **translation-invariant**. To understand this, let's imagine a picture with a face, located in the right half of the picture. Translation invariance means that a CNN is very good at telling us that the picture contains a face, but it cannot tell us whether the face is in the left or right part of the image. The main culprit for this behavior is the pooling layers. Every pooling layer introduces a little translation invariance. For example, the max pooling routes forward the activation of only one of the input neurons, but the subsequent layers don't have any knowledge of which neuron is routed. By stacking multiple pooling layers, we gradually increase the receptive field size. But the detected object can be anywhere in the new receptive field, because none of the pooling layers relay such information. Therefore, we also increase the translation invariance. At first, this might seem to be a good thing, because the final labels have to be translation-invariant. But it poses a problem, as CNNs cannot identify the position of one object relative to another. It would identify both images following image as a face, because they both contain the ingredients of a face, a nose, mouth, and eyes, regardless of their relative positions to one another.

This is also known as the "Picasso problem," as demonstrated in the following diagram:



A convolutional network would identify both of these images as a face

But that's not all. A CNN would be confused even if the face had a different **orientation**, for example, if it was turned upside down. One way to overcome this is with data augmentation (rotation) during training. But this only shows the limitations of the network. We have to explicitly show it the object in different orientations and tell it that this is, in fact, the same object.

So far, we've seen that a CNN discards the translation information (transitional invariance) and doesn't understand the orientation of an object. In computer vision, the combination of translation and orientation is known as **pose**. The pose is enough to uniquely identify the object's properties in the coordinate system. Let's use computer graphics to illustrate this. A 3D object, say a cube, is entirely defined by its pose and the edge length. The process of transforming the representation of a 3D object into an image on the screen is called rendering. Knowing just its pose and the edge length of the cube, we can render it from any point of view we like. Therefore, if we can somehow train a network to understand these properties, we won't have to feed it with multiple augmented versions of the same object. A CNN cannot do that, because its internal data representation doesn't contain information about the object's pose (only about its type). In contrast, capsule networks *preserve information* for both the type and the pose of an object. Therefore, they can detect objects that can transform to each other, which is known as **equivariance**. We can also think of this as "reverse graphics," that is, a reconstruction of the object's properties by its rendered image.

Capsules

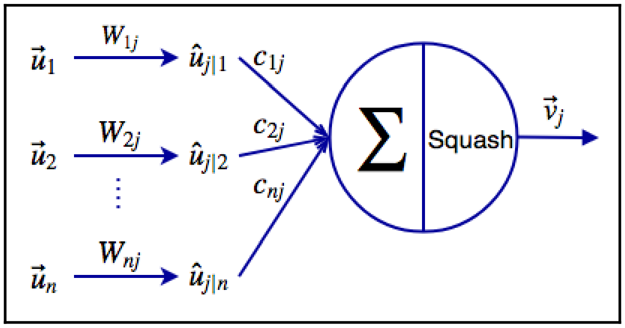
To solve these problems, the authors of the paper propose a new type of network building block, called a **capsule**, instead of the neuron. The output of a neuron is a scalar value. In contrast, the output of a capsule is a vector (a list of values), which consists of the following:

- The elements of the vector represent the pose and other properties of the object.
- The length of the vector is in the (0, 1) range and represents the probability of detecting the feature at that location. As a reminder, the length of a vector is $||\vec{v}|| = \sqrt{\sum_{i=1}^{n} v_i^2}$, where v_i are the vector elements.

Let's consider a capsule, which detects faces. If we start moving a face across the image, the values of the capsule vector will change to reflect the change in the position. However, its length will always stay the same, because the probability of the face doesn't change with the location.

The capsules are organized in interconnected layers, just such as a regular network. The capsules in one layer serve as input to the capsules in the next. And such as a CNN, the earlier layers detect basic features, and the deeper layers combine them in more abstract and complex ones. But now the capsules also relay positional information, instead of just detected objects. This allows the deeper capsules to analyze not only the presence of features, but also their relationship. For example, a capsule layer may detect a mouth, face, nose, and eyes. The subsequent capsule layer will be able to not only verify the presence of these features, but also whether they have the correct spatial relationship. Only if both conditions are true can the subsequent layer verify that a face is present. This is a high-level overview of capsule networks. Now, let's see how exactly capsules work.

We can see the schematic of a capsule in the following diagram:



A capsule

Let's analyze it in the following bullets:

- The capsule inputs are the output vectors, $u_1, u_2, ... u_n$, from the capsules of the previous layer.
- We multiply each vector, u_i, by its corresponding weight matrix, W_{ij}, to produce **prediction vectors**, $\hat{u}_{j|i} = W_{1j}u_i$. The weight matrices, *W*, encode spatial and other relationships between the lower-level features, coming from the capsules of the previous layer, and the high-level ones in the current layer. For example, imagine that the capsule in the current layer detects faces and the capsules from the previous layer detect the mouth (u_1), eyes (u_2), and nose (u_3). Then, $\hat{u}_{j|1} = W_{1j}u_1$ is the predicted position of the face, given where the location of the mouth is. In the same way, $\hat{u}_{j|2} = W_{2j}u_2$ predicts the location of the face based on the detected location of the eyes, and $\hat{u}_{j|3} = W_{3j}u_3$ predicts the location of the face based on the location of the nose. If all three lower-level capsule vectors agree on the same location, then the current capsule can be confident that a face is indeed present. We only used location for this example, but the vectors could encode other types of relationships between the features, such as scale and orientation. The weights, *W*, are learned with backpropagation.
- Next, we multiply the $\hat{u}_{i|j}$ vectors by the scalar coupling coefficients, c_{ij}. These coefficients are a separate set of parameters, apart from the weight matrices. They exist between any two capsules and indicate which high-level capsules will receive input from a lower-level capsule. But unlike weight matrices, which are adjusted via backpropagation, coupling coefficients are computed on the fly during the forward pass via a process called **dynamic routing**. We'll describe it in the next section.
- Then, we perform the sum of the weighted input vectors. This step is similar to the weighted sum in neurons, but with vectors:

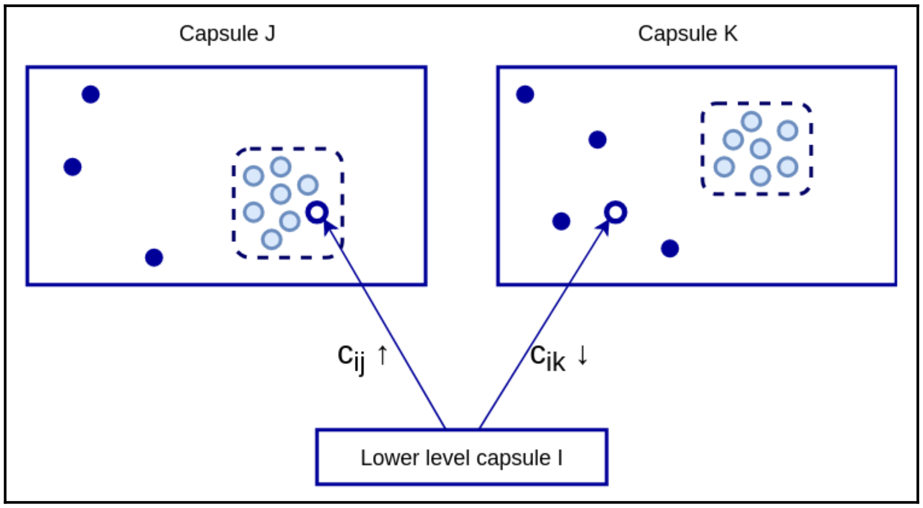
$$\vec{s}_j = \sum_i c_{ij}\hat{u}_{j|i}$$

- Finally, we'll compute the output of the capsule, v_j, by squashing the vector, s_j. In this context, squashing means transforming the vector in such a way that its length comes in the (0, 1) range, without changing its direction. As mentioned, the length of the capsule vector represents the probability of the detected feature and squashing it in the (0, 1) range reflects that. To do this, the authors propose a novel formula:

$$\vec{v}_j = \frac{||\vec{s}_j||^2}{1+||\vec{s}_j||^2}\frac{\vec{s}_j}{||\vec{s}_j||}$$

Dynamic routing

Let's describe the dynamic routing process to compute the coupling coefficients, c_{ij}. In the following diagram, we have a lower capsule, *I*, that has to decide whether to send its output to one of two higher-level capsules, *J* and *K*. The dark and light dots represent prediction vectors, $\hat{u}_{j|*}$ and $\hat{u}_{k|*}$, which *J* and *K* have already received from other lower-level capsules. The arrows from the *I* capsule to the *J* and *K* capsules point to the $\hat{u}_{j|i}$ and $\hat{u}_{k|i}$ prediction vectors from *I* to *J* and *K*:



Dynamic routing example. The grouped dots indicate lower-level capsules that agree with each other

The clustered prediction vectors (lighter dots) indicate lower-level capsules that agree with each other with regards to the high-level feature. For example, if the *K* capsule describes a face, then the clustered predictions would indicate lower-level features, such as mouth, nose, and eyes. Conversely, the dispersed (darker) dots indicate disagreement. If the *I* capsule predicts a vehicle tire, it would disagree with the clustered predictions in *K*.

However, if the clustered predictions in *J* represent features such as headlights, windshield, or fenders, then the prediction of *I* would be in agreement with them. The lower-level capsules have a way of determining whether they fall in the clustered or dispersed group of each higher-level capsule. If they fall in the clustered group, they will increase the corresponding coupling coefficient with that capsule and will route their vector in that direction. Conversely, if they fall in the dispersed group, the coefficient will decrease.

Let's formalize this knowledge with a step-by-step algorithm, introduced by the authors:

1. For all *i* capsules in the *l* layer, and *j* capsules in the *(l + 1)* layer, we'll initialize $b_{ij} \leftarrow 0$, where b_{ij} is a temporary variable equivalent to c_{ij}. The vector representation of all b_{ij} is $\vec{b}_i$. At the start of the algorithm, the *i* capsule has an equal chance to route its output to any of the capsules of the *(l + 1)* layer.
2. Repeat for *r* iterations, where *r* is a parameter:
    - For all *i* capsules in the *l* layer: $\vec{c}_i \leftarrow softmax(\vec{b}_i)$. The sum of all outgoing coupling coefficients, c_i, of a capsule amounts to 1 (they have a probabilistic nature), hence the softmax.
    - For all *j* capsules in the *(l + 1)* layer: $\vec{s}_j \leftarrow \sum_i c_{ij}\hat{u}_{j|i}$. That is, we'll compute all non-squashed output vectors of the *(l + 1)* layer.
    - For all *j* capsules in the *(l + 1)* layer, we'll compute the squashed vectors: $\vec{v}_j \leftarrow squash(\vec{s}_j)$.
    - For all *i* capsules in the *l* layer, and *j* capsules in the *(l + 1)* layer: $b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i} \cdot \vec{v}_j$. Here, $\hat{u}_{j|i} \cdot \vec{v}_j$ is the dot product of the prediction vector of the low-level *i* capsule and the output vector of the high-level *j* capsule vectors. If the dot product is high, then the *i* capsule is in agreement with the other low-level capsules, which route their output to the *j* capsule, and the coupling coefficient increases.

The authors have recently released an updated dynamic routing algorithm using a clustering technique called Expectation–Maximization. You can read more about it in the original paper, *Matrix Capsules with EM Routing* (`https://ai.google/research/pubs/pub46653`) by Geoffrey Hinton, Sara Sabour, and Nicholas Frosst.

Structure of the capsule network

In this section, we'll describe the structure of the capsule network, which the authors used to classify the MNIST dataset. The input of the network is the 28 x 28 MNIST greyscale images and the following are the steps:

1. We'll start with a single convolutional layer with 256 9 x 9 filters, stride 1, and ReLU activation. The shape of the output volume is (256, 20, 20).
2. We have another convolutional layer with 256 9 x 9 filters and stride 2. The shape of the output volume is (256, 6, 6).
3. Use the output of the layer as a foundation for the first capsule layer, called PrimaryCaps. Take the (256, 6, 6) output volume and split it in to 32 separate (8, 6, 6) blocks. That is, each of the 32 blocks contains eight 6 x 6 slices. Take one activation value with the same coordinates from each slice and combine these values in a vector. For example, we can take activation (3, 7) of slice 1, (3, 7) of slice 2, and so on and combine them in a vector with a length 8. We'll have 36 of these vectors. Then we'll "transform" each vector into a capsule for a total of 36 capsules. The shape of the output volume of the PrimaryCaps layer is (32, 8, 6, 6).
4. The second capsule layer is called DigitCaps. It contains 10 capsules (one per digit), whose output is a vector with length which is 16. The shape of the output volume of the DigitCaps layer is (10, 16). During inference, we compute the length of each DigitCaps capsule vector. We then take the capsule with the longest vector as the prediction result of the network.
5. During training, the network includes three additional, fully-connected layers after DigitCaps, the last of which has 784 neurons (28 x 28). In the forward training pass, the longest capsule vector serves as input to these layers. They try to reconstruct the original image, starting from that vector. Then, the reconstructed image is compared to the original one and the difference serves as additional regularization loss for the backward pass.

Capsule networks are a new and promising approach to computer vision. However, they are not widely adopted yet and don't have an official implementation in any of the deep learning libraries discussed in this book, but you can find multiple third-party implementations.

For more information about capsule networks, check out the original paper, *Dynamic Routing Between Capsules* (`https://arxiv.org/abs/1710.09829`), by Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton.

Advanced computer vision tasks

So far, we've discussed classification tasks ; a CNN can tell us what object is in the image and a confidence score, but nothing more. In this section, we'll discuss two more advanced and interesting tasks: object detection and semantic segmentation.

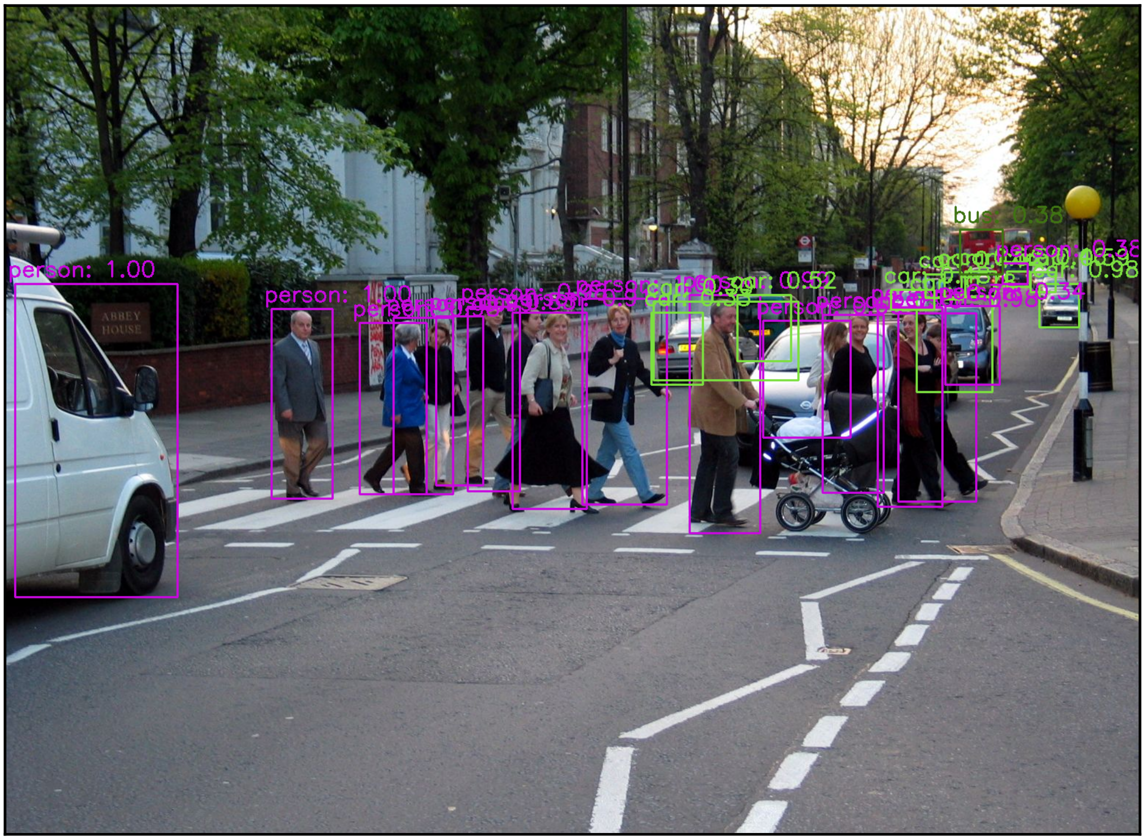
Object detection

Object detection is the process of finding object instances of a certain class, such as faces, cars, and trees, in images or videos. Unlike classification, object detection can detect multiple objects, as well as their location in the image.

An object detector would return a list of detected objects with the following information for each object:

- The class of the object (person, car, tree, and so on).
- Probability (or confidence score) in the [0, 1] range, which conveys how confident the detector is that the object exists in that location. This is similar to the output of a regular classifier.
- The coordinates of the rectangular region of the image where the object is located. This rectangle is called a **bounding box**.

We can see the typical output of an object-detection algorithm in the following photograph. The object type and confidence score are above each bounding box:



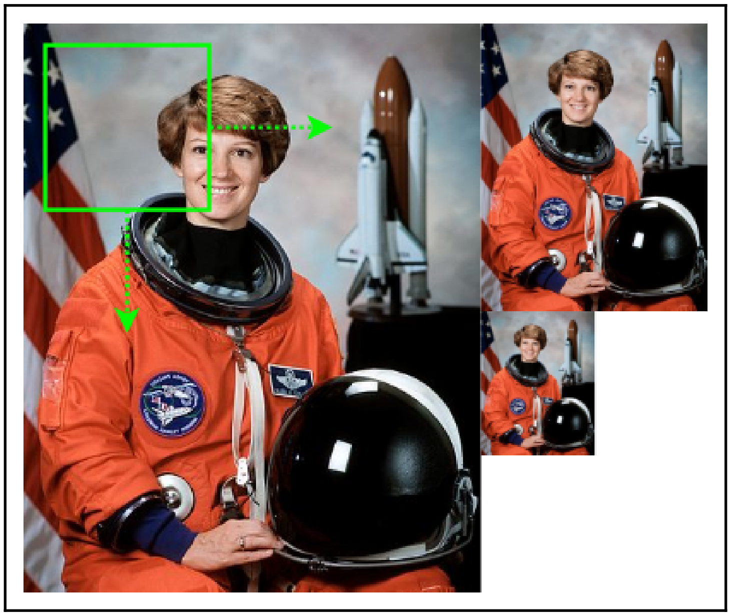
The output of an object detector. The vehicle on the left is wrongly classified as person, but the rest of the objects are classified correctly.

Approaches to object detection

In this section, we'll outline three approaches:

- **Classic sliding window:** Here, we'll use a regular classification network (classifier). This approach can work with any type of classification algorithm, but it's relatively slow and error-prone:
    1. Build an image pyramid. This is a combination of different scales of the same image (see the following photograph). For example, each scaled image can be two times smaller than the previous one. In this way, we'll be able to detect objects regardless of their size in the original image.
    2. Slide the classifier across the whole image. That is, we'll use each location of the image as an input to the classifier and the result will determine what type of object is in that location. The bounding box of that location is just the image region that we used as input.
    3. We'll have multiple overlapping bounding boxes for each object. We'll use some heuristics to combine them in a single prediction.

    Here is an illustration of the sliding window approach:


Sliding window + image pyramid object detection

- **Two-stage detection methods:** These methods are very accurate, but relatively slow. As the name suggests, they involve two steps:
    1. A special type of CNN, called a Region Proposal Network, scans the image and proposes a number of possible bounding boxes where objects might be located. However, this network doesn't detect the type of the object, but only whether an object is present in the region.
    2. The regions of interest are sent to the second stage for object classification.
- **One-stage detection methods:** Here, a single CNN produces both the object type and the bounding box. These approaches are usually faster, but less accurate compared to two-stage methods.

Object detection with YOLOv3

In this section, we'll discuss one of the most popular detection algorithms, called YOLO. The name is an acronym for the popular motto "You only live once," which reflects the one-stage nature of the algorithm. The authors have released three versions with incremental improvements of the algorithm. We'll first discuss the latest, v3.

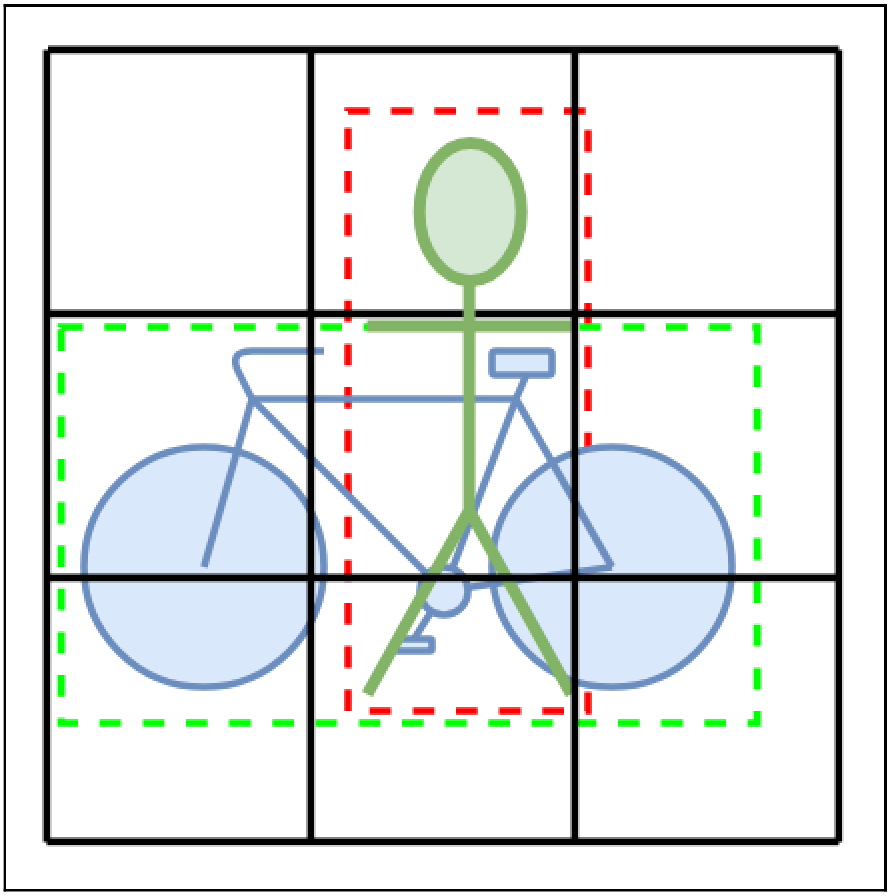
Before diving deeper (pun intended), we should mention a few things about YOLO:

- It works with a fully-convolutional network (without pooling layers), not unlike the ones we've seen in this chapter. It uses residual connections and batch normalization. The YOLOv3 network uses three different scales of the image for prediction. What makes it different, though, is the use of special type of groundtruth/output data, which is a combination of classification and regression.
- The network takes the whole image as an input and outputs the bounding boxes, object classes, and confidence scores of all detected objects in just a single pass. For example, the bounding boxes in the image of people on the crosswalk at the beginning of this section were generated using a single network pass.

With that introduction, let's see how YOLO works:

1. Split the image into a grid of *S x S* cells (in the following diagram, we can see a 3 x 3 grid):
    - The network treats the center of each grid cell as the center of the region, where an object might be located.
    - An object might lie entirely within a cell. Then, its bounding box will be smaller than the cell. Alternatively, it can span over multiple cells and the bounding box will be larger. YOLO covers both cases.

- YOLO can detect multiple objects in a grid cell with the help of **anchor boxes** (more on that later), but an object is associated with one cell only (1-to-n relation). That is, if the bounding box of the object covers multiple cells, we'll associate the object with the cell, where the center of the bounding box lies. For example, the two objects in the following diagram span multiple cells, but they are both assigned to the central cell, because their centers lie in it.
- Some of the cells may contain object and others might not. We are only interested in the ones that do:



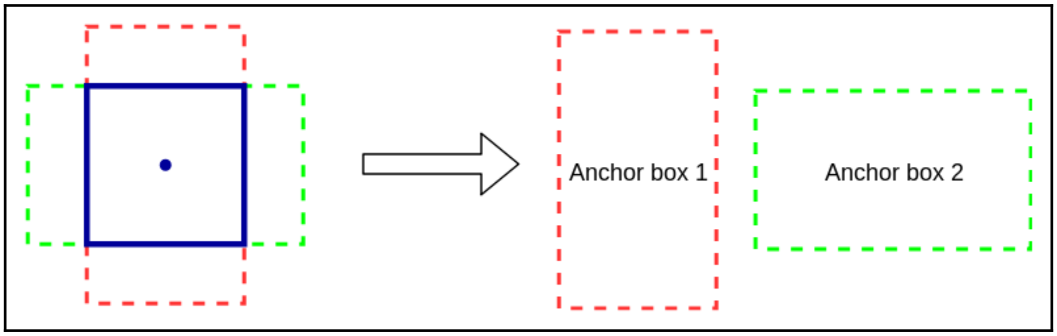
An object detection YOLO example with a 3 x 3 cell grid, 2 objects, and their bounding boxes (dashed lines). Both objects are associated with the middle cell, because the centers of their bounding boxes lie in that cell

2. The network output and target data is a one-stage classifier. The network outputs possible detected objects for each grid cell. For example, if the grid is 3 x 3, then the output will contain nine possible detected objects. For the sake of clarity, let's discuss the output data (and its corresponding label) for a single grid cell/detected object. It is an array with values, `[`b_x`, `b_y`, `b_h`, `b_w`, `p_c`, `c_1`, `c_2`, ..., `c_n`]`, where:
    - b_x, b_y, b_h, b_w describes the bounding box, (if an object exists). b_x and b_y are the coordinates of the upper-left coordinate of the box. They are normalized in the [0, 1] range with respect to the size of the image. That is, if the image is of size 100 x 100 and $b_x = 20$ and $b_y = 50$, their normalized values would be 0.2 and 0.5. b_h and b_w represent the box height and width. They are normalized with respect to the grid cell. If the bounding box is larger than the cell, its value will be greater than 1. Predicting the box parameters is a regression task.
    - p_c is a confidence score in the [0, 1] range. The labels for the confidence score are either 0 (not present) or 1 (present), making this part of the output a classification task. If an object is not present, we can discard the rest of the array values.
    - c_1, c_2, ..., c_n is a one-hot encoding of the object class. For example, if we have car, person, tree, cat, and dog classes, and the current object is of the cat type, its encoding will be `[0, 0, 0, 1, 0]`. If we have `n` possible classes, the size of the output array for one cell would be `5 + n` (9 in our example).

   The network output/labels will contain `SxS` such arrays. For example, the length of the YOLO output for a 3 x 3 cell grid and four classes will be 3 x 3 x 9 = 81.

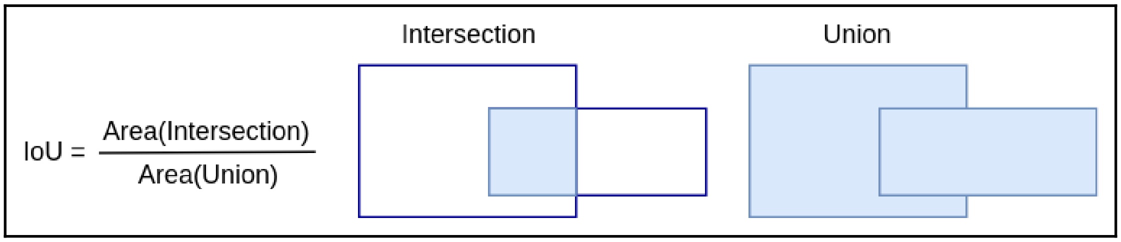
3. Let's address the scenario with multiple objects in the same cell. Thankfully, YOLO proposes an elegant solution to this problem. We'll have multiple candidate boxes (known as anchor boxes or priors) with a slightly different shape for each cell. In the following diagram, we can see the grid cell (square, uninterrupted line) and two anchor boxes – vertical and horizontal (dashed lines). If we have multiple objects in the same cell, we'll associate each object with one of the anchor boxes. Conversely, if an anchor box doesn't have an associated object, it will have a confidence score of 0. This arrangement will also change the network output. We'll have multiple output arrays per grid cell (one output array per anchor box). To extend our previous example, let's assume we have a 3 x 3 cell grid with four classes and two anchor boxes per cell. Then, we'll have 3 x 3 x 2 = 18 output bounding boxes and a total output length of 3 x 3 x 2 x 9 = 162.

Following is a figure of a grid cell with two anchor boxes:



Grid cell (square, uninterrupted line) with two anchor boxes (dashed lines)

The only question now is how to choose the proper anchor box for an object during training (during inference the network will choose by itself). We'll do this with the help of Intersection over Union (IoU). This is just the ratio between the area of the intersection of the object bounding box/anchor box, and the area of their union:



Intersection over Union (IoU)

We'll compare the bounding box of each object to all anchor boxes, and assign the object to the anchor box with the highest IoU.

4. Now that we (hopefully) know how YOLO works, we can use it for predictions. However, the output of the network might be noisy – that is, the output includes all possible anchor boxes for each cell, regardless of whether an object is present in them. Many of these boxes will overlap and actually predict the same object. We'll get rid of the noise using **non-maximum suppression**. Here's how it works:
    1. Discard all bounding boxes with a confidence score <= 0.6.
    2. From the remaining bounding boxes, pick the one with the highest possible confidence score.
    3. Discard any box whose IoU >= 0.5 with the box we selected in the previous step.

If you are worried that the network output/groundtruth data will become too complex or large, don't be. CNNs work well with the ImageNet dataset, which has 1,000 categories, and therefore 1,000 outputs.

For more information about YOLO, check out the original sequence of papers:

- You Only Look Once: Unified, Real-Time Object Detection (`https://arxiv.org/abs/1506.02640`) by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi
- YOLO9000: Better, Faster, Stronger (`https://arxiv.org/abs/1612.08242`) by Joseph Redmon and Ali Farhadi
- YOLOv3: An Incremental Improvement (`https://arxiv.org/abs/1804.02767`) by Joseph Redmon and Ali Farhadi

A code example of YOLOv3 with OpenCV

In this section, we'll demonstrate how to use the YOLOv3 object detector with OpenCV. For this example, you'll need OpenCV 3.4.2 or higher, and 250 MB of disk space for the pre-trained YOLO network. Let's begin with the following steps:

1. Start with the imports:

```
import os.path
import cv2  # opencv import
import numpy as np
import requests
```

2. Add some boilerplate code, which downloads and stores the following:
    - The YOLOv3 network configuration. We'll use the YOLO author's GitHub and personal website to do this.
    - The names of the classes that the network can detect. We'll also load them from the file.
    - A test image from Wikipedia. We'll also load the image from the file:

```
# Download YOLO net config file
# We'll it from the YOLO author's github repo
yolo_config = 'yolov3.cfg'
if not os.path.isfile(yolo_config):
    url =
'https://raw.githubusercontent.com/pjreddie/darknet/master/cfg/yolo
v3.cfg'
    r = requests.get(url)
    with open(yolo_config, 'wb') as f:
        f.write(r.content)

# Download YOLO net weights
# We'll it from the YOLO author's website
yolo_weights = 'yolov3.weights'
if not os.path.isfile(yolo_weights):
    url = 'https://pjreddie.com/media/files/yolov3.weights'
    r = requests.get(url)
    with open(yolo_weights, 'wb') as f:
        f.write(r.content)

# Download class names file
# Contains the names of the classes the network can detect
classes_file = 'coco.names'
if not os.path.isfile(classes_file):
    url =
'https://raw.githubusercontent.com/pjreddie/darknet/master/data/coc
o.names'
    r = requests.get(url)
    with open(classes_file, 'wb') as f:
        f.write(r.content)

# load class names
with open(classes_file, 'r') as f:
    classes = [line.strip() for line in f.readlines()]

# Download object detection image
image_file = 'source.jpg'
if not os.path.isfile(image_file):
    url =
"https://upload.wikimedia.org/wikipedia/commons/c/c7/Abbey_Road_Zeb
```

```
ra_crossing_2004-01.jpg"
    r = requests.get(url)
    with open(image_file, 'wb') as f:
        f.write(r.content)

# read and normalize image
image = cv2.imread(image_file)
blob = cv2.dnn.blobFromImage(image, 1 / 255, (416, 416), (0, 0, 0),
True, crop=False)
```

3. Initialize the network with the `weights` and `config` we just downloaded:

```
# Load the network
net = cv2.dnn.readNet(yolo_weights, yolo_config)
```

4. Feed the image to the network and do the inference:

```
# set as input to the net
net.setInput(blob)

# get network output layers
layer_names = net.getLayerNames()
output_layers = [layer_names[i[0] - 1] for i in
net.getUnconnectedOutLayers()]

# inference
# the network outputs multiple lists of anchor boxes,
# one for each detected class
outs = net.forward(output_layers)
```

5. Iterate over the classes and anchor boxes and prepare them for the next step:

```
# extract bounding boxes
class_ids = list()
confidences = list()
boxes = list()

# iterate over all classes
for out in outs:
    # iterate over the anchor boxes for each class
    for detection in out:
        # bounding box
        center_x = int(detection[0] * image.shape[1])
        center_y = int(detection[1] * image.shape[0])
        w = int(detection[2] * image.shape[1])
        h = int(detection[3] * image.shape[0])
        x = center_x - w // 2
        y = center_y - h // 2
```

```
            boxes.append([x, y, w, h])

            # class
            class_id = np.argmax(detection[5:])
            class_ids.append(class_id)

            # confidence
            confidence = detection[4]
            confidences.append(float(confidence))
```

6. Remove the noise with non-max suppression. You can experiment with different values of `score_threshold` and `nms_threshold` to see how the detected objects change. For example, setting `score_threshold=0.3` will detect more cars in the distance:

```
# non-max suppression
ids = cv2.dnn.NMSBoxes(boxes, confidences, score_threshold=0.3,
nms_threshold=0.5)
```

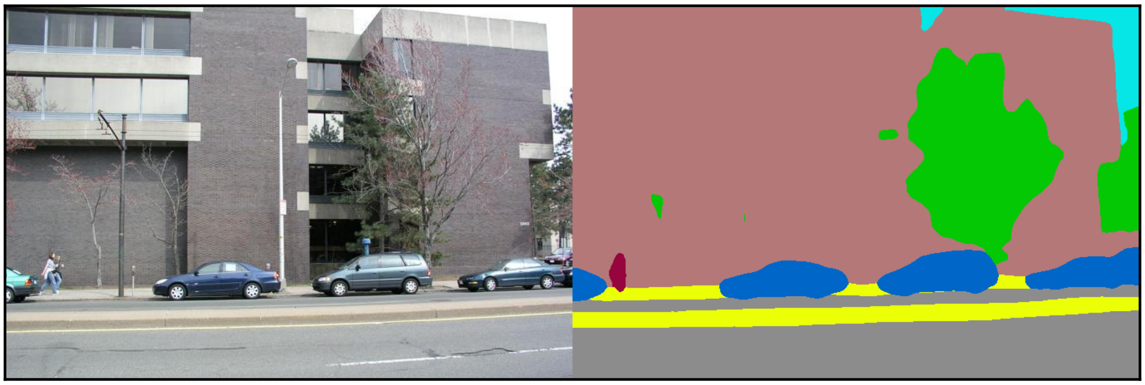
7. Draw the bounding boxes on the image and display the result:

```
# draw the bounding boxes on the image
colors = np.random.uniform(0, 255, size=(len(classes), 3))

for i in ids:
    i = i[0]
    x, y, w, h = boxes[i]
    class_id = class_ids[i]

    color = colors[class_id]

    cv2.rectangle(image, (round(x), round(y)), (round(x + w),
round(y + h)), color, 2)

    label = "%s: %.2f" % (classes[class_id], confidences[i])
    cv2.putText(image, label, (x - 10, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 1, color, 2)

cv2.imshow("Object detection", image)
cv2.waitKey()
```

If everything goes alright, this code block will produce the same image that we saw at the beginning of this section.

Semantic segmentation

Semantic segmentation is the process of assigning a class label (such as person, car, or tree) to each pixel of the image. You can think of it as classification, but on a pixel level – instead of classifying the entire image under one label, we'll classify each pixel separately. Here is an example of semantic segmentation:



Semantic segmentation

To train a segmentation algorithm, we'll need a special type of groundtruth data, where the labels for each image are the semantically segmented version of the image.

There are many approaches to semantic segmentation, which we can see the in the following bullets:

- The easiest way to do this is using the familiar sliding-window technique, which we described in the *Approaches to object detection* section. That is, we'll use a regular classifier and we'll slide it in either direction with stride 1. After we get the prediction for a location, we'll take the pixel that lies in the middle of the input region and we'll assign it with the predicted class. Predictably, this approach is very slow, due to the large number of pixels in an image (even a 1024 x 1024 image has more than 1,000,000 pixels).
- We can use a special type of CNN, called **Fully Convolutional Network** (**FCN**), to classify all pixels in the input region in a single pass. We can separate an FCN into two virtual components (in reality, this is just a single network):
    - The encoder is the first part of the network. It is such as a regular CNN, without the fully-connected layers at the end. The role of the encoder is to learn highly abstract representations of the input image (nothing new here).

- The decoder is the second part of the network. It starts after the encoder and uses it as input. The role of the decoder is to "translate" these abstract representations into the segmented groundtruth data. To do this, the decoder uses the opposite of the encoder operations. This includes unpooling (the opposite of pooling) and deconvolutions (the opposite of convolutions). We'll talk more about this concept (but in different context) in `Chapter 6`, *Generating images with GANs and VAEs*.

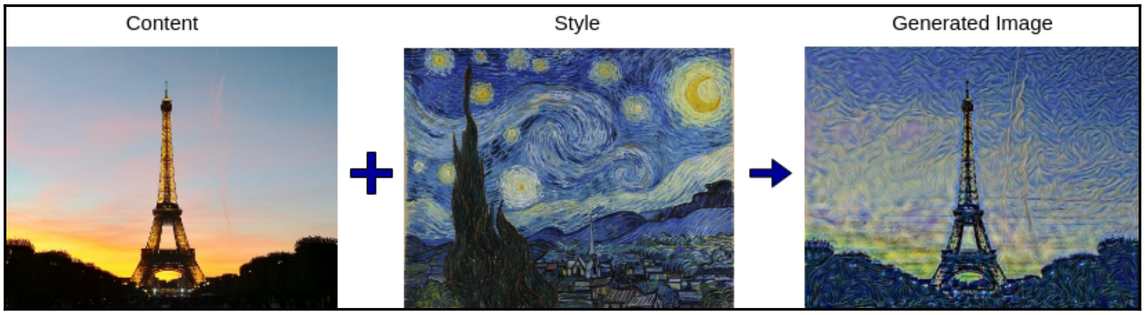
Artistic style transfer

Artistic style transfer is the use of the style (or texture) of one image to reproduce the semantic content of another. It can be implemented with different algorithms, but the most popular way was introduced in 2015 in the paper *A Neural Algorithm of Artistic Style* (`https://arxiv.org/abs/1508.06576`) by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. It's also known as neural style transfer and it uses (you guessed it!) CNNs. The basic algorithm has been improved and tweaked over the past few years, but in this section we'll look at the way it was first introduced, because it will give us a good foundation for understanding the latest versions.

The algorithm takes two images as input:

- Content image (*C*) we would like to redraw
- Style image (I) whose style (texture) we'll use to redraw *C*

The result of the algorithm is a new image: *G* = *C* + *S*. Here is an example of artistic style transfer:



An example of neural style transfer

To understand how neural style transfer works, let's recall that CNNs learn a hierarchical representation of the features. We know that the initial convolutional layers learn basic features, such as edges and lines. Conversely, the deeper layers learn more complex features, such as faces, cars, and trees. This is best visible in the diagrams in the *What is deep learning?* section of `Chapter 3`, *Deep Learning Fundamentals*. Knowing this, let's start with the following steps:

1. The authors propose we use a regular pre-trained VGG network. Next comes the interesting part.
2. Feed the network with the content image, *C*. Extract and store the output activations (or feature maps or slices) of one more of the hidden layers in the middle of the network. Let's denote these activations with A_c^l, where *l* is the index of the layer. We're interested in middle layers, because the level of feature abstraction encoded in them is best suited for the task.
3. Do the same with the style image, *S*. This time, denote the style activations of the *l* layer with A_s^l. The layers we choose for the content and style are not necessarily the same.
4. Generate a single random image (white noise), *G*. This random image will gradually turn into the end result of the algorithm. We'll repeat for a number of iterations:
    1. Propagate *G* through the network. This is the only image we'll use throughout the whole process. Such as before, we'll store the activations for all the *l* layers (here, *l* is the combination of all layers we used for the content and style images). Let's denote these activations with A_g^l.
    2. Compute the difference between the random noise activations, A_g^l, on one hand, and A_c^l and A_s^l on the other. These will be the two components of our loss function:
        - $J_c(C,G) = \frac{1}{2}\sum_l ||A_c^l - A_g^l||^2$, known as **content loss**: This is just the mean-square error over the element-wise difference between the two activations of all *l* layers.
        - $J_s(S,G)$, known as **style loss**: It's similar to the content loss, but instead of raw activations, we'll compare their **gram matrices** (we won't go into detail about that).

3. Use the content and style losses to compute the total loss, $J(G) = \alpha J_C(C, G) + \beta J_s(S, G)$, which is just a weighted sum of the two. The α and β coefficients determine which of the components will carry more weight.
4. Backpropagate the gradients to the start of the network and update the generated image, $G \leftarrow G - \frac{d}{dG}J(G)$. In this way, we make *G* more similar to both the content and style images, since the loss function is a combination of both.

This algorithm makes it possible to harness the powerful representational power of convolutional networks for artistic style transfer. It does this with a novel loss function and a smart use of backpropagation.

If you are interested in implementing neural style transfer, check out the official PyTorch tutorial at `https://pytorch.org/tutorials/advanced/neural_style_tutorial.html`.

One shortcoming of this algorithm is that it's relatively slow. Typically, we have to repeat this pseudo-training procedure for a couple hundred iterations to produce a visually-appealing result. Fortunately, the paper *Perceptual Losses for Real-Time Style Transfer and Super-Resolution* (`https://arxiv.org/abs/1603.08155`) by Justin Johnson, Alexandre Alahi, and Li Fei-Fei, builds on top of the original algorithm to provide a solution, which is three orders of magnitude faster.

Summary

In this chapter, we introduced some new and advanced computer vision techniques. We started with transfer learning, which is a way to bootstrap network training by using pre-trained models. Next, we discussed some of the popular neural network architectures in use today. Then, we talked about capsule networks, which are a promising new approach to computer vision. After that, we moved on to tasks beyond objects classification, such as object detection and semantic segmentation. And finally, we introduced neural style transfer.

In the next chapter, we'll explore a new type of ML algorithms, called generative models. We can use them to generate new content, such as images. Stay tuned, it will be fun!

6
Generating Images with GANs and VAEs

"What I cannot create, I do not understand."- Richard Feynman

This quote is often cited in the same sentence as generative models, and for good reason. In the previous two chapters (`Chapter 4`, *Computer Vision with Convolutional Networks* and `Chapter 5`, *Advanced Computer Vision*), we focused on supervised computer vision problems, such as classification and object detection. Now, we'll discuss how to create new images with the help of unsupervised neural networks. After all, it's a lot better knowing that you don't need labeled data. More specifically, we'll talk about generative models.

This chapter will cover the following topics:

- Intuition and justification of generative models
- Variational autoencoders
- Generative Adversarial networks

Intuition and justification of generative models

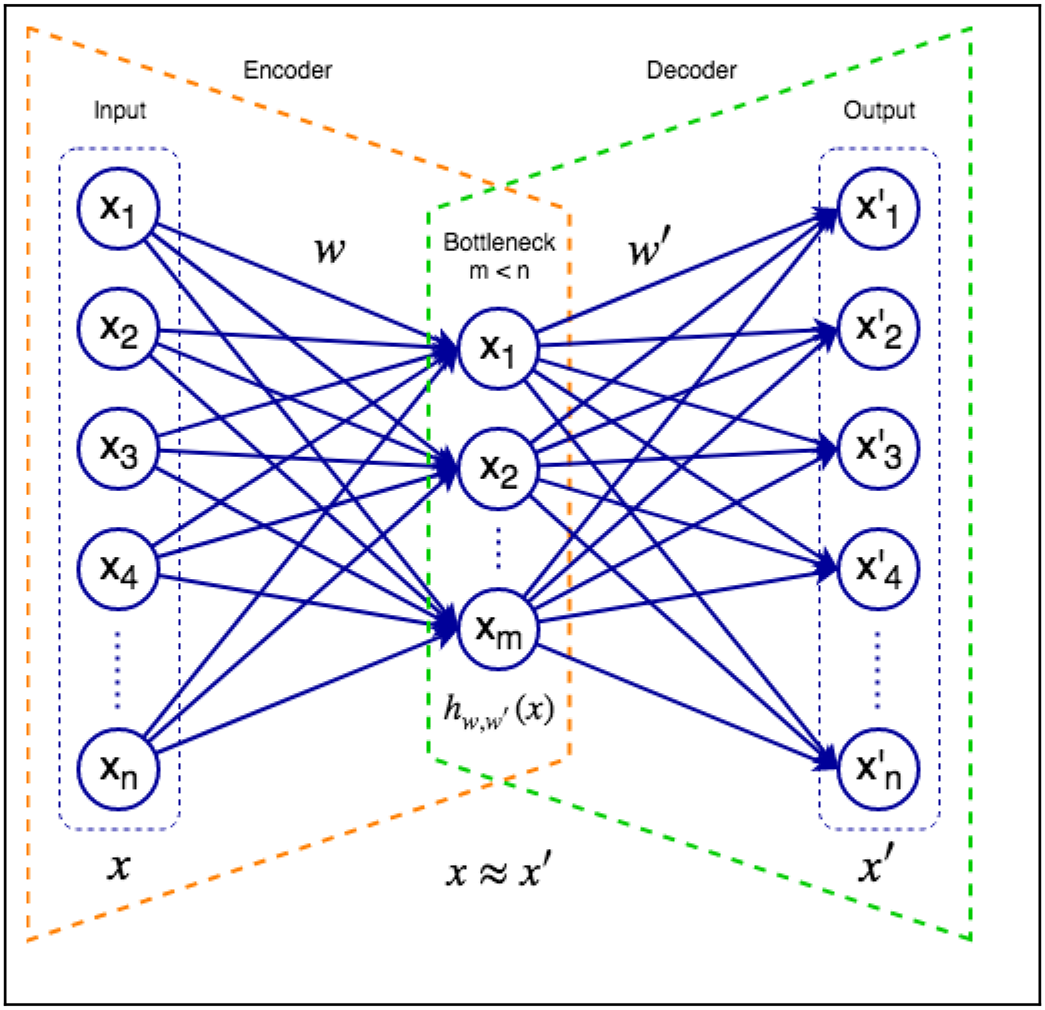
So far, we've used neural networks as **discriminative models**. This simply means that given input data, a discriminative model will map it to a certain label (in other words, a classification). A typical example is the classification of MNIST images in 1 of 10 digit classes, where the neural network maps the input data features (pixel intensities) to the digit label. We can also say this in another way, a discriminative model gives us the probability of y (class), given x (input) $P(Y|X = x)$. In the MNIST case, this is the probability of the digit, given the pixel intensities of the image.

On the other hand, a generative model learns the distribution of the classes. You can think of it as the opposite of what the discriminative model does. Instead of predicting the class probability, y, given certain input features, it tries to predict the probability of the input features, given a class, y- $P(X|Y = y)$. For example, a generative model will be able to create an image of a handwritten digit, given the digit class. Since we only have 10 classes, it will be able to generate just 10 images. But we used this example just to better illustrate the concept. In reality, the y "class" could be an arbitrary tensor of values, and the model would be able to generate an unlimited number of images with different features. If you don't understand this now, don't worry, we'll see many examples throughout the chapter.

Two of the most popular ways to use neural networks in a generative way are **variational autoencoders**(**VAEs**) and **Generative Adversarial networks**(**GANs**).

Variational autoencoders

To understand VAEs, let's talk about regular autoencoders first. An autoencoder is a feed-forward neural network that tries to reproduce its input. In other words, the target value (label) of an autoencoder is equal to the input data, $y^i = x^i$, where i is the sample index.We can formally say that it tries to learn an identity function, $h_{w,w'}(x) = x$ (a function that repeats its input). Since our "labels" are just the input data, the autoencoder is an unsupervised algorithm. The following diagram represents an autoencoder:



An autoencoder

An autoencoder consists of an input, hidden (or bottleneck), and output layers. Although it's a single network, we can think of it as a virtual composition of two components:

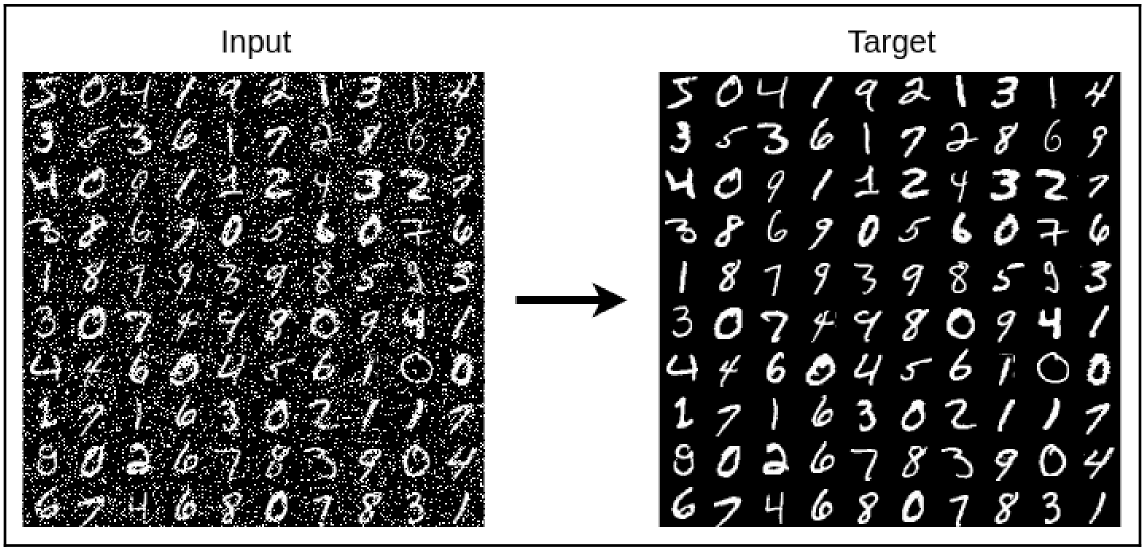
- **Encoder**: Maps the input data to the network's internal representation. For the sake of simplicity, in this example the encoder is a single, fully-connected hidden bottleneck layer. The internal state is just its activation vector. In general, the encoder can have multiple hidden layers, including convolutional.
- **Decoder**: Tries to reconstruct the input from the network's internal data representation. The decoder can also have a complex structure, which typically mirrors the encoder.

We can train the autoencoder by minimizing a loss function, which is known as the **reconstruction error** $\mathcal{L} = (x, x')$. It measures the distance between the original input and its reconstruction. We can minimize it in the usual way with gradient descent and backpropagation. Depending on the approach, we can use either mean square error (MSE) or binary cross-entropy (such as cross-entropy, but with two classes) as reconstruction errors. We first introduced MSE in `Chapter 1`, *Machine Learning: an introduction* and the cross-entropy loss in `Chapter 3`, *Deep Learning Fundamentals*.

At this point, you might wonder what the point of the autoencoder is, since it just repeats its input. However, we are not interested in the network output, but in its internal data representation (which is also known as representation in the **latent space**). The latent space contains hidden data features, which are not directly observed, but are inferred by the algorithm instead. The key is that the bottleneck layer has fewer neurons than the input/output ones. There are two main reasons for this:

- Because the network tries to reconstruct its input from a smaller feature space, it learns a compact representation of the data. You can think of it as a compression (but not lossless).

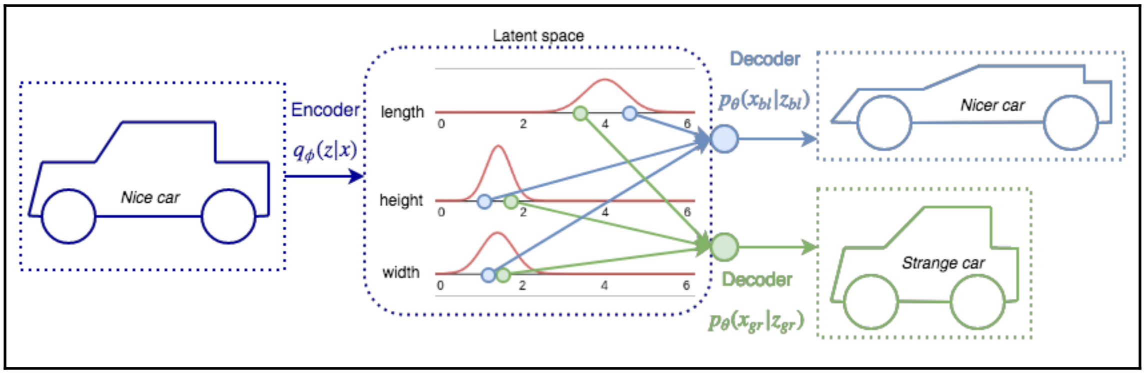
- By using fewer neurons, the network is forced to learn only the most important features of the data. To illustrate this concept, let's look at denoising autoencoders, where we intentionally use corrupted input data, but non-corrupted target data during training. For example, if we train a denoising autoencoder to reconstruct MNIST images, we can introduce noise by setting max intensity (white) to random pixels of the image (the following screenshot). To minimize the loss with the noiseless target, the autoencoder is forced to look beyond the noise in the input and learn only the important features of the data. However, if the network had more hidden neurons than input, it could overfit on the noise. With the additional constraint of fewer hidden neurons, it has nowhere to go but to try to ignore the noise. Once trained, we can use a denoising autoencoder to remove the noise from real images:



Denoising autoencoder input and target

The encoder maps each input sample to the latent space and each attribute of the latent representation has a discrete value. That means that an input sample can have only one latent representation. Therefore, the decoder can reconstruct the input in only one possible way. In other words, we can generate a single reconstruction of one input sample. But we don't want this. Instead, we want to generate new images that are different from the original. Enter VAEs.

A VAE can describe the latent representation in probabilistic terms. That is, instead of discrete values, we'll have a probability distribution for each latent attribute, making the latent space continuous. This makes it easier for random sampling and interpolation. Let's illustrate this with an example. Imagine that we try to encode an image of a vehicle and our latent representation has *n* attributes (*n* neurons in the bottleneck layer). Each attribute represents one vehicle property, such as length, height, and width (the following diagram).Say that the average vehicle length is four meters. Instead of the fixed value, the VAE can decode this property as a normal distribution with a mean of 4 (the same applies for the others). Then, the decoder can choose to sample a latent variable from the range of its distribution. For example, it can reconstruct a longer and lower vehicle, compared to the input. In this way, the VAE can generate an unlimited number of modified versions of the input:



An example of a variational encoder, sampling different values from the distribution ranges of the latent variables

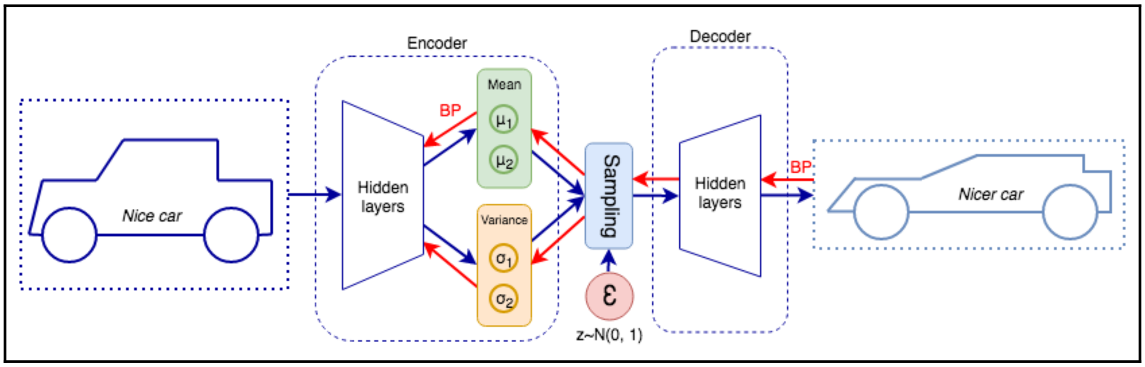
Let's formalize this:

- We'll denote the encoder with $q_\phi(z|x)$, where ϕ are the weights and biases of the network, x is the input, and$z$is the latent space representation. The encoder output is a distribution (for example, Gaussian) over the possible values of z, which could have generated x.
- We'll denote the decoder with $p_\theta(x|z)$, where θ are the decoder weights and biases. First, z is sampled stochastically (randomly) from the distribution. Then, it's sent through the decoder, whose output is a distribution over the possible corresponding values of x.
- The VAE uses a special type of loss function with two terms:

$$L(\theta, \varphi; x) = -D_{KL}(q_\varphi(z|x)||p_\theta(z)) + E_{q_\varphi(z|x)}[\log(p_\theta(x|z))]$$

The first is the Kullback-Leibler divergence between the probability distribution $q_\phi(z|x)$ and the expected probability distribution, $p(z)$. It measures how much information is lost, when we use $q_\phi(z|x)$ to represent $p(z)$ (in other words, how close the two distributions are). It encourages the autoencoder to explore different reconstructions. The second is the reconstruction loss, which measures the difference between the original input and its reconstruction. The more they differ, the more it increases. Therefore, it encourages the autoencoder to better reconstruct the data.

To implement this, the bottleneck layer won't directly output the latent state variables. Instead, it will output two vectors, which describe the **mean** and **variance** of the distribution of each latent variable:



Variational encoder sampling

Once we have the mean and variance distributions, we can sample a state, z , from the latent variable distributions and pass it through the decoder for reconstruction. But we cannot celebrate yet, because this presents us with another problem: backpropagation doesn't work over random processes such as the one we have here. Fortunately, we can solve this with the so-called **reparameterization trick**. First, we'll sample a random vector, ε, with the same dimensions as z from a Gaussian distribution (theε circle in the preceding figure). Then, we'll shift it by the latent distribution's mean, μ, and scale it by the latent distribution's variance, σ:

$$z = \mu + \sigma \odot \varepsilon$$

In this way, we'll be able to only optimize the mean and variance (red arrows) and we'll omit the random generator from the backward pass. At the same time, the sampled data will have the properties of the original distribution.

Generating new MNIST digits with VAE

In this section, we'll see how a VAE can generate new digits for the MNIST dataset and we'll use **Keras** to do so. We chose MNIST because it will illustrate the generative capabilities of the VAE well. Let's start:

1. Do the imports:

```
import matplotlib.pyplot as plt
from matplotlib.markers import MarkerStyle
import numpy as np
from keras import backend as K
from keras.datasets import mnist
from keras.layers import Lambda, Input, Dense
from keras.losses import binary_crossentropy
from keras.models import Model
```

2. Instantiate the MNIST dataset (we've already done that):

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

image_size = x_train.shape[1] * x_train.shape[1]
x_train = np.reshape(x_train, [-1, image_size])
x_test = np.reshape(x_test, [-1, image_size])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
```

3. Implement the `build_vae` function, which will build the VAE:
    - We'll have separate access to the encoder, decoder, and the full network. The function will return them as a tuple.
    - The bottleneck layer will have only `2` neurons(that is, we'll have only `2` latent variables). In this way, we'll be able to display the latent distribution as a 2D plot.
    - The encoder/decoder will contain a single intermediate (hidden) fully-connected layer with `512` neurons. This is not a convolutional network.
    - We'll use cross-entropy reconstruction loss and KL divergence.

The following is the implementation:

```
def build_vae(intermediate_dim=512, latent_dim=2):
    """
    Build VAE
    :param intermediate_dim: size of hidden layers of the
encoder/decoder
    :param latent_dim: latent space size
    :returns tuple: the encoder, the decoder, and the full vae
    """

    # encoder first
    inputs = Input(shape=(image_size,), name='encoder_input')
    x = Dense(intermediate_dim, activation='relu')(inputs)

    # latent mean and variance
    z_mean = Dense(latent_dim, name='z_mean')(x)
    z_log_var = Dense(latent_dim, name='z_log_var')(x)

    # reparameterization trick for random sampling
    # Note the use of the Lambda layer
    # At runtime, it will call the sampling function
    z = Lambda(sampling, output_shape=(latent_dim,),
name='z')([z_mean, z_log_var])

    # full encoder encoder model
    encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
    encoder.summary()

    # decoder
    latent_inputs = Input(shape=(latent_dim,), name='z_sampling')
    x = Dense(intermediate_dim, activation='relu')(latent_inputs)
    outputs = Dense(image_size, activation='sigmoid')(x)

    # full decoder model
    decoder = Model(latent_inputs, outputs, name='decoder')
    decoder.summary()

    # VAE model
    outputs = decoder(encoder(inputs)[2])
    vae = Model(inputs, outputs, name='vae')

    # Loss function
    # we start with the reconstruction loss
    reconstruction_loss = binary_crossentropy(inputs, outputs) *
image_size

    # next is the KL divergence
    kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
    kl_loss = K.sum(kl_loss, axis=-1)
    kl_loss *= -0.5

    # we combine them in a total loss
    vae_loss = K.mean(reconstruction_loss + kl_loss)
    vae.add_loss(vae_loss)

    return encoder, decoder, vae
```

4. Immediately tied to the network definition is the `sampling` function, which implements the random sampling of latent vectors,`z`, using the reparameterization trick (introduced in section *Variational autoencoders*):

```
def sampling(args: tuple):
    """
    Reparameterization trick by sampling z from unit Gaussian
    :param args: (tensor, tensor) mean and log of variance of
q(z|x)
    :returns tensor: sampled latent vector z
    """

    # unpack the input tuple
    z_mean, z_log_var = args

    # mini-batch size
    mb_size = K.shape(z_mean)[0]

    # latent space size
    dim = K.int_shape(z_mean)[1]

    # random normal vector with mean=0 and std=1.0
    epsilon = K.random_normal(shape=(mb_size, dim))

    return z_mean + K.exp(0.5 * z_log_var) * epsilon
```

5. Implement the `plot_latent_distribution` function. It collects the latent representations of all images in the test set and displays them over a 2D plot. We can do this because our network has only two latent variables (for the two axes of the plot). Note that to implement it, we only need the decoder:

```
def plot_latent_distribution(encoder,
                             x_test,
                             y_test,
                             batch_size=128):
    """
    Display a 2D plot of the digit classes in the latent space.
    We are interested only in z, so we only need the encoder here.
    :param encoder: the encoder network
    :param x_test: test images
    :param y_test: test labels
    :param batch_size: size of the mini-batch
    """
    z_mean, _, _ = encoder.predict(x_test, batch_size=batch_size)
    plt.figure(figsize=(6, 6))

    markers = ('o', 'x', '^', '<', '>', '*', 'h', 'H', 'D', 'd',
'P', 'X', '8', 's', 'p')

    for i in np.unique(y_test):
        plt.scatter(z_mean[y_test == i, 0], z_mean[y_test == i, 1],
                    marker=MarkerStyle(markers[i],
fillstyle='none'),
                    edgecolors='black')

    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.show()
```

6. Implement the `plot_generated_images` function.It will sample `n*n` vectors `z` in a `[-4, 4]` range for each of the two latent variables. Next, it will generate images based on the sampled vectors and it will display them in a 2D grid. Note that to do this, we only need the decoder:

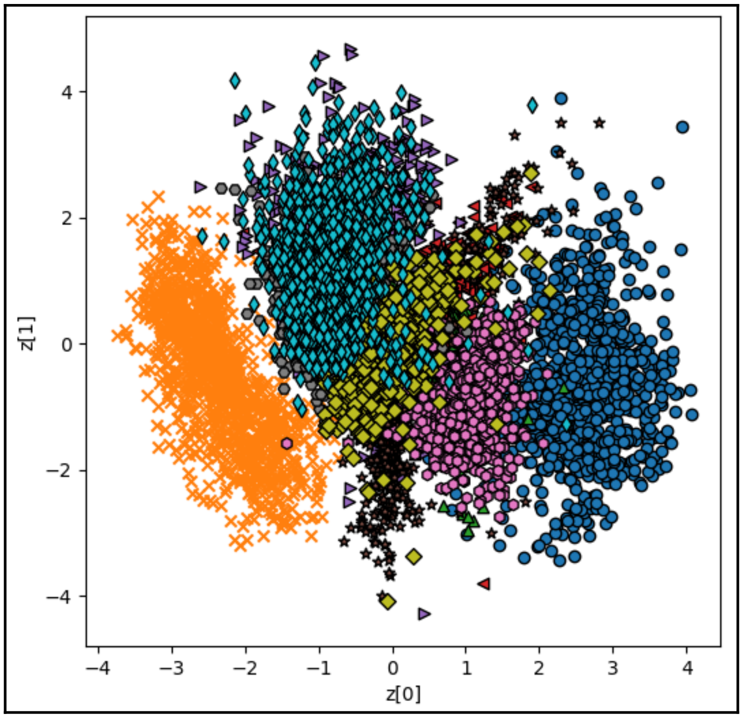
```
def plot_generated_images(decoder):
    """
    Display a 2D plot of the generated images.
    We only need the decoder, because we'll manually sample the
distribution z
    :param decoder: the decoder network
    """

    # display a nxn 2D manifold of digits
    n = 15
    digit_size = 28

    figure = np.zeros((digit_size * n, digit_size * n))
    # linearly spaced coordinates corresponding to the 2D plot
    # of digit classes in the latent space
    grid_x = np.linspace(-4, 4, n)
    grid_y = np.linspace(-4, 4, n)[::-1]

    # start sampling z1 and z2 in the ranges grid_x and grid_y
    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            z_sample = np.array([[xi, yi]])
            x_decoded = decoder.predict(z_sample)
            digit = x_decoded[0].reshape(digit_size, digit_size)
            slice_i = slice(i * digit_size, (i + 1) * digit_size)
            slice_j = slice(j * digit_size, (j + 1) * digit_size)
            figure[slice_i, slice_j] = digit

    # plot the results
    plt.figure(figsize=(6, 5))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range + 1
    pixel_range = np.arange(start_range, end_range, digit_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.imshow(figure, cmap='Greys_r')
    plt.show()
```

7. Run the whole thing. We'll use the Adam optimizer (introduced in `Chapter 3`,*Deep Learning fundamentals*)to train the network for 50 epochs:

```
if __name__ == '__main__':
 encoder, decoder, vae = build_vae()

 vae.compile(optimizer='adam')
 vae.summary()

 vae.fit(x_train,
 epochs=50,
 batch_size=128,
 validation_data=(x_test, None))

 plot_latent_distribution(encoder,
 x_test,
 y_test,
 batch_size=128)

 plot_generated_images(decoder)
```

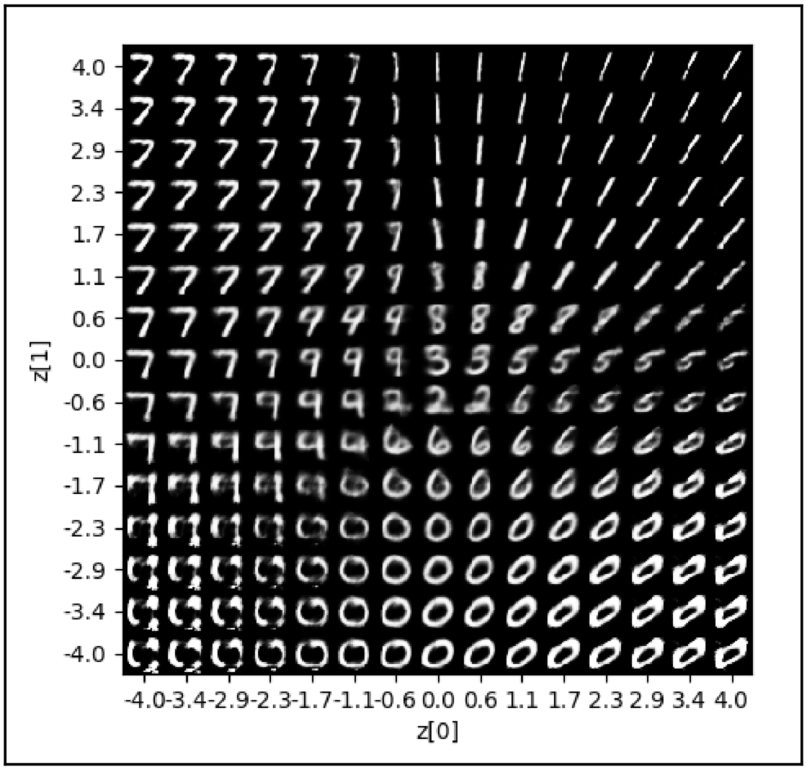
If everything goes to plan, once the training is over, we'll see the latent distribution for each digit class for all test images.

The left and bottom axes represent the `z₁` and `z₂` latent variables. Different marker shapes represent different digit classes:



The latent distributions of the MNIST test images

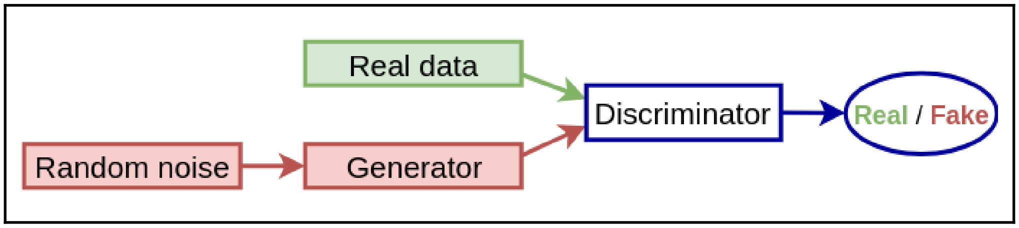
Next, we'll see the images, generated by `plot_generated_images`. The axes represent the particular latent distribution, `z`, used for each image:



Images generated by the VAE

Generative Adversarial networks

In this section, we'll talk about arguably the most popular generative model today: the GANs framework. It was first introduced in 2014 in the landmark paper *Generative Adversarial Nets*(`http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`) by Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair Aaron Courville, and Yoshua Bengio. The GANs framework can work with any type of data, but it's most popular application by far is to generate images, and we'll discuss them in this context only. Let's see how it work:



A GAN system

A GAN is a system of two components (neural networks):

- **Generator**: This is the generative model itself. It takes a probability distribution (random noise) as input and tries to generate a realistic output image. Its purpose is similar to the decoder part of the VAE.
- **Discriminator**: This takes two alternating inputs: the real images of the training dataset or the generated fake samples from the generator. It tries to determine whether the input image comes from the real images or the generated ones.

The two networks are trained together as a system. On the one hand, the discriminator tries to get better at distinguishing between the real and fake images. On the other hand, the generator tries to output more realistic images, so it could "deceive" the discriminator into thinking that the generated image is real. To use the analogy in the original paper, you can think of the generator as a team of counterfeiters, trying to produce fake currency. Conversely, the discriminator acts as a police officer, trying to capture the fake money, and the two are constantly trying to deceive each other (hence the name adversarial). The ultimate goal of the system is to make the generator so good that the discriminator wouldn't be able to distinguish between the real and fake images. Even though the discriminator does classification, a GAN is still unsupervised, since we don't need labels for the images.

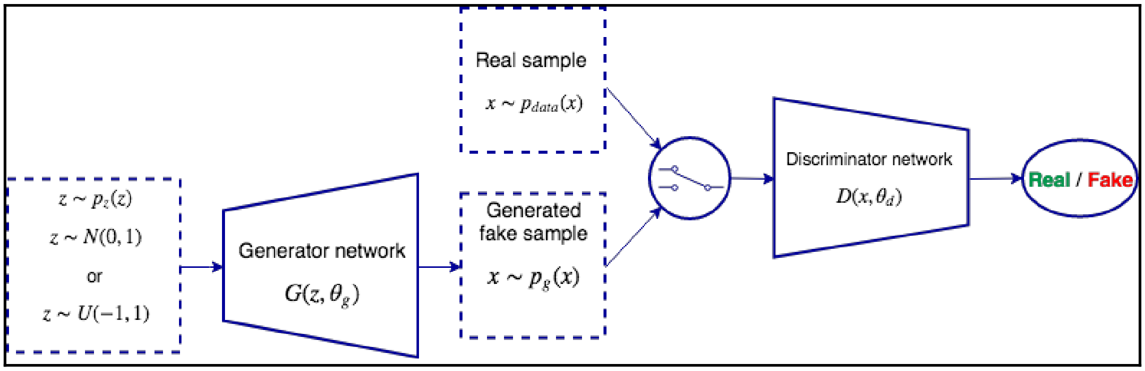
Training GANs

Our main goal is for the generator to produce realistic images and the GAN framework is a vehicle for that goal. We'll train the generator and the discriminator separately and sequentially (one after the other), and alternate between the two phases multiple times.

Before going into more detail, let's use the following figure to introduce some notations:

- We'll denote the generator with $G(z, \theta_g)$, where θ_g are the network weights, and z is the latent vector, which serves as an input to the generator. Think of it as a random seed value to kickstart the image-generation process. It is similar to the latent vector in the VAEs. z has a probability distribution, $p_z(z)$, which is usually random normal or random uniform. The generator outputs fake samples, x ,with a probability distribution of $p_g(x)$. You can think of $p_g(x)$ as the probability distribution of the real data according to the generator.
- We'll denote the discriminator with $D(x, \theta_d)$, where θ_d are the network weights. It takes as input either the real data with the $x \sim p_{data}(x)$ distribution, or the generated samples, $x \sim p_g(x)$. The discriminator is a binary classifier, which outputs whether the input image is part of the real (network output 1) or the generated data (network output 0).
- During training, we'll denote the discriminator and generator loss functions with $J^{(D)}$ and $J^{(G)}$, respectively.

Here is a more detailed diagram of a GAN framework:



Detailed example of a Generative Adversarial network

GAN training is different compared to the training of a regular DNN, because we have two networks. We can think of it as a sequential minimax zero-sum game of two players (generator and discriminator):

- **Sequential**: Means that the players take turns after one another, similar to chess or tic-tac-toe (as opposed to simultaneous). First, the discriminator tries to minimize $J^{(D)}$, but it can only do so by adjusting the weights, θ_d. Next, the generator tries to minimize $J^{(G)}$, but it can only adjust the weights, θ_g. We repeat this process multiple times.
- **Zero-sum**: Means that the gains or losses of one player are exactly balanced by the gains or losses of the opposite player. That is, the sum of the generator's loss and the discriminator's loss is always 0:

$$J^{(G)} = -J^{(D)}$$

- **Minimax**: Means that the strategy of the first player (generator) is to **minimize** the opponent's (discriminator) **maximum** score (hence the name). When we train the discriminator, it becomes better at distinguishing between real and fake samples (minimizing $J^{(D)}$). Next, when we train the generator, it tries to step up to the level of the newly-improved discriminator (we minimize $J^{(G)}$, which is equivalent to maximizing $J^{(D)}$). The two networks are in constant competition. We'll denote the minimax game by the following, where V is the cost function:

$$\min_G \max_D V(G, D)$$

Let's assume that after a number of training steps, both $J^{(G)}$ and $J^{(D)}$ will be at some local minimum. Then, the solution to the minimax game is called the Nash equilibrium. A Nash equilibrium happens when one of the actors doesn't change its action, regardless of what the other actor may do. A Nash equilibrium in a GAN framework happens when the generator becomes so good that the discriminator is no longer able to distinguish between the generated and real samples. That is, the discriminator output will always be $\frac{1}{2}$ regardless of the presented input.

Training the discriminator

The discriminator is a classification neural network and we can train it in the usual way, using gradient descent and backpropagation. However, the training set is composed of equal parts real and generated samples. Let's see how to incorporate that in the training process:

1. Depending on the input sample (real or fake), we have two paths:
    - Select the sample from the real data, $x \sim p_{data}$, and use it to produce $D(x)$.
    - Generate fake sample, $x \sim p_g$. Here, generator and discriminator work as a single network. We start with a random vector, z, which we use to produce the generated sample, $G(z)$. Then, we use it as input to the discriminator to produce the final output, $D(G(z))$.
2. Compute the loss function, which reflects the duality of the training data (more on that later).
3. Backpropagate the error gradient and update the weights. Although the two networks work together, the generator weights, θ_g, will be locked and we'll only update the discriminator weights, θ_d. This ensures that we'll improve the discriminator performance by making it better, as opposed to making the generator worse.

To understand the discriminator loss, let's recall the formula for the cross-entropy loss:

$$H(p,q) = -\sum_{i=1}^{n} p_i(x)\log(q_i(x))$$

Where $q_i(x)$ is the estimated probability of the output belonging to the *i* class (out of *n* total classes) and $p_i(x)$ is the actual probability. For the sake of simplicity, we'll assume that we apply the formula over a single training sample. In the case of binary classification, this formula can be simplified as follows:

$$H(p,q) = -(p(x)\log q(x) + (1-p(x))\log(1-q(x)))$$

TIP

In the case where the target probabilities are $p(x) \to \{0,1\}$ (one-hot-encoding), one of the loss terms is always 0.

We can expand the formula for a mini-batch of m samples:

$$H(p,q) = -\frac{1}{m}\sum_{j=1}^{m}(p(x_j)\log(q(x_j)) + (1-p(x_j))\log(1-q(x_j)))$$

Knowing all this, let's define the discriminator loss:

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}}\log(D(x)) - \frac{1}{2}\mathbb{E}_z\log(1-D(G(z)))$$

Although it seems complex, it is just a cross-entropy loss for a binary classifier with some GAN-specific bells and whistles. Let's discuss them:

- The two components of the loss reflect the two possible classes (real or fake), which are in equal number in the training set.
- $\frac{1}{2}\mathbb{E}_{x\sim p_{data}}\log D(x)$ is the loss when the input is sampled from the real data. Ideally, in such cases, we'll have $D(x)=1$.
- In this context, the term $\mathbb{E}_{x\sim p_{data}}$ (called **expectation**) implies that the x is sampled from p_{data}. In essence, this part of the loss means "when we sample x from p_{data}, we expect the discriminator output, $D(x)=1$". Finally, 0.5 is the cumulative class probability of the real data, $p(x)$, since it comprises exactly half of the whole set.
- $\frac{1}{2}\mathbb{E}_z\log(1-D(G(z)))$ is the loss, when the input is sampled from the generated data. Here, we can make the same observations as with the real data component. However,this term is maximized when $D(G(z))=0$.

To summarize, the discriminator loss will be zero when $D(x)=1$ for all $x\sim p_{data}$ and $D(x)=0$ for all generated $x\sim p_g$ (or $x=G(z)$).

Training the generator

We'll train the generator by making it better at deceiving the discriminator. To do this, we'll need both networks, similar to the way we train the discriminator with fake samples:

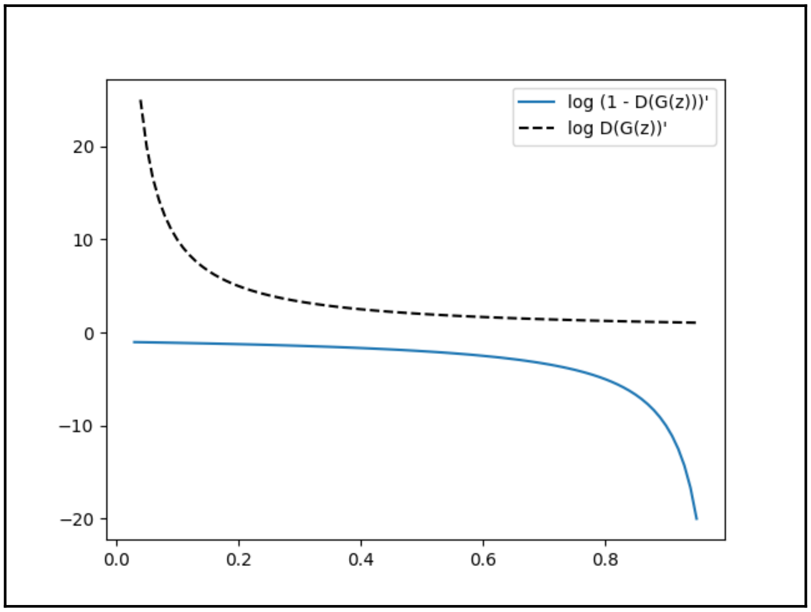
1. We start with a random latent vector, z, and feed it through both the generator and discriminator, to produce the output, $D(G(z))$.

2. The loss function is the same as the discriminator loss. However, our goal here is to maximize it, rather than minimize it, since we want to deceive the discriminator.
3. In the backward pass, the discriminator weights, θ_d, are locked and we can only adjust θ_g . This forces us to maximize the discriminator loss by making the generator better, instead of making the discriminator worse.

You may notice that in this phase we only use generated data. The part of the loss function that deals with real data will always be 0. Therefore,we can simplify it to the following:

$$J^{(G)} = \mathbb{E}_z \log(1 - D(G(z)))$$

The derivative (gradient) of this formula is $-\frac{1}{1-D(G(z))}$, displayed in the following figure with an uninterrupted line.This imposes a limitation on the training. Early on, when the discriminator can easily distinguish between real and fake samples, ($D(G(z)) \approx 0$), the gradient will be close to zero. This would result in little learning of the weights, θ_g (this problem is known as diminished gradient):



Gradients of the two generator loss functions

We can solve this issue by using a different loss function:

$$J^{(G)} = -\mathbb{E}_z \log(D(G(z))$$

The derivative of this function is displayed in the preceding figure with a dashed line. This loss is still minimized, when $D(G(z)) \approx 1$ and at the same time the gradient is large, when the generator underperforms. With this loss, the game is no longer zero-sum, but this won't have a practical effect on the GAN framework.

Putting it all together

With our newfound knowledge, we can define the minimax objective in full:

$$\min_G \max_D V(G, D) = \frac{1}{2}\mathbb{E}_{x \sim p_{data}} \log(D(x)) + \frac{1}{2}\mathbb{E}_z \log(1 - D(G(z)))$$

In short, the generator tries to minimize the objective, while the discriminator tries to maximize it. Note that while the discriminator should minimize its loss, the minimax objective is a negative of the discriminator loss, and therefore the discriminator has to maximize it.

The following is a step-by-step training algorithm, as it introduced by the authors of the GAN framework.

Repeat for a number of iterations:

1. Repeat for *k* steps, where *k* is a hyperparameter:
    - Sample a mini-batch of *m* random samples from the latent space, $\{z^{(1)}, z^{(2)}, \ldots z^{(m)}\} \sim p_g(z)$.
    - Sample a mini-batch of *m* samples from the real data, $\{x^{(1)}, x^{(2)}, \ldots x^{(m)}\} \sim p_{data}(x)$.
    - Update the discriminator weights, θ_d, by ascending the stochastic gradient of its loss:

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

2. Sample a mini-batch of *m* random samples from the latent space, $\{z^{(1)}, z^{(2)}, \ldots z^{(m)}\} \sim p_g(z)$.
3. Update the generator by descending the stochastic gradient of its loss:

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

At the end of this section, we'll mention that the gradient descent algorithm is designed to find the minimum of the loss function, rather than the Nash equilibrium, which is not the same thing. As a result, sometimes the training may fail to converge. But due to the popularity of GANs, many improvements have been proposed. If the reader is interested in training GANs, do your own research to learn more about them.

Types of GANs

Since the the GAN framework was first introduced, a lot of new variations have emerged. In fact, there are so many new GANs now that in order to stand out, some of the authors have come up with creative GAN names, such as BicycleGAN,DiscoGAN, GANs for LIFE, and ELEGANT.In this section, we'll discuss some of them.

DCGAN

In the original GAN framework proposal, the authors used only fully-connected networks. The first major improvement of the GAN framework is **Deep Convolutional Generative Adversarial networks** (**DCGANs**). In this new architecture, both the generator and the discriminator are convolutional networks. They have some constraints, which help to stabilize the training:

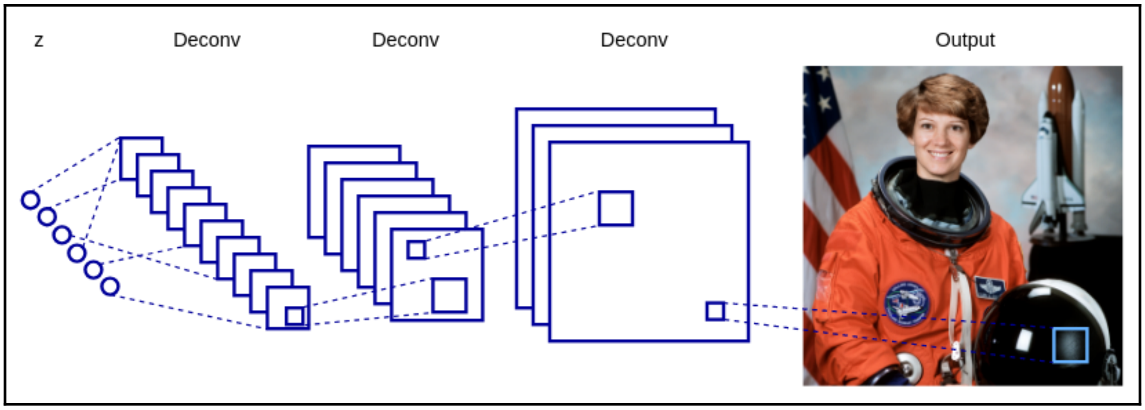
- The discriminator uses strided convolutions instead of pooling layers.
- The generator is a special type of CNN, which uses fractional-strided convolutions to increase the size of the images. We'll discuss it in the next section.
- Both networks use batch normalization.
- No fully-connected layers, with the exception of the last layer of the discriminator.

- LeakyReLU (`https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#Leaky_ReLUs`)activations for all layers of the generator, except the output, which uses Tanh (introduced in `Chapter 2`, *Neural networks*).
- LeakyReLU activations for all layers of the discriminator, except the output, which uses sigmoid.

You can think of these as general guidelines for GAN training and not just for DCGAN.

The generator in DCGAN

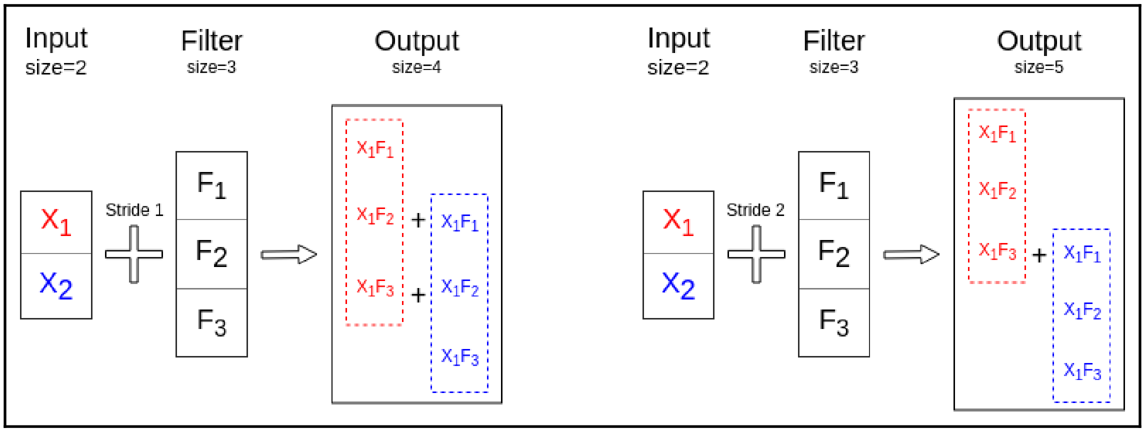
In the following diagram, we can see a sample generator network in the DCGAN framework:



Generator network with deconvolutional layers

As usual, the generator starts with a random latent vector, z . To transform it into an image, we'll use a network with a special type of convolution operation, called transposed convolution (also known as deconvolution or fractionally-strided convolution). We briefly touched on it in `Chapter 4`, *Computer Vision with Convolutional Networks,* in the *Backpropagation in convolutional layers* section, but let's discuss it in a little more detail now.You can think of the transposed convolution as an opposite of the regular convolution. As usual, we have input, output, and a filter with weights. But here, we'll apply the filter over a single input neuron to produce multiple outputs.

Let's illustrate this concept with a simple 1D transposed convolution example:



Transposed convolution with stride 1 (left) and stride 2 (right)

We multiply a neuron's output by each of the filter weights to produce a patch with the same dimensions as the filter. Then, we sum the overlapping regions of the patches to produce the final output. Note that the overlap is in the output layer, as opposed to regular convolution, where we had overlapping regions in the input.As a consequence, the stride is also relevant to the output layer. By setting the stride larger than 1, we can increase the output size, compared to the input. We can use this property of the transpose convolutions to gradually up-sample the latent vector, z , in the generator.

Let the size of the input slice be *I*, the size of the filter *F*, the stride *S*, and the input padding *P*. Then, the size,*O*,of the output slice is given by the following:

$$O = S(I-1) + F - 2P$$

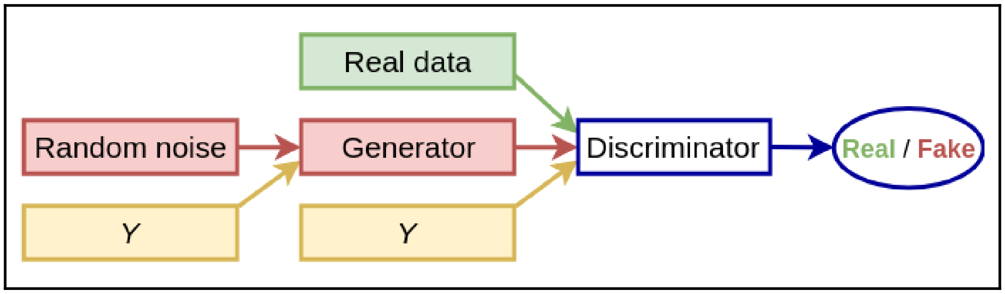
For example, the output size of the left example is `1*(2 - 1) + 3 - 2*0 = 4`. The output size of the right example is `2*(2 - 1) + 3 - 2*0 = 5`.

For more information about DCGANs, check out the original paper,*Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*(`https://arxiv.org/abs/1511.06434`), by Alec Radford, Luke Metz, and Soumith Chintala.

Conditional GANs

Conditional GANs (CGANs) are an extension of the GAN framework where both the generator and discriminator receive some additional conditioning input information,y. This could be the class of the current image or some other property.

For example, if we train a GAN to generate new MNIST images, we could add an additional input layer with values of one-hot-encoded image labels:



Conditional GAN

CGANs have the one disadvantage that are not strictly unsupervised and we need some kind of labels for them to work. However, they have some other advantages:

- By using more and better-structured information for training, the model can learn better data representations and generate better samples.
- In regular GANs, all the image information is stored in the latent vector,z. This poses a problem: since$z$can be complex, we don't have much control over the properties of the generated image. For example, suppose that we want our MNIST GAN to generate a certain digit, say 7. We would have to experiment with different latent vectors until we reach the desired output. But with CGAN, we could simply combine the one-hot vector of 7 with some random$z$and the network will generate the correct digit. We could still try different$z$values forand the model would generate different versions of the digit 7. In short, CGANs provide a way to control (condition) the generator output.

For more information about CGANs, check out the original paper,*Conditional Generative Adversarial Nets* (`https://arxiv.org/abs/1411.1784`), by Mehdi Mirza and Simon Osindero.

Generating new MNIST images with GANs and Keras

In this section, we'll demonstrate how to use GANs to generate new MNIST images with Keras. Let's start:

1. Do the imports:

```
import matplotlib.pyplot as plt
import numpy as np
from keras.datasets import mnist
from keras.layers import BatchNormalization, Input, Dense, Reshape,
Flatten
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Sequential, Model
from keras.optimizers import Adam
```

2. Implement the `build_generator` function. In this example, we'll use a simple fully-connected generator. However, we'll still follow the guidelines outlined in the *DCGAN* section:

```
def build_generator(latent_dim: int):
    """
    Build discriminator network
    :param latent_dim: latent vector size
    """

    model = Sequential([
        Dense(128, input_dim=latent_dim),
        LeakyReLU(alpha=0.2),
        BatchNormalization(momentum=0.8),
        Dense(256),
        LeakyReLU(alpha=0.2),
        BatchNormalization(momentum=0.8),
        Dense(512),
        LeakyReLU(alpha=0.2),
        BatchNormalization(momentum=0.8),
        Dense(np.prod((28, 28, 1)), activation='tanh'),
        # reshape to MNIST image size
        Reshape((28, 28, 1))
    ])

    model.summary()

    # the latent input vector z
    z = Input(shape=(latent_dim,))
```

```
        generated = model(z)

        # build model from the input and output
        return Model(z, generated)
```

3. Build the discriminator. Again, it's a simple, fully-connected network:

```
    def build_discriminator():
     """
     Build discriminator network
     """

     model = Sequential([
     Flatten(input_shape=(28, 28, 1)),
     Dense(256),
     LeakyReLU(alpha=0.2),
     Dense(128),
     LeakyReLU(alpha=0.2),
     Dense(1, activation='sigmoid'),
     ], name='discriminator')

     model.summary()

     image = Input(shape=(28, 28, 1))
     output = model(image)

     return Model(image, output)
```

4. Implement the `train` function with the actual GAN training. This function implements the procedure outlined in the *Training GANs,Putting it all together* section:

```
    def train(generator, discriminator, combined, steps, batch_size):
        """
        Train the GAN system
        :param generator: generator
        :param discriminator: discriminator
        :param combined: stacked generator and discriminator
        we'll use the combined network when we train the generator
        :param steps: number of alternating steps for training
        :param batch_size: size of the minibatch
        """

        # Load the dataset
        (x_train, _), _ = mnist.load_data()

        # Rescale in [-1, 1] interval
        x_train = (x_train.astype(np.float32) - 127.5) / 127.5
        x_train = np.expand_dims(x_train, axis=-1)

        # Discriminator ground truths
        real = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        latent_dim = generator.input_shape[1]

        for step in range(steps):
            # Train the discriminator

            # Select a random batch of images
            real_images = x_train[np.random.randint(0,
    x_train.shape[0], batch_size)]

            # Random batch of noise
            noise = np.random.normal(0, 1, (batch_size, latent_dim))

            # Generate a batch of new images
            generated_images = generator.predict(noise)

            # Train the discriminator
            discriminator_real_loss =
    discriminator.train_on_batch(real_images, real)
            discriminator_fake_loss =
    discriminator.train_on_batch(generated_images, fake)
            discriminator_loss = 0.5 * np.add(discriminator_real_loss,
    discriminator_fake_loss)

            # Train the generator
            # random latent vector z
            noise = np.random.normal(0, 1, (batch_size, latent_dim))

            # Train the generator
            # Note that we use the "valid" labels for the generated
    images
            # That's because we try to maximize the discriminator loss
            generator_loss = combined.train_on_batch(noise, real)

            # Display progress
            print("%d [Discriminator loss: %.4f%%, acc.: %.2f%%]
    [Generator loss: %.4f%%]" %
                  (step, discriminator_loss[0], 100 *
    discriminator_loss[1], generator_loss))
```

5. Implement a boilerplate function,`plot_generated_images`, to display some generated images after the training is finished:
    1. Create an `nxn` grid (the `figure` variable).
    2. Create `nxn` random latent vectors (the `noise` variable), one for each generated image.
    3. Generate the images and place them in the grid cells.
    4. Display the result.

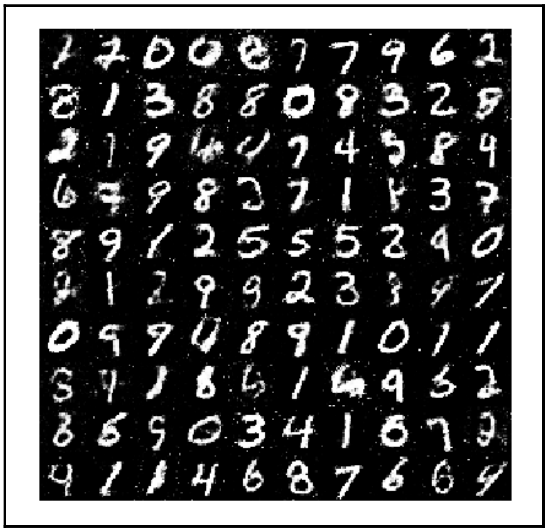
The following is the implementation:

```
def plot_generated_images(generator):
    """
    Display a nxn 2D manifold of digits
    :param generator: the generator
    """
    n = 10
    digit_size = 28

    # big array containing all images
    figure = np.zeros((digit_size * n, digit_size * n))

    latent_dim = generator.input_shape[1]

    # n*n random latent distributions
    noise = np.random.normal(0, 1, (n * n, latent_dim))

    # generate the images
    generated_images = generator.predict(noise)

    # fill the big array with images
    for i in range(n):
        for j in range(n):
            slice_i = slice(i * digit_size, (i + 1) * digit_size)
            slice_j = slice(j * digit_size, (j + 1) * digit_size)
            figure[slice_i, slice_j] =
np.reshape(generated_images[i * n + j], (28, 28))

    # plot the results
    plt.figure(figsize=(6, 5))
    plt.axis('off')
    plt.imshow(figure, cmap='Greys_r')
    plt.show()
```

6. Build the generator, discriminator, and the combined network. Run the training for 15,000 steps using the Adam optimizer, and plot the results once it's done:

```
if __name__ == '__main__':
    latent_dim = 64

    # Build and compile the discriminator
    discriminator = build_discriminator()
    discriminator.compile(loss='binary_crossentropy',
                          optimizer=Adam(lr=0.0002, beta_1=0.5),
                          metrics=['accuracy'])

    # Build the generator
    generator = build_generator(latent_dim)

    # Generator input z
    z = Input(shape=(latent_dim,))
    generated_image = generator(z)

    # Only train the generator for the combined model
    discriminator.trainable = False

    # The discriminator takes generated image as input and
determines validity
    real_or_fake = discriminator(generated_image)

    # Stack the generator and discriminator in a combined model
    # Trains the generator to deceive the discriminator
    combined = Model(z, real_or_fake)
    combined.compile(loss='binary_crossentropy',
                     optimizer=Adam(lr=0.0002, beta_1=0.5))

    # train the GAN system
    train(generator=generator,
          discriminator=discriminator,
          combined=combined,
          steps=15000,
          batch_size=128)

    # display some random generated images
    plot_generated_images(generator)
```

If everything goes as planned, we should see something similar to the following:


Newly-generated MNIST images

Summary

In this chapter, we discussed how to create new images with generative models, which is one of the most exciting machine learning areas at the moment. We talked about two of the most popular generative algorithms: VAEs and GANs. First, we learned their theoretical foundations and then we implemented simple programs to generate new MNIST digits with each algorithm.

This chapter concludes the series of the last three chapters, which were dedicated to computer vision. In the next chapter, we'll discuss how to apply DL algorithms in the field of **natural language processing** (**NLP**). We'll also introduce the main NLP paradigms and a new type of neural network, called the recurrent network, which is especially suited for NLP tasks.

7
Recurrent Neural Networks and Language Models

The neural network architectures we discussed in the previous chapters take in fixed sized input and provide fixed sized output. This chapter will lift this constraint by introducing **Recurrent Neural Networks** (**RNNs**). RNNs help us deal with sequences of variable length by defining a recurrence relation over these sequences (hence the name).

The ability to process arbitrary sequences of input makes RNNs applicable for **natural language processing** (**NLP**) and speech recognition tasks. In fact, RNNs can be applied to any problem since it has been proven that they are Turing complete – theoretically, they can simulate any program that a regular computer would not be able to compute. For example, Google's DeepMind has proposed a model called Differentiable Neural Computer, which can learn how to execute simple algorithms, such as sorting.

In this chapter, we will cover the following topics:

- Recurrent neural networks
- Language modeling
- Sequence to sequence learning
- Speech recognition

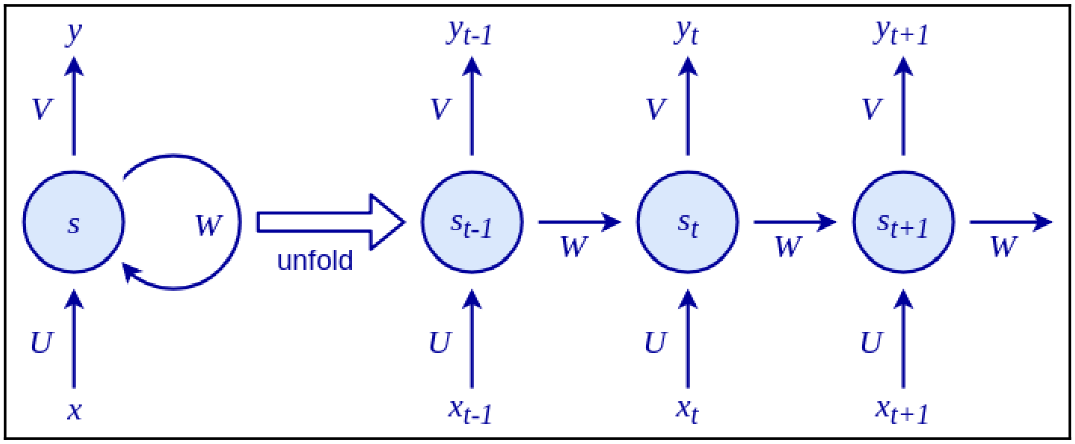
Recurrent neural networks

RNN is a type of neural network, which can process sequential data with variable length. Examples of such data include the words of a sentence or the price of a stock in various moments of time. By using the word sequential, we imply that the elements of the sequence are related to each other and their order matters. For example, if we take a book and shuffle randomly all the words in it, the text will loose it's meaning, even though we'll still know the individual words.

RNNs get their name because they apply the same function over a sequence recurrently. We can define an RNN as a recurrence relation:

$$s_t = f(s_{t-1}, x_t)$$

Here, f is a differentiable function, s_t is a vector of values called internal network state (at step t), and x_t is the network input at step t. Unlike regular networks, where the state only depends on the current input (and network weights), here s_t is a function of both the current input, as well as the previous state, s_{t-1}. You can think of s_{t-1} as the network's summary of all previous inputs. The recurrence relation defines how the state evolves step by step over the sequence via a feedback loop over previous states, as illustrated in the following diagram:



Left: Visual illustration of the RNN recurrence relation: $s_t = s_{t-1} * W + x_t * U$. The final output will be $y_t = s_t * V$. Right: RNN states recurrently unfolded over the sequence *t-1, t, t+1*. Note that the parameters *U, V, and W* are shared between all the steps

The RNN has three sets of parameters (or weights):

- *U* transforms the input x_t to the state s_t
- *W* transforms the previous state s_{t-1} to the current state s_t
- *V* maps the newly computed internal state s_t to the output y_t

U, *V*, and *W* apply linear transformation over their respective inputs. The most basic case of such a transformation is the familiar weighted sum we know and love. We can now define the internal state and the network output as follows:

$$s_t = f(s_{t-1} * W + x_t * U)$$

$$y_t = s_t * V$$

Here, *f* is the non-linear activation function (such as tanh, sigmoid, or ReLU).

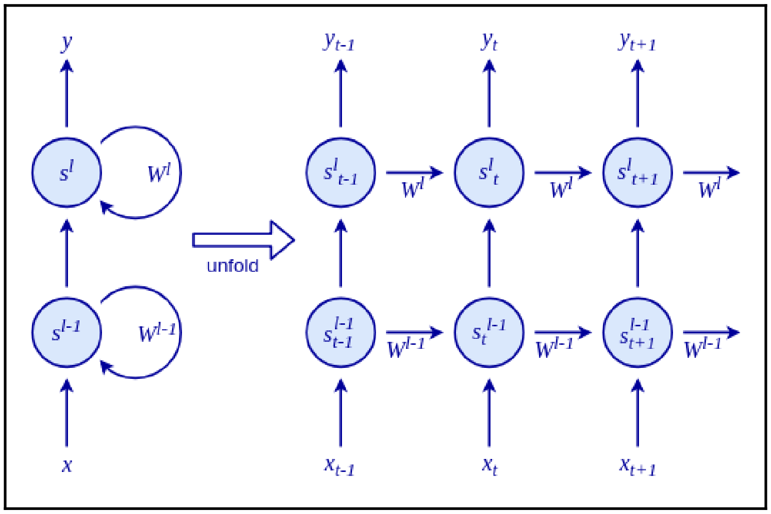
For example, in a word-level language model, the input *x* will be a sequence of words encoded in input vectors $(x_1 \ldots x_t \ldots)$. The state *s* will be a sequence of state vectors $(s_1 \ldots s_t \ldots)$. Finally, the output *y* will be a sequence of probability vectors $(y_1 \ldots y_t \ldots)$ of the next words in the sequence.

Note that in a RNN, each state is dependent on all previous computations via this recurrence relation. An important implication of this is that RNNs have memory over time, because the states *s* contains information based on the previous steps. In theory, RNNs can remember information for an arbitrarily long period of time, but in practice they are limited to looking back only a few steps. We will address this issue in more detail in the *Vanishing and exploding gradients* section.

The RNN we described is somewhat equivalent to a single layer regular neural network (with an additional recurrence relation). As we now know from `Chapter 2`, *Neural Networks*, a network with a single layer has some serious limitations. Fear not! As with regular networks, we can stack multiple RNNs to form a **stacked RNN**. The cell state s^l_t of a RNN cell at level *l* at time *t* will take the output y^{l-1}_t of the RNN cell from level *l-1* and previous cell state s^l_{t-1} of the cell at the same level *l* as the input:

$$s^l_t = f(s^l_{t-1}, y^{l-1}_t)$$

In the following diagram, we can see an unfolded, stacked RNN:

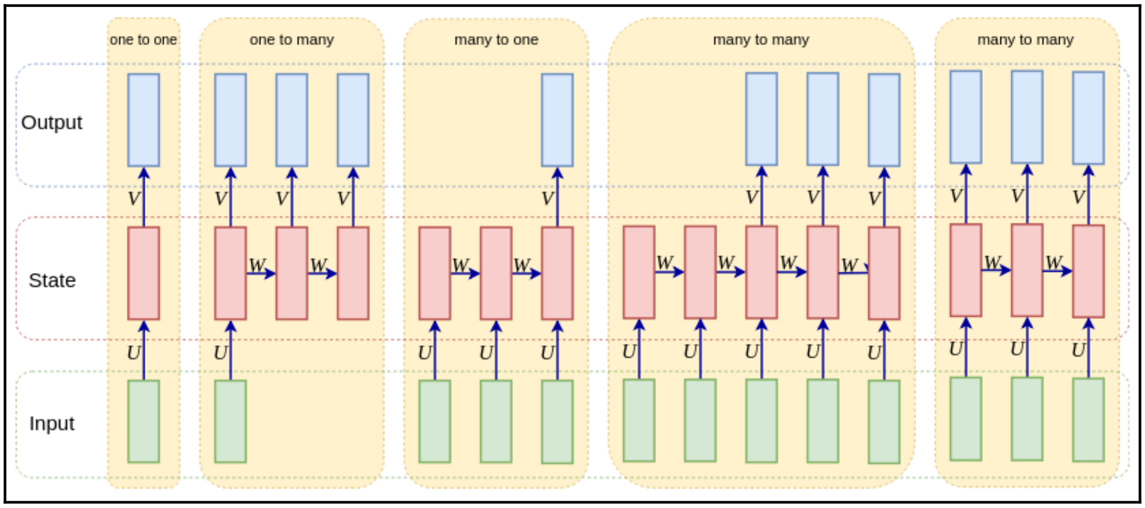


Stacked RNN

Because RNNs are not limited to processing fixed size inputs, they really expand the possibilities of what we can compute with neural networks, such as sequences of different lengths or images of varied sizes. The following are some combinations:

- **One-to-one**: This is non-sequential processing, such as feedforward neural networks and convolutional neural networks. Note that there isn't much difference between a feedforward network and applying an RNN to a single time step. An example of one-to-one processing is image classification, which we looked at in `Chapter 4`, *Computer Vision with Convolutional Networks*.
- **One-to-many**: This processing generates a sequence based on a single input, for example, caption generation from an image (`https://arxiv.org/abs/1411.4555v2`).
- **Many-to-one**: This processing outputs a single result based on a sequence, for example, sentiment classification from text.
- **Many-to-many indirect**: A sequence is encoded into a state vector, after which this state vector is decoded into a new sequence, for example, language translation (`https://arxiv.org/abs/1406.1078v3` and `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf`).
- **Many-to-many direct:** This outputs a result for each input step, for example, frame phoneme labeling in speech recognition (see the *Speech recognition* section for more details).

The following is a graphical representation of the preceding input-output combinations (idea from `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`):



RNN input-output combinations

RNN implementation and training

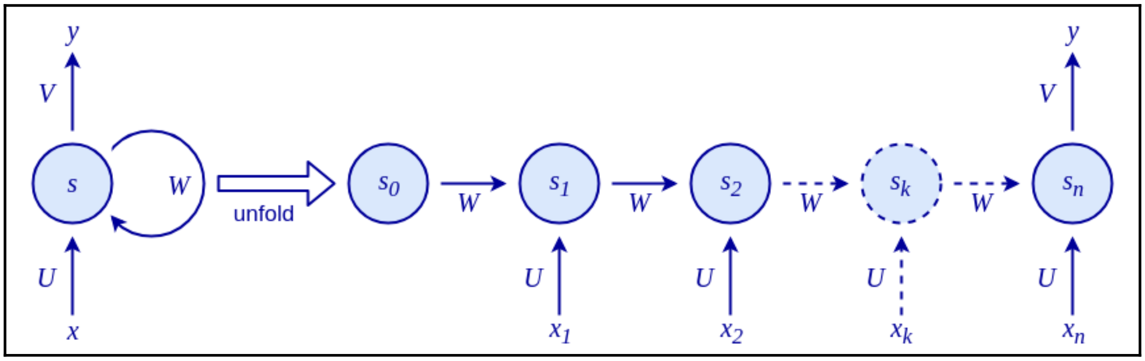
In the preceding section, *Recurrent neural networks*, we briefly discussed what RNNs are and what problems they can solve. Let's dive into the details of an RNN and how to train it with a very simple toy example: counting ones in a sequence.

In this problem, we will teach a basic RNN how to count the number of ones in the input, and then output the result at the end of the sequence. This is an example of a "many-to-one" relationship, which we defined in the previous section.

We'll implement this example with Python (no DL libraries) and NumPy. An example of input and output is as follows:

```
In: (0, 0, 0, 0, 1, 0, 1, 0, 1, 0) Out: 3
```

The RNN we'll use is illustrated in the following diagram:



Basic RNN for counting ones in the input

The network will have only two parameters: an input weight *U* and a recurrence weight *W*. The output weight *V* is set to 1 so that we just read out the last state as the output *y*.

Before we continue, let's add some code so that our example can be executable. We'll import `numpy` and we'll define our training and data, `x`, and labels, `y`. `x` is two dimensional, since the first dimension represents the sample in the mini-batch. For the sake of simplicity, we'll use a mini-batch with a single sample:

```
import numpy as np

# The first dimension represents the mini-batch
x = np.array([[0, 0, 0, 0, 1, 0, 1, 0, 1, 0]])

y = np.array([3])
```

The recurrence relation defined by this network is $s_t = s_{t-1} * W + x_t * U$. Note that this is a linear model since we don't apply a non-linear function in this formula. We can implement a recurrence relation as follows:

```
def step(s, x, U, W):
    return x * U + s * W
```

The states s_t and the weights *W* and *U* are single scalar values. A good solution to this is to just get the sum of the inputs across the sequence. If we set *U=1*, then whenever input is received, we will get its full value. If we set *W=1*, then the value we would accumulate would never decay. So, for this example, we would get the desired output: 3.

Nevertheless, let's use this simple example to network the training and implementation of this neural network. This will be interesting, as we will see in the rest of this section. Let's look at how we could get this result through backpropagation.

Backpropagation through time

Backpropagation through time is the typical algorithm we use to train recurrent networks (`http://axon.cs.byu.edu/~martinez/classes/678/Papers/Werbos_BPTT.pdf`). As the name suggests, it's based on the backpropagation algorithm we discussed in `Chapter 2`, *Neural Networks*.

The main difference between regular backpropagation and backpropagation through time is that the recurrent network is unfolded through time for a certain number of time steps (as illustrated in the preceding diagram). Once the unfolding is complete, we end up with a model that is quite similar to a regular multilayer feedforward network. That is, one hidden layer of that network represents one step through time. The only differences are that each layer has multiple inputs: the previous state s_{t-1} and the current input x_t. The parameters *U* and *W* are shared between all hidden layers.

The forward pass unwraps the RNN along the sequence and builds a stack of states for each step. The following is an implementation of the forward pass, which returns the activation *s* for each recurrent step and each sample in the batch:

```
def forward(x, U, W):
    # Number of samples in the mini-batch
    number_of_samples = len(x)

    # Length of each sample
    sequence_length = len(x[0])

    # Initialize the state activation for each sample along the sequence
    s = np.zeros((number_of_samples, sequence_length + 1))

    # Update the states over the sequence
    for t in range(0, sequence_length):
        s[:, t + 1] = step(s[:, t], x[:, t], U, W)  # step function

    return s
```

Now that we have our forward step and loss function, we can define how the gradient is propagated backward. Since the unfolded RNN is equivalent to a regular feedforward network, we can use the chain rule we introduced in `Chapter 2`, *Neural Networks,* that is backpropagation.

Because the weights *W* and *U* are shared across the layers, we'll accumulate the error derivatives for each recurrent step and in the end we'll update the weights with the accumulated value.

First, we need to get the gradient of the output s_t with respect to the cost function ($\partial J/\partial s$). Once we have it, we'll propagate it backward through the stack of activities we built during the forward step. This backward pass pops activities off the stack to accumulate their error derivatives at each time step. The recurrence relation to propagate this gradient through the network can be written as follows (chain rule):

$$\frac{\partial J}{\partial s_{t-1}} = \frac{\partial J}{\partial s_t}\frac{\partial s_t}{\partial s_{t-1}} = \frac{\partial J}{\partial s_t}W$$

Here, *J* is the loss function.

The gradients of the parameters are accumulated as follows:

$$\frac{\partial J}{\partial U} = \sum_{t=0}^{n}\frac{\partial J}{\partial s_t}x_t$$

$$\frac{\partial J}{\partial W} = \sum_{t=0}^{n}\frac{\partial J}{\partial s_t}s_{t-1}$$

The following is an implementation of the backward pass:

1. The gradients for `U` and `W` are accumulated in `gU` and `gW`, respectively:

```
def backward(x, s, y, W):
    sequence_length = len(x[0])

    # The network output is just the last activation of sequence
    s_t = s[:, -1]

    # Compute the gradient of the output w.r.t. MSE cost function
at final state
    gS = 2 * (s_t - y)

    # Set the gradient accumulations to 0
    gU, gW = 0, 0

    # Accumulate gradients backwards
    for k in range(sequence_length, 0, -1):
        # Compute the parameter gradients and accumulate the
results.
        gU += np.sum(gS * x[:, k - 1])
        gW += np.sum(gS * s[:, k - 1])

        # Compute the gradient at the output of the previous layer
        gS = gS * W

    return gU, gW
```

2. We can now try to use gradient descent to optimize our network. We'll use the mean square error:

```
def train(x, y, epochs, learning_rate=0.0005):
    """Train the network"""

    # Set initial parameters
    weights = (-2, 0)  # (U, W)

    # Accumulate the losses and their respective weights
    losses = list()
    weights_u = list()
    weights_w = list()

    # Perform iterative gradient descent
    for i in range(epochs):
        # Perform forward and backward pass to get the gradients
        s = forward(x, weights[0], weights[1])

        # Compute the MSE cost function
        loss = (y[0] - s[-1, -1]) ** 2

        # Store the loss and weights values for later display
        losses.append(loss)

        weights_u.append(weights[0])
        weights_w.append(weights[1])

        gradients = backward(x, s, y, weights[1])

        # Update each parameter `p` by p = p - (gradient *
learning_rate).
        # `gp` is the gradient of parameter `p`
        weights = tuple((p - gp * learning_rate) for p, gp in
zip(weights, gradients))

    print(weights)
```

```
        return np.array(losses), np.array(weights_u),
np.array(weights_w)
```

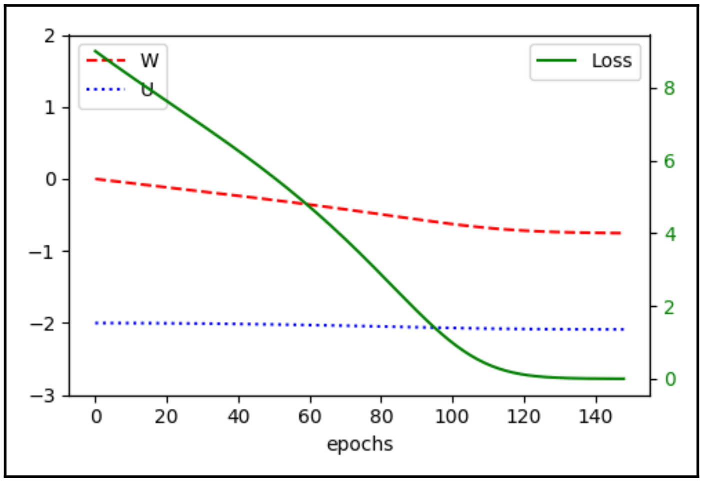
3. Next, we'll implement the related `plot_training` function, which displays the weights and the loss:

```
def plot_training(losses, weights_u, weights_w):
    import matplotlib.pyplot as plt

    # remove nan and inf values
    losses = losses[~np.isnan(losses)][:-1]
    weights_u = weights_u[~np.isnan(weights_u)][:-1]
    weights_w = weights_w[~np.isnan(weights_w)][:-1]

    # plot the weights U and W
    fig, ax1 = plt.subplots(figsize=(5, 3.4))

    ax1.set_ylim(-3, 2)
    ax1.set_xlabel('epochs')
    ax1.plot(weights_w, label='W', color='red', linestyle='--')
    ax1.plot(weights_u, label='U', color='blue', linestyle=':')
    ax1.legend(loc='upper left')

    # instantiate a second axis that shares the same x-axis
    # plot the loss on the second axis
    ax2 = ax1.twinx()

    # uncomment to plot exploding gradients
    ax2.set_ylim(-3, 200)
    ax2.plot(losses, label='Loss', color='green')
    ax2.tick_params(axis='y', labelcolor='green')
    ax2.legend(loc='upper right')

    fig.tight_layout()

    plt.show()
```

4. Finally, we can run this code:

```
losses, weights_u, weights_w = train(x, y, epochs=150)
plot_training(losses, weights_u, weights_w)
```

This will produce the following graph:



The RNN loss and the uninterrupted line represents the loss, while the dashed lines represent the weights

Vanishing and exploding gradients

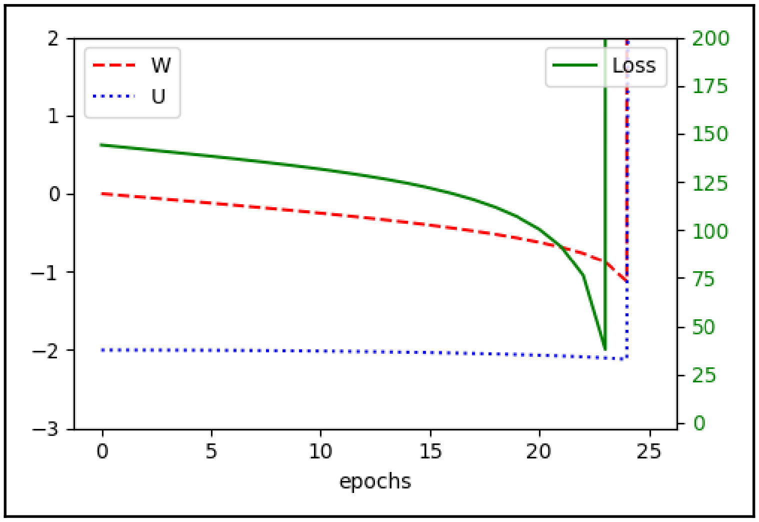
The preceding example has an issue, though. Let's run the training with a longer sequence:

```
x = np.array([[0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1,
0]])

y = np.array([12])

losses, weights_u, weights_w = train(x, y, epochs=150)
plot_training(losses, weights_u, weights_w)
```

The output is:

```
chapter_07_001.py:5: RuntimeWarning: overflow encountered in multiply
    return x * U + s * W
chapter_07_001.py:40: RuntimeWarning: invalid value encountered in multiply
    gU += np.sum(gS * x[:, k - 1])
chapter_07_001.py:41: RuntimeWarning: invalid value encountered in multiply
    gW += np.sum(gS * s[:, k - 1])
```

The reason for these warnings is that the final parameters *U* and *W* end up as **Not a Number** (**NaN**). In the following graph, we can see the weight updates and loss during the training steps:



Parameters and loss function during exploding gradients scenario

The weights slowly move toward the optimum and the loss decreases until it overshoots at epoch 23 (the exact epoch is unimportant, though). What happens is that the cost surface we are training on is highly unstable. Using small steps, we might move to a stable part of the cost function, where the gradient is low, and suddenly hits upon a jump in cost and a corresponding huge gradient. Because the gradient is so huge, it will have a big effect on our weights via the weight updates – they become NaNs (as illustrated by the jump outside the plot). This problem is known as exploding gradients.

There is also the vanishing (as opposed to exploding) gradients problem, which we first mentioned in `chapter 3`, *Deep Learning Fundamentals*. The gradient decays exponentially over the number of steps to a point where it becomes extremely small in the earlier states. In effect, they are overshadowed by the larger gradients from more recent time steps, and the network's ability to retain the history of these earlier states vanishes. This problem is harder to detect because the training will still work and the network will produce valid outputs (unlike with exploding gradients). It just won't be able to learn long-term dependencies.

Although vanishing and exploding gradients are present in regular neural networks, they are especially pronounced in RNNs. The reasons for this are as follows:

- Depending on the sequence's length, an unfolded RNN can be much deeper compared to a regular network.
- The weights *W* are shared across all steps. This means that the recurrence relation that propagates the gradient backward through time forms a geometric sequence:

$$\frac{\partial s_t}{\partial s_{t-m}} = \frac{\frac{\partial s_t}{\partial s_{t-1}} * \ldots * \partial s_{t-m+1}}{\partial s_{t-m}} = W^m$$

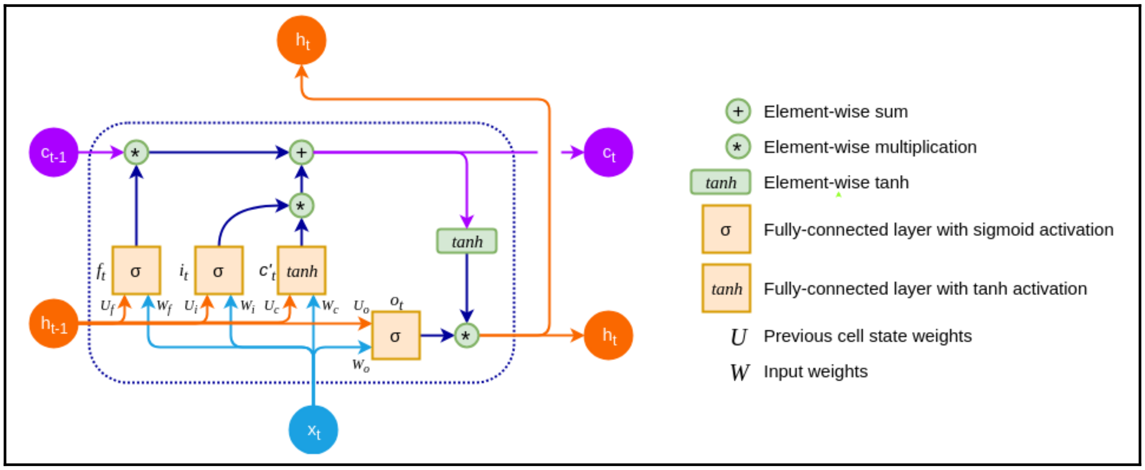
In our simple linear RNN, the gradient grows exponentially if *|W| > 1* (exploding gradient). For example, 50 time steps over W=1.5 is *W50 = 1.550 ≈ 6 * 108*. The gradient shrinks exponentially if *|W| <1* (vanishing gradient). For example, 20 time steps over *W=0.6* is *W20 = 0.620 ≈ 3*10-5*. If the weight parameter *W* is a matrix instead of a scalar, this exploding or vanishing gradient is related to the largest eigenvalue (ϱ) of *W* (also known as a spectral radius). It is sufficient for $\varrho < 1$ for the gradients to vanish, and it is necessary for $\varrho > 1$ for them to explode.

Long short-term memory

Hochreiter and Schmidhuber studied the problems of vanishing and exploding gradients extensively and came up with a solution called **long short-term memory** (LSTM) (`https://www.bioinf.jku.at/publications/older/2604.pdf`). LSTMs can handle long-term dependencies due to a specially crafted memory cell. In fact, they work so well that most of the current accomplishments in training RNNs on a variety of problems are due to the use of LSTMs. In this section, we'll explore how this memory cell works and how it solves the vanishing gradients issue.

The key idea of LSTM is the cell state (in addition to the hidden RNN state), where the information can only be explicitly written in or removed so that the state stays constant if there is no outside interference. The cell state can only be modified by specific gates, which are a way to let information pass through. These gates are composed of a logistic sigmoid function and element-wise multiplication. Because the logistic function only outputs values between 0 and 1, the multiplication can only reduce the value running through the gate. A typical LSTM is composed of three gates: a forget gate, an input gate, and an output gate. The cell state, input, and output are all vectors, so the LSTM can hold a combination of different information blocks at each time step.

The following is a diagram of a LSTM cell (idea from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`):


h_t
c_{t-1}
c_t
h_{t-1}
h_t
x_t
f_t
i_t
c'_t
o_t
σ
tanh
U_f
W_f
U_i
W_i
U_c
W_c
U_o
W_o
Element-wise sum
Element-wise multiplication
Element-wise tanh
Fully-connected layer with sigmoid activation
Fully-connected layer with tanh activation
U Previous cell state weights
W Input weights

LSTM cell

Before we continue, let's introduce some notations:

- x_t, c_t, and h_t are the LSTM input, cell memory state, and output (or hidden state) in moment t. c'_t is the candidate cell state (more on that later). The input x_t and the previous cell output h_{t-1} are connected to each gate and the candidate cell vector with sets of weights W and U, respectively.
- c_t is the cell state in moment t.
- f_t, i_t, and o_t are the forget, input, and output gates of the LSTM cell.

Let's start with the forget gate. As the name suggests, it decides whether we want to erase the cell state or not. This gate was not in the original LSTM that was proposed by Hochreiter. Instead, it was proposed by Gers and others (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.5709&rep=rep1&type=pdf`). The forget gate bases its decision on the output of the previous cell h_{t-1} and the current input x_t:

$$f_t = \sigma(W_f x_t + U_f h_{t-1})$$

It applies element-wise logistic functions on each element of the previous cell's vector c_{t-1}. Because the operation is element-wise, the values of this vector are squashed in the [0, 1] range. An output of 0 erases a specific c_{t-1} cell block completely and an output of 1 allows the information in that cell block to pass through. This means that the LSTM can get rid of irrelevant information in its cell state vector.

The input gate decides what new information is going to be added to the memory cell. This is done in two parts. The first part decides whether information is going to be added. As in the input gate, it bases it decision on h_{t-1} and x_t. It outputs 0 or 1 through the logistic function for each cell block of the cell's vector. An output of 0 means that no information is added to that cell block's memory. As a result, the LSTM can store specific pieces of information in its cell state vector:

$$i_t = \sigma(W_i x_t + U_i h_{t-1})$$

The candidate input to be added, c'_t, is based on the previous output h_{t-1} and the current input x_t. It is transformed via a *tanh* function:

$$c'_t = tanh(W_c x_t + U_c h_{t-1})$$

The forget and input gates decide the new cell state by choosing which parts of the new and the old state to include:

$$c_t = f_t * c_{t-1} \oplus i_t * c'_t$$

The output gate decides what the total cell output is going to be. It takes h_{t-1} and x_t as inputs and outputs 0 or 1 (via the logistic function) for each block of the cell's memory. An output of 0 means that the block doesn't output any information, while an output of 1 means that the block can pass through as a cell's output. The LSTM can thus output specific blocks of information from its cell state vector:

$$o_t = \sigma(W_o x_t + U_o h_{t-1})$$

Finally, the LSTM cell output is transferred by a *tanh* function:

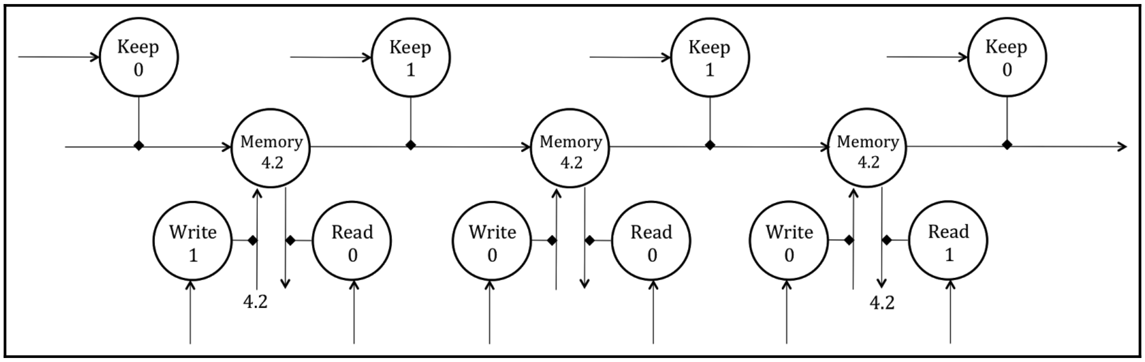
$$h_t = o_t * tanh(c_t)$$

Because all of these formulas are derivable, we can chain LSTM cells together just like we chain simple RNN states together and train the network via backpropagation through time.

But how does the LSTM protect us from vanishing gradients? Notice that the cell state is copied identically from step to step if the forget gate is 1 and the input gate is 0. Only the forget gate can completely erase the cell's memory. As a result, memory can remain unchanged over a long period of time. Also, notice that the input is a *tanh* activation that's been added to the current cell's memory. This means that the cell memory doesn't blow up and is quite stable.

Let's use an example to demonstrate how a LSTM is unfolded. We'll start by using the value of 4.2 as network input. The input gate is set to 1 so that the complete value is stored. Then, for the next two time steps, the forget gate is set to 1. In this way, all the information is kept throughout these steps and no new information is being added because the input gates are set to 0. Finally, the output gate is set to 1, and 4.2 is outputted and remains unchanged.

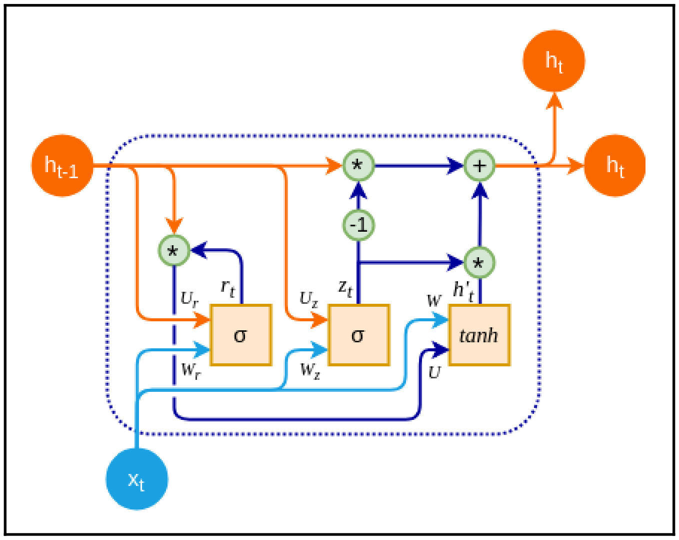
The following is an example of a LSTM unfolding through time (source: `http://nikhilbuduma.com/2015/01/11/a-deep-dive-into-recurrent-neural-networks/`):



Unrolling a LSTM through time

Gated recurrent units

A **gated recurrent unit** (GRU) is a type of recurrent block that was introduced in 2014 by Kyunghyun Cho et al. (`https://arxiv.org/abs/1406.1078`, `https://arxiv.org/abs/1412.3555`), as an improvement over LSTM (see the following diagram). A GRU unit usually has similar or better performance to a LSTM, but it does so with fewer parameters and operations:



A GRU cell

Similar to the "classic" RNN, a GRU cell has a single hidden state, h_t. You can think of it as a combination of the hidden and cell states of an LSTM. The GRU cell has two gates:

- An update gate z_t, which is a combination of the input and forget LSTM gates. It decides what information to discard and what new information to include in its place, based on the network input x_t and the previous cell hidden state h_{t-1}. By combining the two gates, we can ensure that the cell will forget information, but only when we are going to include new information in its place:

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

- A reset gate, r_t, which uses the previous cell state h_{t-1} and the network input x_t to decide how much of the previous state to pass through:

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

Next, we have the candidate state, $h't$:

$$h'_t = tanh(Wx_t + U(r_t * h_{t-1}))$$

Finally, the GRU output h_t at time t is a linear interpolation between the previous output h_{t-1} and the candidate output h'_t:

$$h_t = (1 - z_t) * h_{t-1} \oplus z_t * h'_t$$

Language modeling

Language modeling is the task of computing the probability of a sequence of words. Language models are crucial to a lot of different applications, such as speech recognition, optical character recognition, machine translation, and spelling correction. For example, in American English, the two phrases *wreck a nice beach* and *recognize speech* are almost identical in pronunciation, but their respective meanings are completely different from each other. A good language model can distinguish which phrase is most likely to be correct, based on the context of the conversation. This section will provide an overview of word and character-level language models and how RNNs can be used to build them.

Word-based models

A word-based language model defines a probability distribution over sequences of words. Given a sequence of words of length m, it assigns a probability $P(w_1, \ldots, w_m)$ to the full sequence of words. We can use these probabilities as follows:

- To estimate the likelihood of different phrases in natural language processing applications.
- As a generative model to create new text. A word-based language model can compute the likelihood of a given word to follow a sequence of words.

N-grams

The inference of the probability of a long sequence, say $w_1, \ldots, w_m$, is typically infeasible. Calculating the joint probability of $P(w_1, \ldots, w_m)$ would be done by applying the following chain rule:

$$P(w_1, \ldots, w_m) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \ldots P(w_m|w_1, \ldots, w_{m-1})$$

The probability of the later words given the earlier words would be especially difficult to estimate from the data. That's why this joint probability is typically approximated by an independence assumption that the *i*th word is only dependent on the *n-1* previous words. We'll only model the joint probabilities of combinations of *n* sequential words, called n-grams. For example, in the phrase *the quick brown fox*, we have the following n-grams:

- **1-gram**: "The," "quick," "brown," and "fox" (also known as a unigram)
- **2-grams**: "The quick," "quick brown," and "brown fox" (also known as a bigram)
- **3-grams**: "The quick brown" and "quick brown fox" (also known as a trigram)
- **4-grams**: "The quick brown fox"

The inference of the joint distribution is approximated via n-gram models that split the joint distribution into multiple independent parts.

The term *n-grams* can be used to refer to other types of sequences of length *n*, such as *n* characters.

If we have a huge corpus of text, we can find all the n-grams up until a certain *n* (typically 2 to 4) and count the occurrence of each n-gram in that corpus. From these counts, we can estimate the probabilities of the last word of each n-gram, given the previous *n-1* words:

- **1-gram**: $P(word) = \frac{count(word)}{\text{total number of words in corpus}}$
- **2-gram**: $P(w_i|w_{i-1}) = \frac{count(w_{i-1},w_i)}{count(w_{i-1})}$
- **N-gram**: $P(w_{n+i}|w_n,\ldots,w_{n+i-1}) = \frac{count(w_n,\ldots,w_{n+i-1},w_{n+i})}{count(w_n,\ldots,w_{n+i-1})}$

The independence assumption that the *i*th word is only dependent on the previous *n-1* words can now be used to approximate the joint distribution.

For example, for a unigram, we can approximate the joint distribution by using the following formula:

$$P(w_1,\ldots,w_m) = P(w_1)P(w_2)P(w_3)\ldots.P(w_m)$$

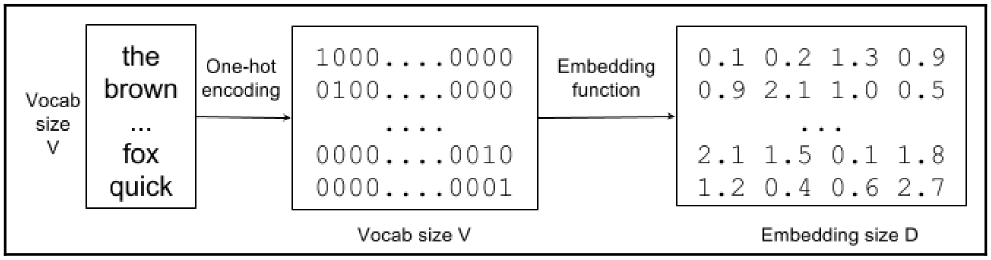
For a trigram, we can approximate the joint distribution by using the following formula:

$$P(w_1,\ldots,w_m) = P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)\ldots.P(w_m|w_{m-2},w_{m-1})$$

We can see that, based on the vocabulary size, the number of n-grams grows exponentially with *n*. For example, if a small vocabulary contains 100 words, then the number of possible 5-grams would be $100^5 = 10{,}000{,}000{,}000$ different 5-grams. In comparison, the entire works of Shakespeare contain around *30,000* different words, illustrating the infeasibility of using n-grams with a large *n*. Not only is there the issue of storing all the probabilities, but we would also need a very large text corpus to create decent n-gram probability estimations for larger values of *n*. This problem is known as the curse of dimensionality. When the number of possible input variables (words) increases, the number of different combinations of these input values increases exponentially. The curse of dimensionality arises when the learning algorithm needs at least one example per relevant combination of values, which is the case in n-gram modeling. The larger our *n*, the better we can approximate the original distribution and the more data we would need to make good estimations of the n-gram probabilities.

Neural language models

One way to overcome the curse of dimensionality is by learning a lower dimensional, distributed representation of the words (`http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf`). This distributed representation is created by learning an embedding function that transforms the space of words into a lower dimensional space of word embeddings, as follows:



Words -> one-hot encoding -> word embedding vectors

Words from the vocabulary with size *V* are transformed into one-hot encoding vectors of size *V* (each word is encoded uniquely). Then, the embedding function transforms this V-dimensional space into a distributed representation of size *D* (here, D=4).

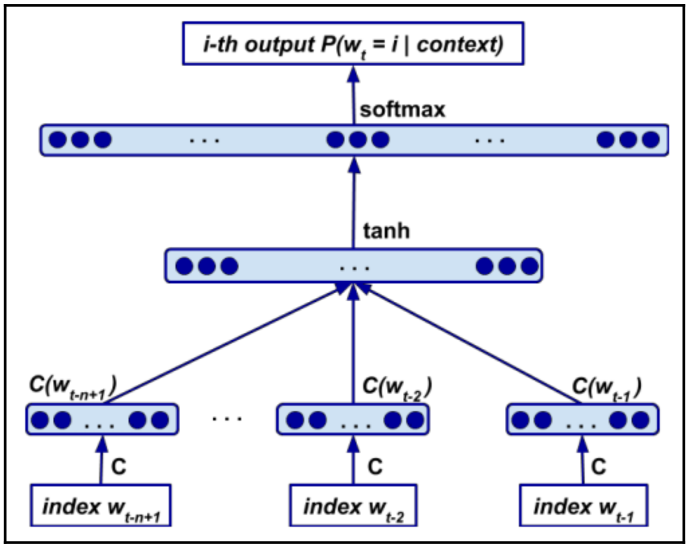
The idea is that the embedding function learns semantic information about the words. It associates each word in the vocabulary with a continuous-valued vector representation, that is, the word embedding. Each word corresponds to a point in this embedding space, and different dimensions correspond to the grammatical or semantic properties of these words.

The goal is to ensure that the words close to each other in the embedding space should have similar meanings. In this way, the information that some words are semantically similar can be exploited by the language model. For example, it might learn that "fox" and "cat" are semantically related and that both "the quick brown fox" and "the quick brown cat" are valid phrases. A sequence of words can then be replaced with a sequence of embedding vectors that capture the characteristics of these words. We can use this sequence as a base for various NLP tasks. For example, a classifier trying to classify the sentiment of an article might be trained on using previously learned word embedding's, instead of one-hot encoding vectors. In this way, the semantic information of the words becomes readily available for the sentiment classifier.

Word embedding's is one of the central paradigms when solving NLP tasks. We can use them to improve the performance of other tasks where there might not be a lot of labeled data available.

Neural probabilistic language model

It is possible to learn the language model and, implicitly, the embedding function via a feedforward fully-connected network. Given a sequence of *n-1* words (w_{t-n+1} , ..., w_{t-1}), it tries to output the probability distribution of the next word w_t (the following diagram is based on `http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf`):



Neural network language model that outputs the probability distribution of the word w_t, given the words w_{t-1} ... w_{t-n+1}. *C* is the embedding matrix

The network layers play different roles:

1. The embedding layer takes the one-hot representation of the word w_i and transforms it into the word's embedding vector by multiplying it with the embedding matrix *C*. This computation can be efficiently implemented with table lookup. The embedding matrix *C* is shared over the words, so all words use the same embedding function. *C* is represented by a *V * D* matrix, where *V* is the size of the vocabulary and *D* the size of the embedding.
2. The resulting embeddings are concatenated and serve as an input to the hidden layer, which uses *tanh* activation. The output of the hidden layer is thus represented by the function $z = tanh(H \cdot (concat(C(w_{t-n+1}), ..., C(w_{t-1})) + d))$, where *H* is the embedding-to-hidden layer weights and *d* are the hidden biases.
3. Finally, we have the output layer with weights *U*, bias *b*, and softmax activation, which maps the hidden layer to the word space probability distribution: $y = softmax(z*U + b)$.

This model simultaneously learns an embedding of all the words in the vocabulary (embedding layer) and a model of the probability function for sequences of words (network output). It is able to generalize this probability function to sequences of words that were not seen during training. A specific combination of words in the test set might not be seen in the training set, but a sequence with similar embedding features is much more likely to be seen during training. Since we can construct the training data and labels based on the positions of the words (which already exist in the text), training this model is an unsupervised learning task.

word2vec

A lot of research has gone into creating better word embedding models, in particular by omitting learning the probability function over sequences of words. One of the most popular ways to do this is via word2vec (`http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf` and `https://arxiv.org/pdf/1301.3781.pdf`). To create embedding vectors with a word2vec model, we'll need a simple neural network that has the following:

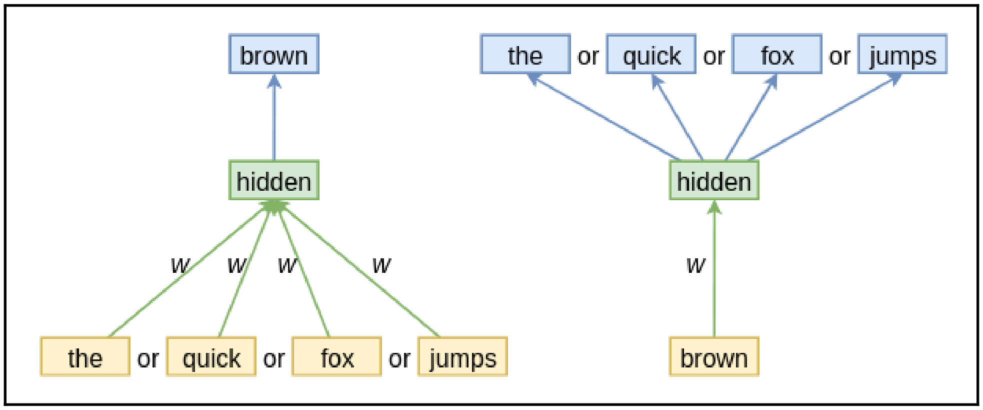
- It has an input, hidden, and an output layer
- The input is a single one-hot encoded word representation
- The output is a single softmax, which predicts the most likely word to be found in the context (proximity) of the input word

We'll train the network with gradient descent and backpropagation. The training set consists of (input, label) one-hot encoded pairs of words, appearing in close proximity to each other in the text. For example, if part of the text is the sequence (quick, brown, fox), the training samples will include the pairs (quick,brown), (brown,fox), and so on.

The embedding vectors are represented by the input-to-hidden weights of the network. They are $V * D$ shaped matrices, where V is the size of the vocabulary and D is the length of the embedding vector (which is the same as the number of neurons in the hidden layer). We can think of the weights as a table, where each row represents one word embedding vector. Because the input is one-hot encoded, we'll always have only a single active row of the weights during training. That is, for each input sample (word), only the word's own embedding vector will participate. Since we are only interest in the embeddings, we'll discard the rest of the network when the training is finished.

Depending on the way we train the model, we have two flavors:

- **Continuous bag of words (CBOW):** Here, the neural network is trained to predict which word fits in a sequence of words, where a single word has been intentionally removed. For example, given the sqeuence "The quick _____ fox jumps", the network will predict "brown". But, as we mentioned previously, the network takes only a single word as input. Therefore, we'll transform this sentence into multiple training (input, target) pairs: `(the, brown), (quick, brown), (fox, brown), and (fox, jumps).`
- **Skip-gram:** This is the opposite of CBOW. Given an input word, it predicts which words surround it. For example, the word "brown" will predict the words "The quick fox jumps". As with CBOW, we'll transform this sentence into pairs `(brown, the), (brown, quick), (brown, fox), (brown, jumps):`

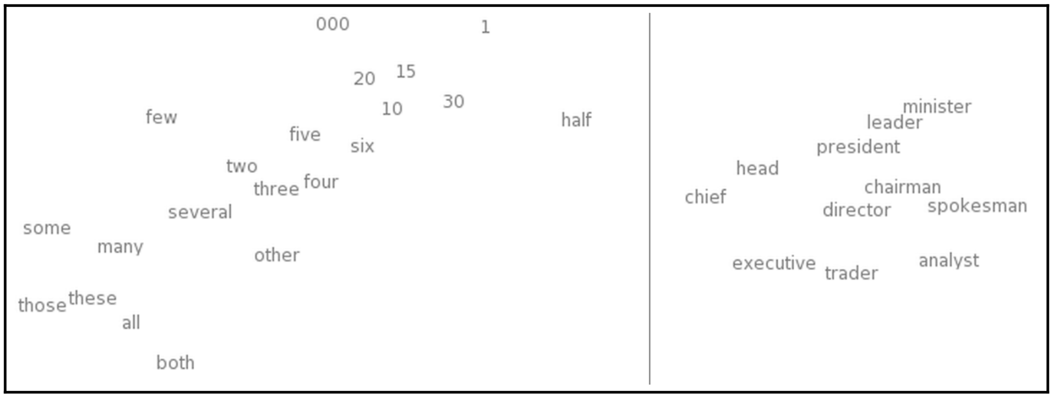


CBOW (left) and skip-gram (right)

There are other popular embedding models such as GloVe (`https://nlp.stanford.edu/projects/glove/`) and fastText (`https://fasttext.cc/`).

Visualizing word embedding vectors

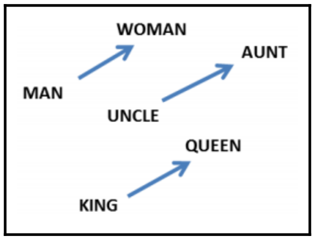
In the following diagram, we can see a 2D projection of some word embeddings (source `http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/`). The words, which are semantically close, are also close to each other in the embedding space:



Related words in a 2D embedding space are close to each other in this space

A surprising result is that these word embedding's can capture analogies between words as differences (as shown in the following diagram; source: `https://www.aclweb.org/anthology/N/N13/N13-1090.pdf`). For example, it might capture that the difference between the embedding of "woman" and "man" encodes the gender and that this difference is the same in other gender-related words such as "queen" and "king":

- embed(woman) - embed(man) ≃ embed(aunt) - embed(uncle)
- embed(woman) - embed(man) ≃ embed(queen) - embed(king):



Word embedding's can capture semantic differences between words

Character-based models for generating new text

In this section, we'll discuss how to generate new text using character-based models via **TensorFlow (TF)**. This is an example of a "many-to-many" relationship, such as the one we defined in the *Recurrent neural networks* section. We'll only discuss the most interesting code sections, but the full example lives at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/tree/master/ch07/language%20model`.

In most cases, language modeling is performed at the word level, where the distribution is over a fixed vocabulary of $|V|$ words. Vocabularies in realistic tasks, such as the language models used in speech recognition, often exceed *100,000* words. This large dimensionality makes modeling the output distribution very challenging. Furthermore, these word level models are quite limited when it comes to modeling text data that contains non-word strings, such as multi-digit numbers or words that were never part of the training data (out-of-vocabulary words).

To overcome these issues, we can use character-level language models (`https://arxiv.org/abs/1308.0850`). They model the distribution over sequences of characters instead of words, thus allowing you to compute probabilities over a much smaller vocabulary. The vocabulary of a character-level model comprises all the possible characters in our text corpus. There is a downside to these models, though. By modeling the sequence of characters instead of words, we need to model much longer sequences to capture the same information over time. To capture these long-term dependencies, let's use an LSTM language model.

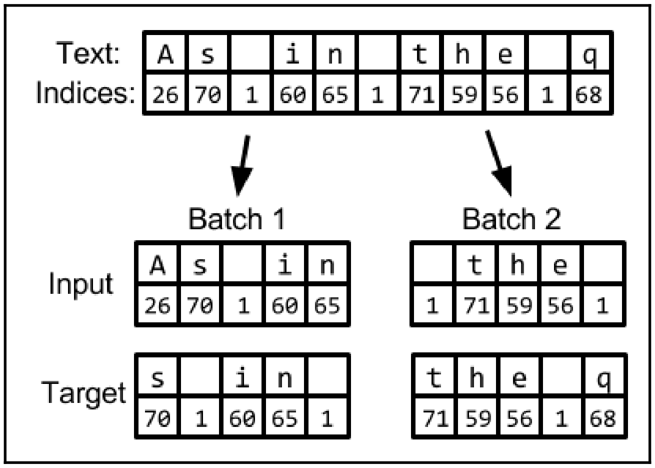
The following part of this section will go into detail on how to implement a character-level LSTM in TensorFlow and how to train it on Leo Tolstoy's *War and Peace*. This LSTM will model the probability of the next character, given the previously seen characters: $P(c_t \mid c_{t-1} \ldots c_{t-n})$.

Because the full text is too long to train a network with **backpropagation through time** (**BPTT**), we'll use a batched variant called truncated BPTT. We'll divide the training data into batches of fixed sequence length and train the network batch by batch. Since the batches are sequential, we can use the final state of one batch as the initial state of the next. This way, we can exploit the information stored in the state without having to do a full backward pass through the full input text.

Preprocessing and reading data

To train a good language model, we need a lot of data. The English translation of Leo Tolstoy's "War and peace", which contains more than *500,000* words, makes it a good candidate for our small example. The book is in the public domain and can be downloaded as plain text for free from Project Gutenberg (`http://www.gutenberg.org/`). As part of preprocessing, we'll remove the Gutenberg license, book information, and table of contents. Next, we will strip out newlines in the middle of sentences and reduce the maximum number of consecutive newlines allowed to two (the code can be found at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch07/language%20model/data_processing.py`).

To feed the data into the network, we'll convert it into a numerical format. Each character will be associated with an integer. In our example, we will extract a total of 98 different characters from the text corpus. Next, we will extract input and targets. For each input character, we will predict the next character. Because we are training with truncated BPTT, we'll extract all training batches from sequential locations in the text to exploit the continuity of the sequence. The process of converting the text into a list of indices and splitting it up into batches of input and targets is illustrated in the following diagram. The code lives at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch07/language%20model/data_reader.py`:



Converting text into input and target batches of integer labels with length 5. The batches are extracted from sequential locations in the text

LSTM network

Now, we'll train a two-layer LSTM network with 512 cells in each layer. The full code is available at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch07/language%20model/model.py`. Since we'll use truncated BPTT, we need to store the state between batches:

1. First, we'll define placeholders for our input and targets. The placeholders are the links between the model and the training data. We can feed the network with a single batch by setting its values to the placeholders. The first dimension of both the input and targets is the batch size, while the second is along the text sequence. Both placeholders take batches of sequences where the characters are represented by their index:

```
self.inputs = tf.placeholder(tf.int32, [self.batch_size,
self.sequence_length])
self.targets = tf.placeholder(tf.int32, [self.batch_size,
self.sequence_length])
```

2. To feed the characters into the network, we need to transform them into one-hot vectors. This can be done easily in TensorFlow with the following line of code:

```
self.one_hot_inputs = tf.one_hot(self.inputs,
depth=self.number_of_characters)
```

3. Next, we will define our multilayer LSTM architecture. First, we need to define the LSTM cells for each layer. `lstm_sizes` is a list of sizes for each layer. In our case, this is (512, 512):

```
cell_list = [tf.nn.rnn_cell.LSTMCell(lstm_size) for lstm_size in
self.lstm_sizes]
```

4. Then, we wrap the cells in a single multilayer RNN cell:

```
self.multi_cell_lstm = tf.nn.rnn_cell.MultiRNNCell(cell_list)
```

5. To store the state between the batches, we need to get the initial state of the network and wrap it in a variable to be stored. Because of computational reasons, TF stores LSTM states in a tuple of two separate tensors (`c` and `h` from the *Long short-term memory* section). We'll flatten this nested data structure with the `flatten` method, wrap each tensor in a variable, and repack it as the original structure with the `pack_sequence_as` method:

```
self.initial_state =
self.multi_cell_lstm.zero_state(self.batch_size, tf.float32)
```

```
# Convert to variables so that the state can be stored between
batches
# Note that LSTM states is a tuple of tensors, this structure has
to be
# re-created in order to use as LSTM state.
self.state_variables = tf.contrib.framework.nest.pack_sequence_as(
    self.initial_state,
    [tf.Variable(var, trainable=False)
     for var in
tf.contrib.framework.nest.flatten(self.initial_state)])
```

6. Now that we have the initial state defined as a variable, we can start unrolling the network through time. TensorFlow provides the `dynamic_rnn` method, which does this unrolling dynamically as per the sequence length of the input. This method will return a tuple consisting of a tensor representing the LSTM output and the final state:

```
lstm_output, final_state = tf.nn.dynamic_rnn(
    cell=self.multi_cell_lstm, inputs=self.one_hot_inputs,
    initial_state=self.state_variables)
```

7. Next, we need to store the final state as the initial state for the next batch. We'll use the `state_variable.assign` method to store each final state in the right initial state variable. The `control_dependencies` method forces the state to update before we return the LSTM output:

```
store_states = [
    state_variable.assign(new_state)
    for (state_variable, new_state) in zip(
        tf.contrib.framework.nest.flatten(self.state_variables),
        tf.contrib.framework.nest.flatten(final_state))]
with tf.control_dependencies(store_states):
    lstm_output = tf.identity(lstm_output)
```

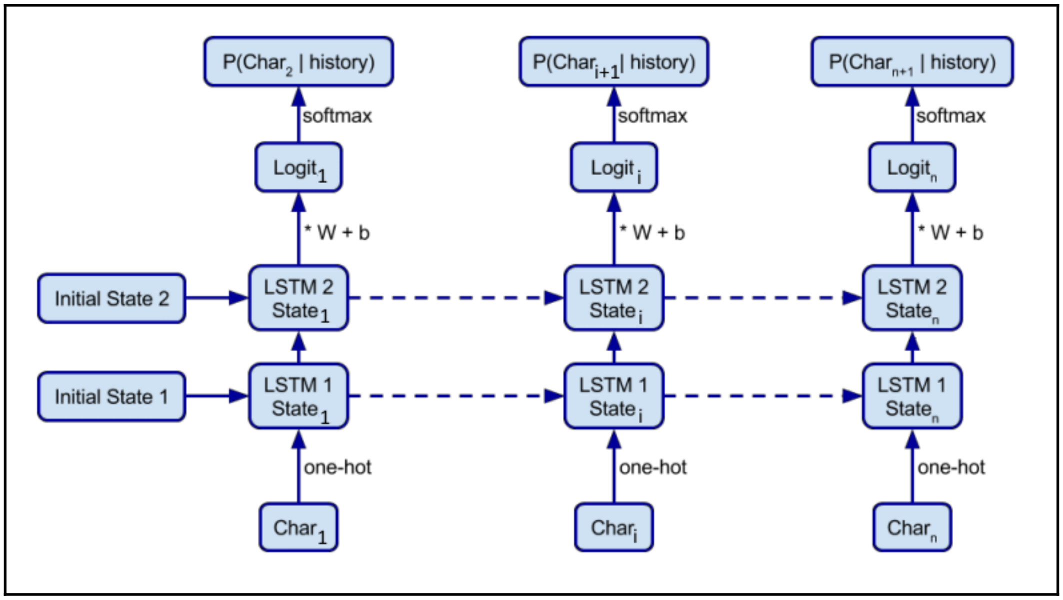
8. To get the logit output from the final LSTM output, we need to apply a linear transformation to the output so that it can have *batch size * sequence length * number of symbols* dimensions. Before this, we need to flatten the output to a matrix of size *number of outputs * number of output features*:

```
output_flat = tf.reshape(lstm_output, (-1, self.lstm_sizes[-1]))
```

Then, we can define and apply the linear transformation with a weight matrix *W* and bias *b* to get the logits, apply the softmax function, and reshape it to a tensor of the size *batch size * sequence length * number of characters*:

```
# Define output layer
self.logit_weights = tf.Variable(
    tf.truncated_normal(
        (self.lstm_sizes[-1], self.number_of_characters),
stddev=0.01),
    name='logit_weights')
self.logit_bias = tf.Variable(
    tf.zeros((self.number_of_characters)), name='logit_bias')
# Apply last layer transformation
self.logits_flat = tf.matmul(
    output_flat, self.logit_weights) + self.logit_bias
probabilities_flat = tf.nn.softmax(self.logits_flat)
self.probabilities = tf.reshape(
    probabilities_flat,
    (self.batch_size, -1, self.number_of_characters))
```

The following is a diagram of the unfolded LSTM character language model:



LSTM character language model unfolded

Training

Now that we have defined the input, targets, and the network architecture, let's implement the training:

1. The first step is to define a loss function, which describes the cost of outputting a wrong sequence of characters, given the input and targets. Because we are predicting the next character considering the previous characters, it's a classification problem and we will use cross-entropy loss. We can do this with the `sparse_softmax_cross_entropy_with_logits` TF function, which takes as input the logit network output (before the softmax) and the targets as class labels. To reduce the loss over the full sequence and all the batches, we'll use their mean value. First, we have to flatten the targets into one-dimensional vectors to make them compatible with the flattened network logit output:

```
# Flatten the targets to be compatible with the flattened logits
targets_flat = tf.reshape(self.targets, (-1,))
# Get the loss over all outputs
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=self.logits_flat, labels=targets_flat, name='x_entropy')
self.loss = tf.reduce_mean(loss)
```

2. Next, we'll define the TF training operation, which will optimize our network over the input and target batches. We'll use the Adam optimizer to stabilize the gradient updates. We'll also clip the gradients to prevent exploding gradients:

```
trainable_variables = tf.trainable_variables()
gradients = tf.gradients(loss, trainable_variables)
gradients, _ = tf.clip_by_global_norm(gradients, 5)
self.train_op = optimizer.apply_gradients(zip(gradients,
trainable_variables))
```

3. Then, we can start with the mini-batch optimization. If `data_feeder` is a generator that returns consecutive batches of input and targets, we can train the model by iteratively feeding them to the network via the placeholders. We'll reset the initial state every 100-mini batches so that the network learns how to deal with the initial states in the beginning of sequences. You can save the model with a TensorFlow saver to reload it for sampling later:

```
with tf.Session() as sess:
    sess.run(init_op)
    if restore:
        print('Restoring model')
        model.restore(sess)
    model.reset_state(sess)
```

```
            start_time = time.time()
            for i in range(minibatch_iterations):
                input_batch, target_batch = next(iter(data_feed))
                loss, _ = sess.run(
                    [model.loss, model.train_op],
                    feed_dict={model.inputs: input_batch, model.targets:
        target_batch})
```

Sampling

Once the model has been trained, we can generate new text by sampling the sequences. We can initialize our sampling architecture with the same code we used to train the model, but we'll need to set `batch_size` to `1` and `sequence_length` to `None`. In this way, we can generate a single string and sample sequences of different lengths. Then, we can initialize the parameters of the model with the parameters that were saved after the training. To start with the sampling, we'll feed the model an initial string (`prime_string`) to prime the state of the network. After that, we can sample the next character based on the output distribution of the softmax. Then, we can feed the newly sampled character as a new network input and get the output distribution for the next one. We can continue this process for a number of steps:

```
        def sample(self, session, prime_string, sample_length):
            self.reset_state(session)
            # Prime state
            print('prime_string: ', prime_string)
            for character in prime_string:
                character_idx = self.label_map[character]
                out = session.run(
                    self.probabilities,
                    feed_dict={self.inputs: np.asarray([[character_idx]])})
            output_sample = prime_string
            print('start sampling')
            # Sample for sample_length steps
            for _ in range(sample_length):
                sample_label = np.random.choice(
                    self.labels, size=(1), p=out[0, 0])[0]
                output_sample += sample_label
                sample_idx = self.label_map[sample_label]
                out = session.run(
                    self.probabilities,
                    feed_dict={self.inputs: np.asarray([[sample_idx]])})

            return output_sample
```

Example training

Now that we have our code for training and sampling, we can train the network on Leo Tolstoy's War and Peace and sample what the network has learned every couple of batch iterations. Let's prime the network with the phrase "*She was born in the year*" and see how it completes it during training.

The result we get after 500 batches is as follows: *She was born in the year sive but us eret tuke Toffhin e feale shoud pille saky doctonas laft the comssing hinder to gam the droved at ay vime*. The network has already picked up some distribution of characters and has come up with things that look like words.

After 5,000 batches, the network picks up a lot of different words and names: *She was born in the year he had meaningly many of Seffer Zsites. Now in his crownchy-destruction, eccention, was formed a wolf of Veakov one also because he was congrary, that he suddenly had first did not reply. It still invents plausible looking words likes "congrary" and "eccention".*

After 50,000 batches, the network outputs the following text: *She was born in the year 1813. At last the sky may behave the Moscow house there was a splendid chance that had to be passed the Rostóvs', all the times: sat retiring, showed them to confure the sovereigns."* It seems to have figured out that a year number is a very plausible word to follow up our prime string. Short strings of words seem to make sense, but the sentences on their own don't make sense yet.

After 500,000 batches, we stop the training and the network outputs the following: "*She was born in the year 1806, when he entered his thought on the words of his name. The commune would not sacrifice him: "What is this?" asked Natásha. "Do you remember?""*. We can see that the network is now trying to make sentences, but the sentences are not coherent. It is remarkable that it can now model small conversations in full sentences, including quotes and punctuation.

While not perfect, the RNN language model is still able to generate coherent phrases of text. We would like to encourage you to experiment further by using different architectures, increasing the size of the LSTM layers, putting a third LSTM layer in the network, or downloading more text data from the internet, to see how much you can improve the current model.

The language models we have discussed so far are used in many different applications, ranging from speech recognition to creating intelligent chatbots that are able to build up a conversation with the user.

Sequence to sequence learning

Many, many NLP problems can be formulated as sequence to sequence tasks. This is a type of task where an input sequence is transformed into another, different output sequence, not necessarily with the same length as the input. To better understand this concept, let's look at some examples:

- Machine translation is the most popular type of seq2seq task. The input sequences are the words of a sentence in one language and the output sequences are the words of the same sentence, translated into another language. For example, we can translate the English sequence "Tourist attraction" to the German "Touristenattraktion." Not only is the output sentence a different length – there is no direct correspondence between the elements of the input and output sequences. In particular, one output element corresponds to a combination of two input elements. Machine translation that's implemented with a single neural network is called **neural machine translation (NMT)**.
- Speech recognition, where we take different time frames of an audio input and convert them into text transcript.
- Question answering chatbots, where the input sequences are the words of a textual question and the output sequence is the answer to that question.
- Text summarization, where the input is a text document and the output is a short summary of the text's contents.

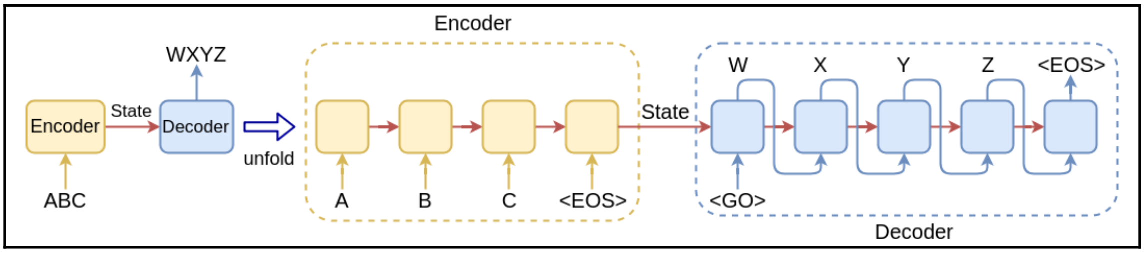
In 2014, Sutskever et al. (`https://arxiv.org/abs/1409.3215`) and Cho et al. (`https://arxiv.org/abs/1406.1078`) introduced a method called sequence to sequence (seq2seq, or encoder-decoder) learning, which uses RNNs in a way that's especially suited for solving tasks such as these. It's an example of an "indirect many-to-many" relationship, which we defined in the *Recurrent neural networks* section.

A seq2seq model consists of two parts: an encoder and a decoder. Here's how it works:

- The encoder is an RNN. The original paper uses LSTM, but GRU or other types would also work. Taken by itself, the encoder works in the usual way – it reads the input sequence, one timestep at a time, and updates its internal state after each step. The encoder will stop reading the input sequence once a special <EOS> – end of sequence – symbol is reached. If we assume that we use a textual sequence, the <EOS> symbol signals the end of a sentence.

- Once the encoder is finished, we'll signal the decoder so that it can start generating the output sequence with a special <GO> input signal. The encoder is also a RNN (LSTM or GRU). The link between the encoder and the decoder is the most recent encoder state vector h_t (also known as the thought vector), which is fed as the recurrence relation at the first decoder step. The decoder output y_{t+1} at step *t+1* is one element of the output sequence. We'll use it as an input at step *t*+2, then we'll generate new output, and so on.

The following is a diagram of the seq2seq model (inspired by `https://arxiv.org/abs/1409.3215`):



A seq2seq model

In the case of textual sequences, we'll use word embedding vectors as the encoder input. The decoder output would be the softmax over all the words in the vocabulary.

To summarize, the seq2seq model solves the problem of varying input/output sequence lengths by encoding the input sequence in a fixed length state vector and then using this vector as a base to generate the output sequence. We can formalize this by saying that it tries to maximize the probability:

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$$

This is equivalent to the following:

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = p(y_1 | v)\, p(y_2 | v, y_1) \ldots\, p(y_{T'} | v, y_1, \ldots, y_{T'-1})$$

Here are the following steps:

- $p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T)$ is the conditional probability where $(x_1, \ldots, x_T)$is the input sequence with length *T* and$(y_1, \ldots, y_{T'})$ is the output sequence with length *T'*.
- v is the fixed length encoding of the input sequence (the last state vector of the encoder).
- $p(y_{T'} | v, y_1, \ldots, y_{T'-1})$ is the probability of an output word $y_{T'}$ given prior words *y*, as well as the vector *v*.

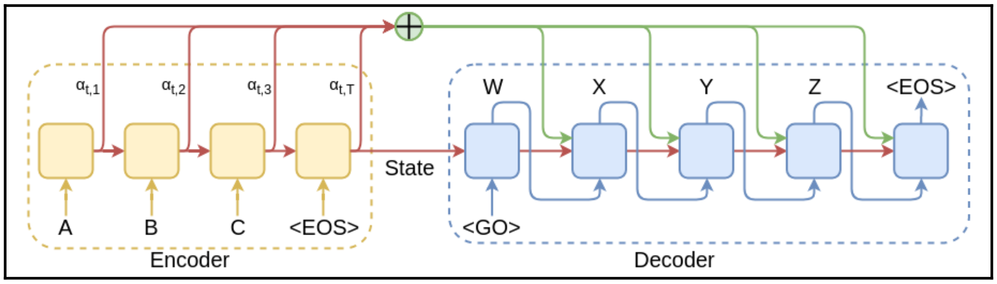
The original seq2seq paper introduces a few tricks to enhance the training and performance of the model:

- The input sequence is fed to the decoder in reverse. For example, "ABC" -> "WXYZ" would become "CBA" -> "WXYZ". There is no clear explanation of why this works, but the authors have shared their intuition. Since this is a step-by-step model, if the sequences were in normal order, each source word in the source sentence would be far from its corresponding word in the output sentence. If we reverse the input sequence, the average distance between input/output words won't change, but the first input words will be very close to the first output words. This will help the model to establish better "communication" between the input and output sequences.
- Besides <EOS> and <GO>, the model uses two other special symbols:
    - <UNK> – unknown: This is used to replace rare words so that the vocabulary size wouldn't grow too large.
    - <PAD>: For performance reasons, we have to train the model with sequences of a fixed length. However, this contradicts the real-world training data, where the sequences can have arbitrary lengths. To solve this, the shorter sequences are filled with the special <PAD> symbol.

Sequence to sequence with attention

The decoder has to generate the entire output sequence based solely on the thought vector. For this to work, the thought vector has to encode the entire information of the input sequence. However, the encoder is an RNN and we can expect that its hidden state will carry more information about the latest sequence elements, compared to the earliest.

Using LSTM cells and reversing the input helps, but cannot prevent it entirely. Because of this, the thought vector becomes something of a bottleneck and the seq2seq model works well for short sentences, but the performance deteriorates for longer ones. To solve this problem, Bahdanau et al. (`https://arxiv.org/abs/1409.0473`) proposed a seq2seq extension called an attention mechanism, which provides a way for the decoder to work with all encoder hidden states, and not just the last one. Although the authors proposed it in the context of NMT, it is generic and can be applied to any seq2seq task:



Attention mechanism

Attention works by plugging an additional **context vector** between the encoder and the decoder. That is, the hidden decoder state s_t at time t is now a function not only of the hidden state and decoder output at step t-1, but also of the context vector c_t:

$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

Each decoder step has a unique context vector and the context vector for one decoder step is just **a weighted sum of all encoder hidden states**:

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} h_i$$

Here:

- c_t is the context vector for a decoder output step t out of T' the total output steps.
- h_i is the hidden state of encoder step i out of T total input steps.

- $\alpha_{t,i}$ is the weight associated with h_i in the context of the current decoder step t. If $\alpha_{t,i}$ is large, then the decoder will pay a lot of attention to h_i at step t. However, the input sequence states will have different weights, depending on the current output step. For example, if the input and output sequences have lengths of 10, the weights will be represented by a 10 x 10 matrix for a total of 100 weights. But how do we compute the weights? First, we should mention that the sum of all weights for a decoder position t is 1. We can implement this with softmax:

$$\alpha_{t,i} = \frac{exp(e_{t,i})}{\sum_{k=1}^{T} exp(e_{t,k})}$$

  Here, $e_{t,k}$ is an alignment model, which scores how well the inputs around position k match the output at position t. This score is based on the previous decoder state s_{t-1} (we use s_{t-1} because we have not computed s_t yet), as well as the encoder state h_k:

$$e_{t,k} = a(s_{t-1}, h_k)$$

  Since training the seq2seq model uses gradient descent and backpropagation, e has to be differentiable. For that reason, e is usually a simple neural network with one hidden layer.

Speech recognition

In the previous sections, we saw how RNNs can be used to learn patterns of many different time sequences. In this section, we will look at how these models can be used for the problem of recognizing and understanding speech. We will give a brief overview of the speech recognition pipeline and provide a high-level view of how we can use neural networks in each part of the pipeline.

Speech recognition pipeline

Speech recognition tries to find a transcription of the most probable word sequence considering the acoustic observations provided:

transcription = argmax(P(words | audio features))

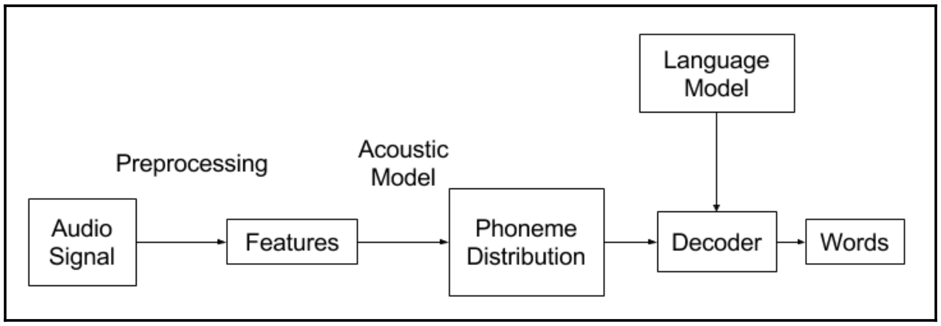
This probability function is typically modeled in different parts (note that the normalizing term P (audio features) is usually ignored):

*P (words | audio features) = P (audio features | words) * P (words)*

*= P (audio features | phonemes) * P (phonemes | words) * P (words)*

Phonemes are basic units of sound that define the pronunciation of words. For example, the word "bat" is composed of three phonemes: `/b/`, `/ae/`, and `/t/`. Each phoneme is tied to a specific sound. Spoken English consists of around 44 phonemes.

Each of these probability functions will be modeled by different parts of the recognition system. A typical speech recognition pipeline takes in an audio signal and performs preprocessing and feature extraction. These features are then used in an acoustic model that tries to learn how to distinguish between different sounds and phonemes: *P (audio features | phonemes)*. These phonemes are then matched to characters or words with the help of pronunciation dictionaries: *P(phonemes | words)*. The probabilities of the words that are extracted from the audio signal are then combined with the probabilities of a language model, *P(words)*. The most likely sequence is then found via a decoding search step. A high-level overview of this speech recognition pipeline is described in the following diagram:



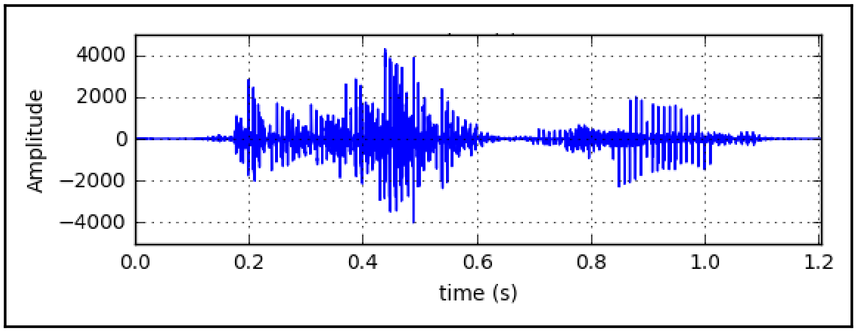
Overview of a typical speech recognition pipeline

Large, real-world vocabulary speech recognition pipelines are usually based on this pipeline. However, they use a lot of tricks and heuristics in each step to make the problem tractable. While these details are out of the scope of this section, there is open source software available – Kaldi (`https://github.com/kaldi-asr/kaldi`) – that allows you to train a speech recognition system with advanced pipelines.

In the following sections, we will briefly describe each of the steps in this standard pipeline and how deep learning can help improve these steps.

Speech as input data

Speech is a type of sound that typically conveys information. It is a vibration that propagates through a medium, such as air. If these vibrations are between 20 Hz and 20 kHz, they are audible to humans. These vibrations can be captured and converted into a digital signal and then used in audio signal processing on computers. They are typically captured by a microphone, after which the continuous signal is sampled at discrete samples. A typical sample rate is 44.1 kHz, which means that the amplitude of the incoming audio signal is measured 44,100 times per second. Note that this is around twice the maximum human hearing frequency. A sampled recording of someone saying "hello world" is plotted in the following diagram:



Speech signal of someone saying "hello world" in the time domain

Preprocessing

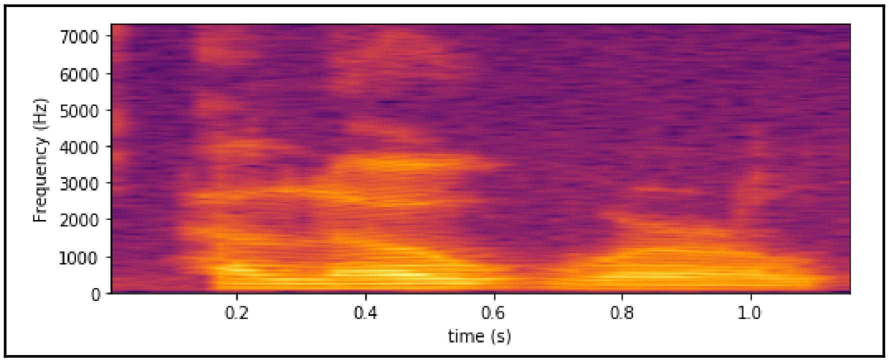
The recording of the audio signal in the preceding diagram was recorded over 1.2 seconds. To digitize the audio, it is sampled 44,100 times per second (44.1 kHz). This means that roughly 50,000 amplitude samples were taken for this 1.2-second audio signal.

For only a small example, these are a lot of points over the time dimension. To reduce the size of the input data, these audio signals are typically preprocessed to reduce the number of time steps before feeding them into speech recognition algorithms. A typical transformation transforms a signal to a spectrogram, which is a representation of how the frequencies in the signal change over time.

This spectral transformation is done by dividing the time signal in overlapping windows and taking the Fourier transform of each of these windows. The Fourier transform decomposes a signal over time into frequencies that make up the signal (`https://pdfs.semanticscholar.org/fe79/085198a13f7bd7ee95393dcb82e715537add.pdf`). The resulting frequency responses are compressed into fixed frequency bins. This array of frequency bins is also known as filter banks. A filter bank is a collection of filters that separate out the signal in multiple frequency bands.

Let's say that the previous "hello world" recording is divided into overlapping windows of 25 ms with a stride of 10 ms. The resulting windows are then transformed into a frequency space with the help of a windowed Fourier transform. This means that the amplitude information for each time step is transformed into amplitude information for each frequency. The final frequencies are mapped to 40 frequency bins according to a logarithmic scale, also known as the Mel scale. The resulting filter bank spectrogram is shown in the following diagram. This transformation resulted in reducing the time dimension from 50,000 to 118 samples, where each sample is a vector of size 40. We can use these vectors as input to our speech recognition model.

Following is a Mel spectrum diagram of the speech signal we introduced in section *Speech as input data*:



Mel spectrum of speech signal from the diagram in the *Speech as input data* section.

Especially in older speech recognition systems, these Mel-scale filter banks are even more processed by decorrelation to remove linear dependencies. Typically, this is done by taking a **discrete cosine transform (DCT)** of the logarithm of the filter banks. This DCT is a variant of the Fourier transform. This signal transformation is also known as **Mel Frequency Cepstral Coefficients (MFCC)**.

More recently, deep learning methods, such as convolutional neural networks, have learned some of these preprocessing steps (`https://arxiv.org/abs/1804.09298`).

Acoustic model

In speech recognition, we want to output the words being spoken as text. We can do this by learning a time-dependent model that takes in a sequence of audio features, as described in the previous section, and outputs a sequential distribution of possible words being spoken. This model is called the acoustic model.

The acoustic model tries to model the likelihood that a sequence of audio features was generated by a sequence of words or phonemes: *P (audio features | words) = P (audio features | phonemes) * P (phonemes | words)*.

A typical speech recognition acoustic model, before deep learning became popular, would use **hidden Markov models** (**HMMs**) to model the temporal variability of speech signals (`http://mi.eng.cam.ac.uk/~mjfg/mjfg_NOW.pdf` and `http://www.cs.ubc.ca/~murphyk/Bayes/rabiner.pdf`). Each HMM state emits a mixture of Gaussians to model the spectral features of the audio signal. The emitted Gaussians form a **Gaussian mixture model** (**GMM**), and they determine how well each HMM state fits in a short window of acoustic features. HMMs are used to model the sequential structure of data, while GMMs model the local structure of the signal.

The HMM assumes that successive frames are independent given the hidden state of the HMM. Because of this strong conditional independence assumption, the acoustic features are typically decorrelated.

One way to improve the speech recognition pipeline is to replace GMMs with deep networks.

Recurrent neural networks

This section describes how RNNs can be used to model sequential data. The problem with the straightforward application of RNNs on speech recognition is that the labels of the training data need to be perfectly aligned with the input. If the data isn't aligned well, then the input to output mapping will contain too much noise for the network to learn anything. Some early attempts try to model the sequential context of the acoustic features by using hybrid RNN-HMM models, where the RNNs would model the emission probabilities of the HMM models, much in the same way that DBNs are used (`http://www.cstr.ed.ac.uk/downloads/publications/1996/rnn4csr96.pdf`).

Later experiments tried to train LSTMs to output the posterior probability of the phonemes at a given frame (`https://www.cs.toronto.edu/~graves/nn_2005.pdf`).

The next step in speech recognition would be to get rid of the necessity of having aligned labeled data and removing the need for hybrid HMM models.

CTC

Standard RNN objective functions are defined independently for each sequence step; each step outputs its own independent label classification. This means that training data must be perfectly aligned with the target labels. However, a global objective function that maximizes the probability of a full correct labeling can be devised. The idea is to interpret the network outputs as a conditional probability distribution over all possible labeling sequences, given the full input sequence. The network can then be used as a classifier by searching for the most probable labeling, given the input sequence.

Connectionist Temporal Classification (**CTC**) is an objective function that defines a distribution over all the alignments with all the output sequences (`ftp://ftp.idsia.ch/pub/juergen/icml2006.pdf`). It tries to optimize the overall edit distance between the output sequence and the target sequence. This edit distance is the minimum number of insertions, substitutions, and deletions required to change the output labeling to target labeling.

A CTC network has a softmax output layer for each step. This softmax function outputs label distributions for each possible label plus an extra blank symbol (∅). This extra blank symbol represents that there is no relevant label at that time step. The CTC network will thus output label predictions at any point in the input sequence. The output is then translated into a sequence label by removing all the blanks and repeated labels from the paths. This corresponds to outputting a new label when the network switches from predicting no label to predicting a label or from predicting one label to another. For example, "∅aa∅ab∅∅" gets translated into "aab." In effect, only the overall sequence of labels has to be correct, thus removing the need for aligned data.

Doing this reduction means that multiple output sequences can be reduced to the same output labeling. To find the most likely output labeling, we have to add all the paths that correspond to that labeling. The task of searching for this most probable output labeling is known as decoding.

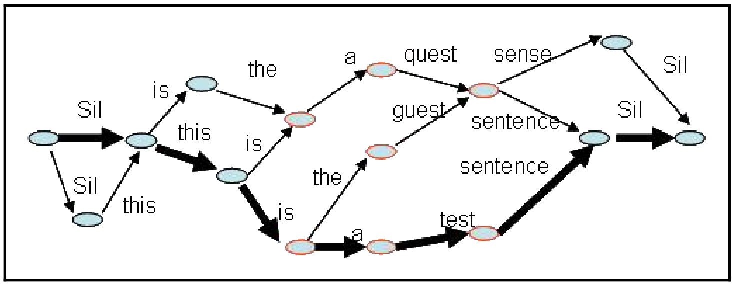
An example of such a labeling in speech recognition could be outputting a sequence of phonemes, given a sequence of acoustic features. The CTC objective's function, built on top of an LSTM, can remove the need for using HMMs to model temporal variability (`https://arxiv.org/pdf/1303.5778.pdf` and `https://arxiv.org/pdf/1512.02595v1.pdf`). An alternative to using the CTC sequence to sequence a model is an attention-based model.

Decoding

The process of obtaining the actual words from the audio features and phoneme distribution is known as decoding. Once we model the phoneme distribution with the acoustic model and train a language model, we can combine it with a pronunciation dictionary to get a probability function of words over audio features:

*P (words | audio features) = P (audio features | phonemes) * P (phonemes | words) * P (words)*

This probability function doesn't give us the final transcript yet; we still need to perform a search over the distribution of the word sequence to find the most likely transcription. This search process is called decoding. All possible paths of decoding can be illustrated in a lattice data structure (source: `https://www.isip.piconepress.com/projects/speech/software/legacy/lattice_tools/`):



A pruned word lattice

The most likely word sequence, given a sequence of audio features, is found by searching through all the possible word sequences (`http://mi.eng.cam.ac.uk/~mjfg/mjfg_NOW.pdf`). A popular search algorithm based on dynamic programming that guarantees it could find the most likely sequence is the Viterbi algorithm (`http://members.cbio.mines-paristech.fr/~jvert/svn/bibli/local/Forney1973Viterbi.pdf`). It is a breadth-first search that is mostly associated with finding the most likely sequence of states in an HMM.

For large vocabulary speech recognition, the Viterbi algorithm becomes intractable for practical use. So, in practice, heuristic search algorithms, such as beam search, are used to try and find the most likely sequence. The beam search heuristic only keeps the *n* best solutions during the search and assumes that all the rest don't lead to the most likely sequence.

Many different decoding algorithms exist (`http://www.cs.cmu.edu/afs/cs/user/tbergkir/www/11711fa16/aubert_asr_decoding.pdf`), and the problem of finding the best transcription from the probability function is mostly seen as an unsolved problem.

End-to-end models

We want to conclude this chapter by mentioning end-to-end techniques. Deep learning methods, such as CTC (`http://www.jmlr.org/proceedings/papers/v32/graves14.pdf` and `https://arxiv.org/abs/1512.02595`) and attention models (`https://arxiv.org/abs/1508.01211`) have allowed us to learn the full speech recognition pipeline in an end-to-end fashion. They do so without modeling phonemes explicitly. This means that these end-to-end models will learn acoustic and language models in one single model and directly output a distribution over words. These models illustrate the power of deep learning by combining everything in one model; with this, the model becomes conceptually easier to understand.

Summary

In this chapter, we discussed recurrent neural networks, how to train them, the training problems unique to RNNs, and how to solve those problems with LSTM and GRU. We described the task of language modeling and how RNNs help us solve some of the difficulties in modeling languages. Then, we put this all together in the form of a practical example on how to train a character-level language model to generate text based on Leo Tolstoy's *War and Peace*. Next, we introduced seq2seq models and the attention mechanism. Finally, we gave a brief overview of how to apply deep learning, and especially RNNs, to the problem of speech recognition.

In the next two chapters, we'll discuss how to teach a computer-controlled agent to navigate a physical or virtual environment with the help of reinforcement learning. Thanks to deep neural networks, this exciting ML area has seen some great improvements over the last few years.

8
Reinforcement Learning Theory

You may have read sci-fi novels from the 50s and 60s; they are full of visions of what life in the 21st century would look like. These stories imagined a world of people with personal jet packs, underwater cities, intergalactic travel, flying cars, and truly intelligent robots capable of independent thought. The 21st century has arrived now; sadly, we are not going to get those flying cars, but thanks to deep learning, we may get the robot.

In `Chapter 9`, *Deep Reinforcement Learning for Games*, and `Chapter 10`, *Deep Learning in autonomous Vehicles*, we'll talk about **Reinforcement learning** (**RL**) – a way to make machines interact with an environment, similar to the way we people interact with the physical world. As with many of the algorithms discussed so far, RL is not a new concept. But, recently, the field has seen something of a resurgence, in no small part thanks to the successes of deep learning. Indeed, we'll later see how integrating deep networks in RL frameworks can produce great results. In this section, we'll talk about the main paradigms and algorithms of RL. Then, we'll see how to combine them with deep networks to teach the computer to navigate a dynamic environment, such as a computer game. Games act as a great playing field for testing RL algorithms. They give us an environment of large, but manageable, possibilities. This is unlike the physical world, where even simple a task, such as getting a robot arm to pick up objects, requires analyzing huge amounts of sensory data and controlling many continuous-response commands for the arm's movement. Furthermore, we can create and simulate different training and evaluation scenarios more easily in a virtual environment, compared to a physical one.

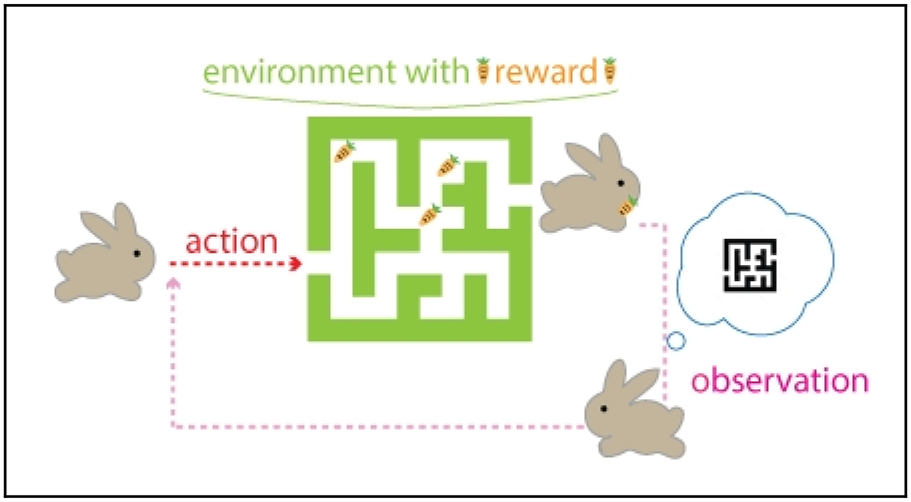
When it comes to computer games, we know that humans can learn to play a game just from the pixels visible on the screen and minimal instructions. If we input the same pixels plus an objective into a computer agent, we know we have a solvable problem, given the right algorithm. This is why so many researchers are looking at games as a great place to start developing true AI self-learning machines that can operate independently of humans. Also, if you like games, it's lots of fun.

In this chapter, we will cover the following topics:

- RL paradigms
- RL as a Markov decision process
- Finding optimal policies with Dynamic Programming
- Monte Carlo methods
- Temporal difference methods
- Value function approximation
- Experience replay
- Q-learning in action

RL paradigms

In this section, we'll talk about the main paradigms of RL. We first mentioned some of them in `Chapter 1`, *Machine Learning: an Introduction*, but it's worth discussing them here to refresh our memory and for the sake of completeness. To help us with this task, we'll use a maze game as an example. The maze is represented by a rectangular grid, where grid cells with a value of 0 represent the walls, and the cells with a value of 1 are the paths. Some locations contain intermediate rewards. An agent in the maze can use the paths to move between locations. Its objective is to navigate its way to the other end of the maze and to get the largest possible reward while doing so. The following is a diagram describing the basic principles of how RL works:



Reinforcement learning scenario

Here are some elements of an RL system:

- **Agent**: The entity for which we are trying to learn actions. In the game, this is the player who tries to find their way through the maze.
- **Environment**: The world in which the agent operates. Here, this is the maze (grid) itself.
- **State**: All of the information available to the agent about its current environment. In a maze, the state is simply the agent's position. In a chess game, the state would be the positions of all the pieces on the board.
- **Action**: A possible response, or set of responses, an agent can take. In a maze, the action might be the direction that the agent chooses when at a crossroad (up, down, left, or right). After each action, the environment will change its state and then provide feedback to the agent.
- **Reward**: The feedback that the agent gets from the environment after each action. It might be the exit square or the carrots in the image that the agent is trying to collect. Some mazes may also have traps that give a negative reward, which the agent should try to avoid. The agent's main objective is to maximize the total return (accumulated rewards) in the long run.

- **Policy**: Determines what actions the agent will take, given the current state. In the context of deep learning, we can train a neural network to make these decisions. In the course of the training, the agent will try to modify its policy to make better decisions. The task of finding the optimal policy is called policy improvement (or **control**) and is one of the major RL tasks.
- **Value function**: Determines what is good for the agent in the long run (unlike the immediate reward). That is, when we apply the value function over a given state, it will tell us the total return we can expect in the future, if we start from that state. The agent's policy will take into consideration the value – and, to a lesser extent, the reward – when deciding what action to take. The task of finding the value function is called **prediction** (also known as policy evaluation) and is the other major RL task.

Before we continue, we'll mention that this chapter was partially inspired by the excellent book *Reinforcement Learning, Second Edition* (`http://incompleteideas.net/book/the-book-2nd.html`), by Richard S. Sutton and Andrew G. Barto.

Differences between RL and other ML approaches

One of the most important distinctions between RL and other **machine learning** (**ML**) approaches is that in RL we have delayed rewards. That is, the agent might have to take a number of actions before the environment provides any reward signal. For example, in the maze game, the reward might come only at the end, when the maze exit square is reached. Therefore, when evaluating an action, the agent has to consider the problem in its entirety and not just the immediate consequences. This is unlike supervised learning, where the algorithm receives some kind of feedback (such as a label) for each training sample and has no knowledge of (or interest in) the end goal. The various RL system elements we just defined provide a mechanism for autonomous decision making in the absence of immediate rewards. But with decision-making power and no immediate feedback, the agent has to maintain a fine balance between exploitation (following the existing policy) and exploration (a trial-and-error approach in the hopes of finding a better policy).

Types of RL algorithms

We can use different classification of RL algorithms, depending on several factors. In this section, we'll outline some of the algorithms.

First, we'll divide RL algorithms based on the nature of the value function. We can identify two main types:

- **Tabular solutions**: The number of possible states and actions is small enough that we can represent the value function as a table (array) and the agent is fully familiar with the environment. One such scenario is the maze example, where the whole maze is stored in a table and the maze itself is not too big. With tabular solutions, we can often find the true optimal value function and optimal policy.
- **Approximate solutions**: The state and action spaces could be arbitrarily large. Imagine that we have to train an agent to play a computer game by just looking at the rendered game images on the screen. There is nothing unusual in that – after all, this is how we humans learn to play games. However, the number of possible rendered images is great. In such situations, it's impossible to know all states and actions, and it's not possible to have a tabular value function. On top of that, the large number of possible states means that the agent will inevitably run into situations it has never seen before. The way to solve a problem such as this is to find an approximation of the value function, which can also generalize over the unseen data. Fortunately, deep neural networks have proven to be good candidates for that role.

In the next sections, we'll discuss tabular RL solutions. We'll later see that we can extend them to include the more complex approximation scenario.

Types of RL agents

We have different types of RL agents:

- **Value-based agents**: These store the value function and base their decisions on it. Such an agent will decide which action to take based on the value of the states, where the action leads. These agents don't use a policy.
- **Policy-based agents**: These use only the policy, and not the value function, when deciding what action to take.
- **Actor-critic agents**: Use both the value function and the policy to make decisions.
- **Model-based agents**: These include a model of the environment. Given a state and an action, the agent can use the model as a simulation of the real environment to predict the next state and reward. In other words, the model allows the agent to **plan** its future actions.
- **Model-free agents**: These don't have an internal model of the environment, and learn the policy with a trial-and-error approach. Model-free agents learn to take their future actions.

Model-based and model-free agents can be value-based, policy-based, or actor-critic. RL agents that use policy can be further classified, as follows:

- **On-policy**: The agent takes actions based on the current policy.
- **Off-policy**: The agent bases its actions on a **behavior policy**, while it tries to optimize another **target policy**. Don't worry if you don't understand this yet, it will become clearer as we progress.

RL as a Markov decision process

A **Markov decision process** (**MDP**) is a mathematical framework for modeling decisions. We can use it to describe the RL problem. We'll assume that we work with a full knowledge of the environment. An MDP provides a formal definition of the properties we defined in the previous section (and adds some new ones):

- $\mathcal{S}$ is the finite set of all possible environment states, and s_t is the state at time t.
- $\mathcal{A}$ is the set of all possible actions, and a_t is the action at time t.

- $\mathcal{P}$ is the dynamics of the environment (also known as transition probabilities matrix). It defines the conditional probability of transitioning to a new state, *s'*, given the existing state, *s*, and an action, *a* (for all states and actions):

$$\mathcal{P}^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$$

  We have transition probabilities between the states, because MDP is stochastic (it includes randomness). These probabilities represent the model of the environment – that is, how it will likely change given its current state and an action, *a*. If the process were deterministic, we wouldn't need them. Model-based agents have an internal representation of $\mathcal{P}_,$ which they use to predict the results of their actions.

  We should note that the new state depends only on the current state and not on any previous states. In other words, the current state completely characterizes the total state of the environment, which makes the MDP a memoryless process. This feature of the MDP is called the **Markov property**.

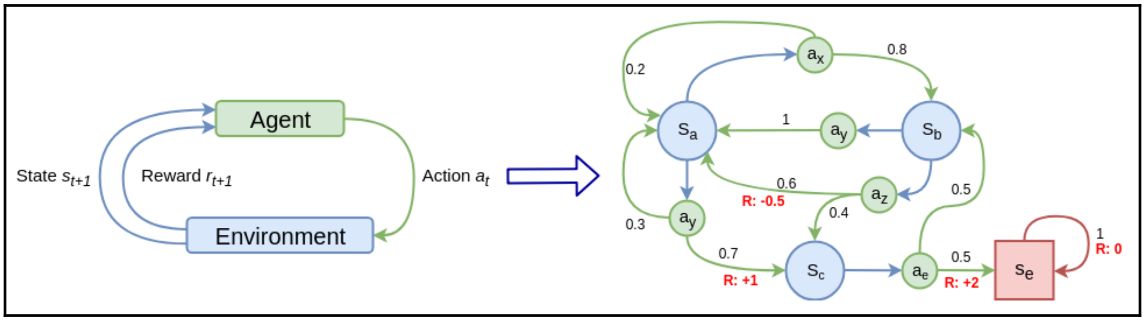
- $\mathcal{R}$ is the reward function. It describes the reward, the agent would receive when it takes action *a* and transitions from *s* -> *s'*:

$$\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1} | s_{t+1} = s', s_t = s, a_t = a]$$

  We need the expectation, $\mathbb{E}$, because the environment described by the MDP is stochastic and the transitions between different states are described by the transition probabilities. In other words, we cannot say with certainty in what new state (and hence which reward) we'll end up when the agent takes an action. Instead, we can reasonably expect what the reward might be. We'll denote the reward at time step *t* with r_t.

- γ is the discount factor. It's a value in the [0:1] range and it determines how much the algorithm values the immediate rewards, as opposed to the future rewards.

We can see an example MDP in the following diagram:



Left: The agent takes an action and receives the new state and reward. Right: An example of a simple MDP with four states (s_a, s_b, s_c, and a terminal state, s_e) and four actions (a_x, a_y, a_z, and a_e). Each action has a transition probability, but only some actions have rewards

To understand an MDP better, let's see how to represent the maze game in a graph similar to the one in the preceding diagram. We'll have as many states, s_a, s_b, s_c, ..., as the number of maze grid cells. Each state will be associated with four actions (up, down, left and right). However, the maze walls will make it impossible to take certain actions to or from some of the states, therefore they will have a probability of 0. Since this is a deterministic model, the rest of the state/action pairs will have a probability of 1. That is, when the agent moves in some direction, it is certain that it will end up in the corresponding neighboring maze position.

We'll now describe the step-by-step execution of one MDP episode (an episode might be one game of chess, for example):

1. The episode starts with the initial state, s_0, at time time $t=0$ and ends at $t=T$ when we reach the terminal state.
2. Repeat until we reach the terminal state:
   1. The agent takes the a_t action.
   2. The environment samples a reward, r_{t+1}, and a new state, s_{t+1}, based on the transitional probabilities, $\mathcal{P}_{s_t s_{t+1}}^{a_t} = Pr(s_{t+1}|s_t, a_t)$.
   3. The agent receives the new state, s_{t+1}, and reward, r_{t+1}.

The MDP together with the agent will produce the following sequence (or trajectory):

$$s_0,\ a_0,\ r_1,\ s_1,\ a_1,\ r_2,\ s_2,\ a_2,\ r_3,\ \ldots\ a_{T-1},\ r_T,\ s_T$$

Some RL problems might be continuous, and not episodic, such as balancing a pole by shifting the bottom end of the pole to the left or right. The goal is to keep the pole upright indefinitely. We won't go into details about this scenario, but we should note that the algorithm we described for episodic tasks can be used for continuous tasks too ($T = \infty$). Still, we need to consider that the agent's goal is to maximize the total return (the sum of all future rewards) in the long run. In a continuous task, this is infinity. We can solve this with the help of the discount factor, which lies in the [0:1] range. The formula for the **discounted return** at time step *t* is as follows:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{j=0}^{T} \gamma^j r_{t+j+1}$$

Although the sum is still infinite, if $\gamma<0$, then G_t will have a finite value. If $\gamma=0$, the agent is only interested in the immediate reward and discards the long-term return. Conversely, if $\gamma=1$, the agent will consider all future rewards equal to the immediate reward. For reasons we'll see later in the chapter, we can rewrite this equation with a recursive relationship:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots + \gamma^{T-1} r_T \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \ldots + \gamma^{T-2} r_T) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned}$$

Next, let's discuss the value function and the policy. The value function estimates the cumulative future rewards of a given state. However, the rewards depend on the future actions of the agent, which are determined by the agent's policy (denoted by π). In formal terms, the policy maps the state, *s*, to a probability of selecting each possible action, *a*, starting from *s*. The value function and the policy are inextricably linked.

We can define two types of value functions:

- **State-value function** $v_\pi(s)$: Describes the expected returns starting from the s_t state and then following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | s = s_t] = \mathbb{E}_\pi[\sum_{j=0}^{T} \gamma^j r_{t+j+1} | s = s_t]$$

$\mathbb{E}_\pi[\cdot]$ is the expected value of the total return, G_t, at time step t. We use expectation, because both the environment transition function and the policy (depends on the type of policy) might act in a stochastic way. On the one hand, when the agent takes an action, a, the environment might end up in any number of different states, depending on the transition probabilities. On the other hand, we may choose a policy that will act in a random way given equal conditions. Therefore, we can only approximate what the value of the s_t state is.

- **Action-value function $q_\pi(s,a)$ or q-function**: This describes the expected return starting from s, then taking action a, and following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|s = s_t, s = a_t] = \mathbb{E}_\pi[\sum_{j=0}^{T}\gamma^j r_{t+j+1}|s = s_t, s = a_t]$$

The action-value definition follows the same assumptions as the state-value function.

Next, let's denote with $\pi(a|s)$ the probability that a policy, π, selects an action, a, given a current state, s. Then, the following equation between the state-value and action-value functions is true:

$$v_\pi(s) = \sum_a \pi(a|s)\ q_\pi(s,a)$$

Intuitively, we can see that the state-value function is equivalent to the sum of the action-value functions of all outgoing (from s) actions, a, multiplied by the policy probability of selecting each action. Note that the sum of probabilities of all outbound actions from s is $\sum_a \pi(a|s) = 1$.

Bellman equations

The Bellman equations are named after Richard Bellman (`https://en.wikipedia.org/wiki/Richard_E._Bellman`), who also introduced the DP method. In DP, we can find the optimal solution of a complex problem by breaking it down into simpler, recursive sub-problems and finding their optimal solutions. For example, we can find the k-th Fibonacci number using a bottom-up dynamic method.

Although there is a strong chance that you are already familiar with this solution, we'll still include it for the sake of completeness (it's just a few sentences). We'll start with the first and second Fibonacci numbers (0 and 1). The third number is just the sum of the first two. Then, the fourth number is the sum of the third and the second, but we already know both of these. Therefore, finding the fourth number is trivial. By just storing the last two numbers, we can compute any number of the sequence in O(n) time complexity and O(1) memory complexity. We did this by splitting the problem into smaller sub-problems. We won't focus further on DP itself, but if you want to learn more, detailed tutorials are widely available.

The Bellman equation describes the relationship between the sub-problems and the main problem in DP. We can also apply it for the MDP. We already saw that we can define the discounted return, G_t, in recursive terms. Let's now see how to recursively define the state-value function.

We'll start by rewriting it with the recursive definition of G_t:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s = s_t] = \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|s = s_t]$$

Let's analyze its two components, starting with the expectation for the immediate reward, r_{t+1}:

$$\mathbb{E}_\pi[r_{t+1}|s = s_t] = \sum_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'}$$

Although it might seem complex, it's not difficult to understand. We are already familiar with $\sum_a \pi(a|s) = 1$. Next, $\mathcal{P}^a_{ss'}$ is the transition probability and $\mathcal{R}^a_{ss'}$ is the expected award the environment will sample, when transitioning from s -> s' via a. Finally, we sum over all possible actions, a, and all possible resulting states, s'. To summarize, the expected immediate reward is the sum over the product of the probabilities of the policy to select each action, a, the transition probabilities, and the expected rewards for the transitions.

Next, let's analyze the second component:

$$\mathbb{E}_\pi[\gamma G_{t+1}|s = s_t]$$
$$= \mathbb{E}_\pi[\gamma \sum_{j=0}^{T} \gamma^k r_{t+j+2}|s = s_t]$$
$$= \sum_a \pi(a|s) \sum_{s'} \mathcal{P}_{ss'}^a \ \gamma \mathbb{E}_\pi[\sum_{j=0}^{T} \gamma^j r_{t+j+2}|s' = s_{t+1}]$$

This equation is the same as the preceding one, but this time we are using the total discounted return, instead of the immediate return.

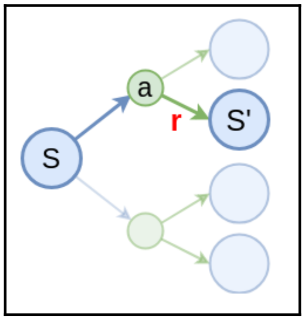
Knowing this, we can rewrite the state-value equation with the new definitions:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \mathcal{P}_{ss'}^a \ [\mathcal{R}_{ss'}^a + \gamma \mathbb{E}_\pi[\sum_{j=0}^{T} \gamma^j r_{t+j+2}|s' = s_{t+1}]]$$

Finally, we can see that the innermost expectation is equal to $v_\pi(s' = s_{t+1})$. Thus, we can define the Bellman equation for the state-value function:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \mathcal{P}_{ss'}^a \ [\mathcal{R}_{ss'}^a + \gamma v_\pi(s')]$$

Here is a diagram of the state-value function:



The state-value function

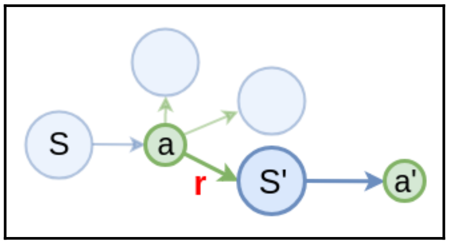
We can intuitively see that it recursively breaks down the value computation into an immediate expected reward from the next state/action pairs (a sum over the policy probabilities) and discounted return for all states, following the current one.

The Bellman equation for the action-value function is as follows:

$$q_\pi(s,a) = \mathbb{E}_\pi[\sum_{j=0}^{T} \gamma^j r_{t+j+1} | s_t = s, a_t = a]$$
$$= \sum_{s'} \mathcal{P}^a_{ss'} \ [\mathcal{R}^a_{ss'} + \gamma \sum_{a'} \pi(a'|s') \ q_\pi(s', a')]$$
$$= \sum_{s'} \mathcal{P}^a_{ss'} \ [\mathcal{R}^a_{ss'} + \gamma v_\pi(s')]$$

We got the second form of the equation because of the equivalency we introduced at the end of the preceding section.

Here is a diagram of the action-value function:



The action-value function

The Bellman equations are important, because they give us the ability to describe the value of a state, s_t, with the value of the s_{t+1} state. That is, if we know the value of s_{t+1} ,we can easily compute the value of s_t, and with an iterative approach, we can calculate the values of all states.

Optimal policies and value functions

The goal of the agent is to maximize the total cumulative reward in the long run. The policy, which maximizes the total cumulative reward is called the optimal policy and is denoted with denoted with π_*. There could be different optimal policies, but they all share the same value functions (optimal value functions).

We'll denote the state-value and action-value functions with respect to the optimal policy, π_*, with the following:

$$v_*(s) = \max_\pi v_\pi(s)$$
$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Let's expand on the optimal action-value function. First, we start with a state, *s*, and an action, *a*. Next, to satisfy the optimal condition, we have to follow the optimal policy, π_* . Therefore, we can write $q_*(s,a)$ in terms of $v_*(s)$:

$$q_*(s,a) = \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1}) | s_t = s, a_t = a]$$

Furthermore, the state-value function of a state, *s*, under the optimal policy must equal the expected return for the best action starting from that state. This can be expressed as follows:

$$v_*(s) = \max_a q_*(s,a)$$

Based on that, we can define the Bellman optimality equation for $v_*(s)$:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} \ [\mathcal{R}^a_{ss'} + \gamma v_*(s')] \end{aligned}$$

Where $s_{t+1} = s'$. Then, we can define the Bellman optimality equation for $q_*(s, a)$:

$$\begin{aligned} q_*(s,a) &= \max_a \mathbb{E}[r_{t+1} + \gamma q_*(s_{t+1}, a') | s_t = s, a_t = a] \\ &= \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_a q_*(s', a') \right] \end{aligned}$$

Where $s_{t+1} = s'$ and $a' = a_{t+1}$.

We wrote a bunch of equations, but why exactly are they useful? As we demonstrated, we can use the Bellman equations to express the value of one state by the value of another state. The Bellman optimality equations build upon that and provide the base for iterative approaches of finding the optimal policy (which maximizes the expected reward in the long run). We'll talk about these algorithms in the next sections.

Finding optimal policies with Dynamic Programming

Dynamic Programming (**DP**) is a base for many RL algorithms. The main paradigm of DP algorithms is to use the state- and action-value functions as tools to find the optimal policy, given a fully-known model of the environment. In this section, we'll see how to do that.

Policy evaluation

We'll start with policy evaluation, or how to compute the state-value function, v_π , given a specific policy, π. This task is also known as **prediction**. As a reminder, we'll assume that the state-value function is a table. We'll implement policy evaluation using the state-value Bellman equation we defined in the *Bellman equations* section. Let's start:

1. Input the following:
    - The policy, π.
    - A small threshold value, θ, which is used to assess when to stop.

2. Initialize the following:
    - The Δ variable with 0. We'll use it in combination with θ to assess whether to stop.
    - The v_π table with some value for all states.
3. Repeat until $\Delta < \theta$:
    - $\Delta = 0$
    - For each state *s_i* in $\mathcal{S}$, do the following:
        1. Extract the expected total return, $v_{s_i} = v_\pi(s_i)$, for s_i.
        2. Update the discounted return of s_i with the Bellman equation:

$$v_\pi(s_i) = \sum_a \pi(a|s_i) \sum_{s'_i} \mathcal{P}^a_{s_i s'_i} \left[\mathcal{R}^a_{s_i s'_i} + \gamma v_\pi(s'_i)\right]$$

Let's analyze this formula to make it clear. Given a current state, s_i, we iterate over all possible actions, *a*, for which the policy, π, gives a non-zero probability. Then, for each of these actions, we calculate the sum of the reward, $\mathcal{R}^a_{s_i s'_i}$ (transition from the s_i -> s'_i state) and the discounted returns, $\gamma v_\pi(s'_i)$, of the new state, s'_i. We wrap all this in the usual probabilities (policy and transition) and the end result gives us the updated discounted return for the s_i state. To summarize, we update the value of a state using the values of its neighboring states.

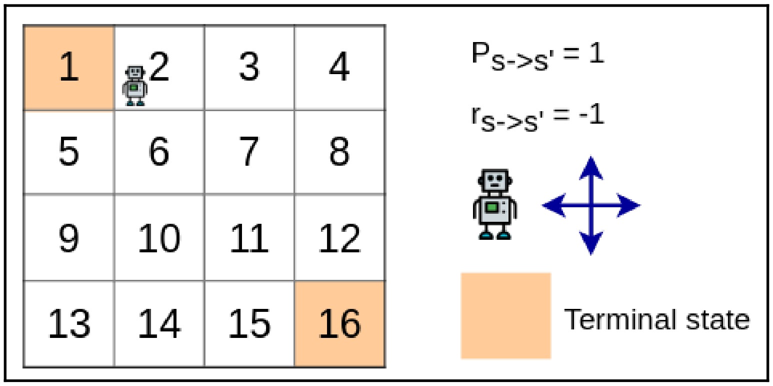
$$\Delta = max(\Delta, |v_{s_i} - v_\pi(s_i)|)$$

Policy evaluation example

To better understand this, let's use an example. Imagine that we have a simple robot, navigating a grid environment (this example is also known as gridworld). We'll assume that:

- The grid is size 4 x 4. It's very similar to the maze example we defined earlier, with the exception that it has no walls. The cells are numbered from 1 to 16, where cells 1 and 16 are terminal states.

- The robot can navigate up, down, left, or right to any of the neighboring states. Actions that take the robot off the grid leave it in its current state (but the reward is still received).
- The environment is deterministic – that is, the transition probability of moving to the corresponding neighbor state when taking an action is always 1. For example, if the robot takes the "up" action, it will move to the upper cell with a probability of 1.
- The transition between any two states carries a negative reward of -1. The only exception is when the transition starts from any of the two terminal states, when the reward is 0.
- We'll use a discount factor of 1.
- The robot uses a simple policy, which gives an equal probability of 0.25 for each of the four actions for any grid cell.
- The values for all states are initialized with 0:



The gridworld environment

Given these conditions, let's assume that we are in the first iteration of the policy-evaluation procedure and we want to update the value of cell 2:

$$v_\pi(s_2) = \sum_{a}^{\uparrow\downarrow\leftarrow\rightarrow} \pi(a|s_2) \sum_{s'_i}^{2,6,1,3} \mathcal{P}^a_{s_2 s'_i} \left[\mathcal{R}^a_{s_2 s'_i} + \gamma v_\pi(s'_i)\right]$$

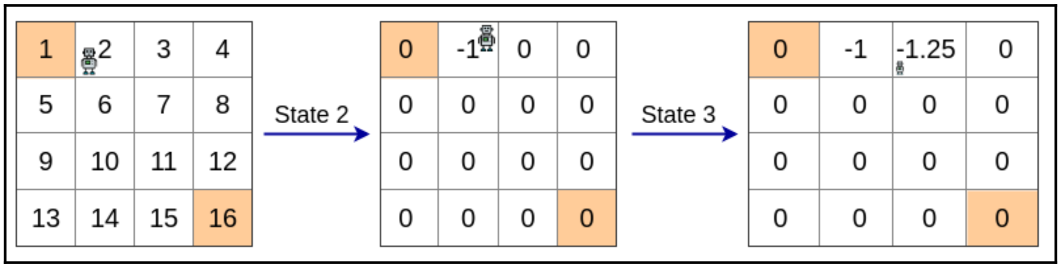
First, we know that $\pi(a|s_2)$ for any action is 0.25. Next, we know that the transition probability, $\mathcal{P}^a_{s_2 s'_i}$, is always 1, even for going upward off the edge, when the robot will "transition" to its current state (2). The reward, $\mathcal{R}^a_{s_2 s'_i}$, is always -1, even for going in the terminal state. Finally, the expected return, $v_\pi(s'_i)$, of each neighbor state is 0, because this is the first iteration of the evaluation and the initial values are all zeros. Therefore, we have the following:

$$v_\pi(s_2) = \sum_a^{\uparrow\downarrow\leftarrow\rightarrow} 0.25 \sum_{s'_i}^{2,6,1,3} [-1+0] = (0.25*-1)+(0.25*-1)+(0.25*-1)+(0.25*-1) = -1$$

Next, let's compute the new expected return for the neighboring state 3. One of the scenarios is a transition from 3 -> 2. In this case, we'll use the newly-updated expected return, $v_\pi(s_2) = -1$, of state 2:

$$v_\pi(s_3) = \sum_a^{\uparrow\downarrow\leftarrow\rightarrow} 0.25 \sum_{s'_i}^{3,7,2,4} [-1+v_\pi(s'_i)] = (0.25*-1)+(0.25*-1)+(0.25*(-1-1))+(0.25*-1) = -1.25$$

In the following diagram, we can see the illustration of the two steps we just described:



The first two steps of the policy-evaluation procedure

We'll continue performing the same steps for the rest of the states. In the next iteration of the evaluation process, we'll repeat the same steps for each state, using the newly-updated value function.

Policy improvements

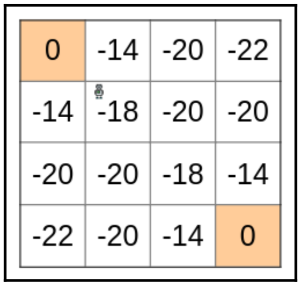
Once we're able to evaluate a policy, let's look at how to improve it. This task is also known as **control**. We'll assume that the policy is represented as a table, where the best actions are stored for each state (tabular solution). We'll also assume that we have an already-existing value function, v_π (the step described in the preceding section), and a policy, π. For each state, *s*, we'll do the following:

1. Assume that we take all possible actions starting from *s*. That also includes the action selected by the policy. Using the action-value Bellman equation, for each action we'll compute the expected returns if we take that action and continue following the policy, π, after that.
2. Compare the expected returns of the action, selected by the policy, to the expected returns for the rest of the actions. If some of the newly-computed expected returns are larger than the existing policy selection, we'll update the policy to take the new action every time we are in the *s* state. This approach is greedy, because we always take the max return. It is also relatively simple, because we are only doing a one-step look-ahead when evaluating the performance of the different actions.
3. Repeat the preceding steps until the policy selects the best action in all cases, that is, until it is no longer necessary to update it.

We can describe this algorithm with the following equation. We'll denote the updated policy with π' . The **arg max** symbolizes the comparison between the actions' expected returns:

$$\begin{aligned} \pi'(s) &= \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}[r_{t+1} + \gamma v(s')|s,a] \\ &= \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^a \ [\mathcal{R}_{ss'}^a + \gamma v_\pi(s')] \end{aligned}$$

To better understand policy improvement, let's use the gridworld example from the *Policy evaluation example* section. We'll assume that the policy evaluation has finished, and the result after all iterations is are :



0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0

Policy evaluation result with expected returns for each cell

Let us say that we want to improve the policy when the robot is located in cell 6 (displayed in the preceding diagram). The initial policy gives an equal probability of taking each of the four actions: `π(6) = [up: 0.25, down: 0.25, left: 0.25`, or `right: 0.25]`. After applying the action-value Bellman equation for each of the four actions, we'll get the following expected returns: `up: - 1 -14 = -15`, `down: -1 - 20 = -21`, `left: - 1 -14 = -15`, and `right: -1 - 20 = -21` (this is just the sum of the transition reward and the expected return of the next state). We can see that the up and left actions have better expected returns (-15), compared to bottom and right (-21). In this case, we can update our policy to `π(6) = [up: 0.5, down: 0, left: 0.5, right: 0]`.

Policy and value iterations

In this section, we'll put together everything we've learned so far and we'll combine policy evaluation and improvement in a single algorithm (so exciting!). Fortunately, the concepts are simple.

We'll start with policy iteration. It refers to alternating steps of policy evaluation and policy improvement until the process converges. Here is a sample diagram of the policy iteration steps:

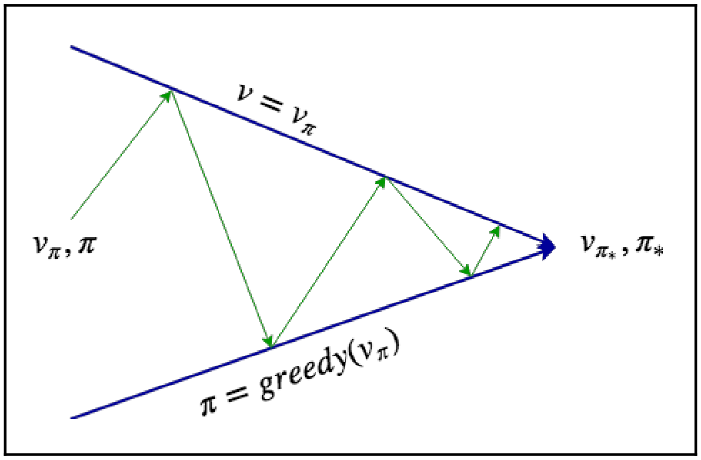
$$\begin{tikzcd}
\pi_0 \arrow[r, "\text{Eval}"] & v_{\pi_0} \arrow[r, "\text{Improve}"] & \pi_1 \arrow[r, "\text{Eval}"] & v_{\pi_1} \arrow[r, "\text{Improve}"] & \pi_2 \arrow[r, "\text{Eval}"] & \cdots\cdots \arrow[r, "\text{Improve}"] & \pi_* \arrow[r, "\text{Eval}"] & v_{\pi_*}
\end{tikzcd}$$

Policy iteration unfolded

Policy iteration has one disadvantage: it performs evaluation in each iteration. Evaluation itself is an iterative process, which might be time-consuming. It turns out that we can improve its performance by doing just a single iteration of policy evaluation, instead of waiting for the delta to fall below the threshold of θ. We can see that the Bellman equations for one evaluation and one improvement step is similar (with the exception of the max value in the improvement case). Because of this, we can combine the two steps for a better-optimized algorithm, called value iteration.

The evaluation and improvement steps have a friendly/adversarial relationship. On the one hand, if the policy is greedy with respect to the value function, it will usually invalidate that value function with respect to the policy. That's because the existing function will no longer represent the actions that would be taken by the updated policy. On the other hand, if we update the value function to better represent the policy, the policy will no longer be greedy with respect to the value function, because there will be no need to update it anymore. At the same time, the interaction between the two would converge toward the optimal policy and value function. A finite MDP has a finite number of policies, which means that the policy iteration will converge in a finite amount of time.

In the following diagram, we can see an illustration of the process in terms of convergence toward the optimal solution:



The interaction between policy evaluation and improvement

If you are interested in an example Python implementations of DP prediction and control, check out `https://github.com/dennybritz/reinforcement-learning/tree/master/DP`.

Monte Carlo methods

In this section, we'll describe our first algorithm, which does not require full knowledge of the environment (model-free): the **Monte Carlo** (**MC**) method (yay, I guess...). Here, the agent uses its own experience to find the optimal policy.

Policy evaluation

In the *Dynamic programming* section, we'll describe how to estimate the value function, $v_\pi(s)$, given a policy, π (planning). MC does this by playing full episodes, and then averaging the cumulative returns for each state over the different episodes.

Let's see how it works in the following steps:

1. Input the policy, π.
2. Initialize the following:
    - The v_π table with some value for all states
    - An empty list of `returns(s)` for each state, *s*
3. For a number of episodes, do the following:
    1. Generate a new episode, following policy π: $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots a_{T-1}, r_T, s_T$.
    2. Initialize the cumulative discounted return, `G = 0`.
    3. Iterate over each `t` step of the episode, starting from `T-1` and going to `0`:
        1. Update *G* with the reward at the *t+1* step: $G = G + \gamma r_{t+1}$.
        2. If the s_t state doesn't appear in any of the preceding episode steps $s_0, s_1, s_2 \ldots s_{t-1}$:
            1. Append the current value of *G* to the `returns(`s_t`)` list associated with s_t.
            2. Update the value function with the average of `returns(`s_t`)`: $v_\pi(s_t) = \text{average}(\text{returns}(s_t))$.

This MC variant is called **first-visit**, because if a state, *s*, appears multiple times in one episode, we'll only add *G* to `returns(s)` the first time that the state occurred in the episode trajectory. We'll ignore the other occurrences. Conversely, we can also add the discounted reward every time the state occurs in the episode. In this case, we call it **every-visit** MC. We can use the same pseudo-code as with first-visit, except that we'll remove the check of whether the state has already occurred. Both first-visit and every-visit MC converge to the real value function, v_π , as the number of occurrences of each state, *s*, approaches infinity. Unlike the DP policy evaluation, the MC method doesn't use the values of other states to compute the value of the current one, and instead only relies on its own experience to do this (the episodes).

Exploring starts policy improvement

MC policy improvement follows the same general pattern as DP. That is, we alternate evaluation and improvement steps until convergence. But because we don't have a model of the environment, it is better to estimate the action-value function, $q_\pi(s, a)$ (state-action pairs), instead of the state-value function. If we had a model, we could just follow a greedy policy and choose the combination of action/reward and next state value with the highest expected return (similar to DP). But here, the action values will be better for choosing new policy. Therefore, to find the optimal policy, we have to estimate $q_\pi(s, a)$. With MC, we can do this in the same way as we estimated the state-value function (preceding section). That is, we'll generate multiple episodes and then average the returns of each state/action pair. But if the policy is deterministic, every time the agent moves to a particular state, *s*, it will choose the same action, *a*. Because of this, there is a chance that some state/action pairs may never get visited and we won't be able to estimate $q_\pi(s, a)$ for them. One way to solve this is to ensure that each episode starts with a state/action pair (and not just with state), and each state/action pair has non-zero probability of starting an episode. As the number of episodes goes to infinity, every state/action pair will participate. This assumption is called **exploring starts** (**ES**).

Next, we'll describe first-visit MC ES. At each step of the algorithm, we'll estimate $q_\pi(s, a)$ for one state/action pair, and then we'll update the policy for the state, *s*, in a greedy way with respect to the action-value function:

1. Input the policy, π
2. Initialize the following:
    - The $q_\pi(s, a)$ table with some value for all state/action pairs
    - An empty `returns(s, a)` list for each state/action pair
3. For a number of episodes, do the following:
    1. Generate a new episode, following the policy, π: s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ... a_{T-1}, r_T, s_T.
    2. Initialize the cumulative discounted return, `G = 0`.
    3. Iterate over each `t` step of the episode, starting from `T-1` and going to `0`:
        1. Update *G* with the reward at the *t+1* step: $G = G + \gamma r_{t+1}$.
        2. If the pair (s_t, a_t) doesn't appear in any of the preceding episode steps s_0, a_0, s_1, a_1 ... s_{t-1}, a_t:
            1. Append the current value of *G* to the `returns(`s_t`, `a_t`)` list associated with (s_t, a_t).
            2. Update the value function with the average of `returns(`s_t`, `a_t`)`: $q_\pi(s_t, a_t) = \text{average}(\text{returns}(s_t, a_t))$.
            3. $\pi(s_t) = \arg\max_a q_\pi(s_t, a)$.

This algorithm is very similar to the one described in the preceding *Policy evaluation* section (but we use $q_\pi(s, a)$ instead of $v_\pi(s)$). Similarly, we have an every-visit version, which averages the discounted reward every time state/action pair occurs in the episode. Worth noting is the policy-improvement step. First, we calculate the new value of $q_\pi(s, a)$ for the *s* state. Then, we update the policy for that state, $\pi(s)$, in a greedy way by simply choosing the action (among all actions starting from *s*) with the maximum expected return, according to the action-value function.

Epsilon-greedy policy improvement

In the preceding section, we discussed that if we follow a **deterministic policy** (**DP**), we might not reach all state/action pairs. This would undermine our efforts to estimate the action-value function. We solved this problem with the exploring-starts assumption. But this assumption is unusual and it would be best to avoid it. In fact, the core of our problem is that we follow the policy blindly, which prevents us from exploring all possible state/action pairs. Can we solve this by introducing a different policy? Turns out it can (surprise!). In this section, we'll introduce MC control with a non-deterministic epsilon-greedy (ε-greedy) policy. The core idea is simple. Most of the time the ε-greedy policy behaves just such as the greedy policy we used so far. But sometimes, with a probability of ε, it will choose a random action, instead of the optimal one. In particular, all non-optimal actions starting from the *s* state can be selected with minimal probability, $\epsilon/|\mathcal{A}(s)|$, where $|A(s)|$is the number of actions for the *s* state. The optimal action (selected by the greedy policy) has a probability of $1 - \epsilon - \epsilon/|\mathcal{A}(s)|$ being selected. Here is the first-visit MC for the ε-greedy policy:

1. Input the policy, π.
2. Initialize the following:
    - The $q_\pi(s, a)$ table with some value for all state/action pairs
    - An empty `returns(s, a)` list for each state/action pair
3. For a number of episodes, do the following:
    1. Generate a new episode, following the policy, π: $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots a_{T-1}, r_T, s_T$.
    2. Initialize the cumulative discounted return `G = 0`.
    3. Iterate over each `t` step of the episode, starting from `T-1` and going to `0`:
        1. Update *G* with the reward at the *t+1* step: $G = G + \gamma r_{t+1}$.
        2. If the (s_t, a_t) pair doesn't appear in any of the preceding episode steps $s_0, a_0, s_1, a_1 \ldots s_{t-1}, a_t$:
            1. Append the current value of *G* to the `returns`(s_t, a_t) list associated with (s_t, a_t).
            2. $a_* = \arg\max_a q_\pi(s_t, a)$.
            3. For all actions a_i, starting from state *s*:

$$\pi(a_i, s_t) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s_t)| & \text{if } a_i = a_* \\ \epsilon/|\mathcal{A}(s_t)| & \text{if } a_i \neq a_* \end{cases}$$

Unlike the regular greedy policy, ε-greedy is non-deterministic (every action is selected with a probability), and we'll assign a probability for each action, a_i, starting from the s_t state.

If you're interested in an example Python implementation of MC prediction and control, check out `https://github.com/dennybritz/reinforcement-learning/tree/master/MC`.

Temporal difference methods

Temporal difference (**TD**) is a class of model-free RL methods. On the one hand, they can learn from the agent's experience, such as MC. On the other hand, they can estimate state values based on the values of other states, such as DP. As usual, we'll explore the policy evaluation and improvement tasks.

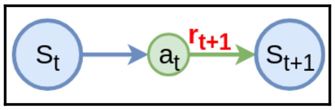
Policy evaluation

TD methods rely on their experience for policy evaluation. But unlike MC, they don't have to wait until the end of an episode. Instead, they can update the action-value function after each step of the episode. In its most basic form, a TD algorithm uses the following formula to perform a state-value update:

$$v(s_t) = v(s_t) + \alpha[r_{t+1} + \gamma v(s_{t+1}) - v(s_t)]$$

Where α is called step size (learning rate) and it's in the range of [0, 1]. Let's analyze this equation. We're going to update the value of the s_t state and we're following a policy, π, that has led the agent to transition from the s_t state to the s_{t+1} state. During the transition, the agent has received a reward, r_{t+1}. Think of $r_{t+1} + v(s_{t+1})$ as the label (target) value for $v(s_t)$.

We can assume the label is more accurate than $v(s_t)$, because it includes the reward, r_{t+1} (besides $v(s_{t+1})$), which was actually given by the environment. On the contrary, $v(s_t)$ is just an estimation? Compare this with the MC method, where the target value is the total discounted return, *G*, of the full episode. In other words, TD uses the estimated value (expected updates), while MC uses the real discounted return (sample updates). Next, $r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ is the difference between the label and the predicted value of the algorithm (known as the TD error), just such as with classification in neural networks. Finally, we'll update the value function using the step size, α, which determines how fast the value will change with each update. The process is very similar to the weight update rule for neural networks. The following diagram illustrates the MDP sequence of states:



This method is called **one-step TD**, or **TD(0)**. Other variants include n-step TD and TD(λ).

Let's see how TD(0) works:

1. Input the policy, π.
2. Initialize the v_π table with some value for all states.
3. Repeat for a number of episodes:
    1. Start new episode with an initial state of $s_{t=0}$.
    2. Repeat until the terminal state is reached:
        1. Select action a_t, using the policy, π, for the current state, s.
        2. Take the action a_t, transition to new state s_{t+1}, and observe reward r_{t+1}.
        3. Update the value function, $v(s_t) = v(s_t) + \alpha[r_{t+1} + \gamma v(s_{t+1}) - v(s_t)]$.
        4. $s_t = s_{t+1}$.

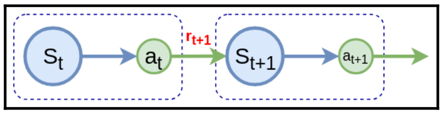
TD has several advantages over MC and DP. First, it's model-free, unlike DP. Next, it is an online method, where we update the value constantly. Compare this to MC, which performs an update only after the episode is finished, which could prove costly for long episodes. This also makes it possible to apply TD for continuous tasks (as opposed to periodic).

Control with Sarsa

Sarsa is an on-policy TD control method. Much such as MC control, we'll try to estimate the action-value function in order to find the optimal policy. We'll do this for the same reasons we outlined in the *Exploring starts policy improvement* section. But this time, we'll follow the blueprint outlined in the preceding section. That is, we'll iterate over multiple episodes and we'll update $q_\pi(s, a)$ online, after each step of an episode. We can represent this process with a formula, similar to the one in the preceding section, with the exception that it is for the action-value function:

$$q(s_t, a_t) = q(s_t, a_t) + \alpha[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)]$$

Where $q_\pi(s_{\text{terminal}}, a) = 0$ for each action of the terminal state. We start at the s_t state, take the a_t action (following the policy), transition to the next state, s_{t+1}, and then take the next action, a_{t+1} (again, following the policy). The following diagram illustrates this MDP sequence of state/action pairs:



The name of the method, Sarsa, comes from the fact that the five-element sequence of the episode's trajectory is s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}.

Next, let's see how Sarsa works (hint: it's similar to TD(0)):

1. Input the policy, π.
2. Initialize the $q_\pi(s, a)$ table with some value for all state/action pairs.
3. Repeat the following for a number of episodes:
    1. Start new episode with initial state/action pair, $s_{t=0}$, $a_{t=0}$.
    2. Repeat the following until the terminal state is reached:
        1. Take the action a_t and transition it to a new state s_{t+1}, and observe reward r_{t+1}.
        2. Select next action, a_{t+1}, following the policy, π (for example, ε-greedy).
        3. Update the action-value function, $q(s_t, a_t) = q(s_t, a_t) + \alpha[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)]$.
        4. $s_t = s_{t+1}$, $a_t = a_{t+1}$.

If you are interested in an example Python implementation of Sarsa, check out `https://github.com/dennybritz/reinforcement-learning/tree/master/TD`.

Control with Q-learning

Q-learning is an off-policy TD control method. It was developed by Watkins, C.J.C.H. (`http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf`) in 1989 (in 2019, Q-learning is able to legally run for a seat in the United States Senate). With some improvements, it is one of the most popular RL algorithms in use today. As with Sarsa and MC, we have to estimate the action-value function. Q-learning is defined as follows:

$$q(s_t, a_t) = q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t)]$$

Although it resembles the definition of Sarsa, it has one substantial difference: tt is an off-policy method, which means that we have two distinct policies:

- **Behavior policy**: The agent uses this to actually navigate through the environment. This is the same, as with Sarsa. We start at the s_t state, take the a_t action (following the behavior policy), transition to the next state, s_{t+1}, and then take next action, a_{t+1} (again, following the behavior policy). It's important to note that the behavior policy might not always choose the a_{t+1} action with the maximum expected return value. For example, an ε-greedy policy will sometimes select random non-optimal actions.
- **Target policy**: The agent uses this to compute the TD error in the action-value update function. The target policy is always greedy. That is, the update rule will always use the at+1 action with maximum expected return, regardless of what action the behavior policy might choose (denoted by $\max_a q(s_{t+1}, a)$ symbolizes).

But how do these two policies help us? On the one hand, we'll directly approximate the optimal action-value function, q_*, because we use the greedy target policy for the estimation. Had we used the same greedy policy to navigate the agent, we would have inevitably excluded some action value pairs (as we discussed in the *Exploring starts policy improvement* section). Using a non-optimal behavior policy will ensure that we can include all state/action pairs in the estimation. Conversely, if we had used a non-optimal policy for estimation, we wouldn't have approximated the optimal function.

Knowing all this, let's see how Q-learning works:

1. Input the policy, π.
2. Initialize the $q_\pi(s, a)$ table with some value for all state/action pairs.
3. Repeat the following for a number of episodes:
    1. Start new episode with the initial state/action pair, $s_{t=0}$, $a_{t=0}$.
    2. Repeat the following until the terminal state is reached:
        1. Take the action a_t, transition to a new state, s_{t+1}, and observe the reward, r_{t+1}.
        2. Select next action, a_{t+1}, following the behavior policy (for example, ε-greedy).
        3. Update the action-value function, $q(s_t, a_t) = q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t)]$, using the greedy target policy.
        4. $s_t = s_{t+1}$, $a_t = a_{t+1}$.

If you are interested in an example Python implementation of Q-learning, check out `https://github.com/dennybritz/reinforcement-learning/tree/master/TD`.

Double Q-learning

Imagine that the majority of the actions, *a*, starting from state, *s*, have true action-values, $q_*(s, a) = 0$. That is, the real return for each action starting from the *s* state is 0. Unfortunately, we don't know the true action-values and instead we try to estimate them, hoping that our estimation will eventually converge toward the optimum. Our estimations, $q(s, a)$, are uncertain – some estimations might be slightly above 0, while others might be slightly below. And here comes the issue. When we compute the estimation of each state/action pair using the greedy target policy, we always use the action-value of the pair with the maximum expected return, which is slightly positive. This means the estimated action-values for all pairs will be slightly higher than the real action-values, which are zeros. Therefore, our approximation of the action-value function will deviate from the optimal by constantly overestimating the expected returns. This issue is called **maximization bias**.

The reason for maximization bias in Q-learning is that we use the same greedy target policy to select the action, a_{max}, with the highest expected return, and at the same time to evaluate its action value, $q(s, a_{max})$. If the action, a_{max}, is overestimated, the action-value estimation will select it and it will use its overestimated value as target for all actions. The idea of double Q-learning is to decompose the selection and evaluation in two separate action-value estimations: q_1 and q_2. Both will try to estimate the optimal action-value function, q_*, but we'll split the state/action pairs in two sets – the first set to train q_1 and the second set to train q_2. We'll use q_1 to select the best action and q_2 to estimate its value. Then the update rule for q_3 becomes the following:

$$q_1(s_t, a_t) = q_1(s_t, a_t) + \alpha[r_{t+1} + \gamma q_2(s_{t+1}, \arg\max_a q_1(s_{t+1}, a)) - q_1(s_t, a_t)]$$

q_1 and q_2 will still suffer from maximization bias. But by using different training sets, we can at least ensure that they will overestimate different actions, *a*, when starting from the same state, *s*. In this way, even if q_1 selects an overestimated action, the action-value, q_2, will not be overestimated, thus minimizing the maximization bias (pun intended). We can also reverse the roles of q_1 and q_2 in the preceding formula. To implement this, we'll flip a coin at each step of the Q-learning algorithm (a chance of 0.5) and based on the result, we'll either update q_1 or q_2.

In the following section, we can see a step-by-step trace of double Q-learning:

1. Input the policy, π.
2. Initialize the $q_0(s,a)$ and $q_1(s,a)$ tables with some value for all state/action pairs.
3. Repeat the following for a number of episodes:
    1. Start a new episode with the initial state, $s_{t=0}$.
    2. Repeat until the terminal state is reached:
        1. Select the a_t action following the behavior policy based on both q_1 and q_2 (for example, ε-greedy).
        2. Take the action a_t, transition to new state s_{t+1}, and observe reward r_{t+1}.
        3. Update one of the two action-value estimations with a probability of 0.5:
        $$q_1(s_t,a_t) = q_1(s_t,a_t) + \alpha[r_{t+1} + \gamma q_2(s_{t+1}, \arg\max_a q_1(s_{t+1},a)) - q_1(s_t,a_t)]$$
        $$q_2(s_t,a_t) = q_2(s_t,a_t) + \alpha[r_{t+1} + \gamma q_1(s_{t+1}, \arg\max_a q_2(s_{t+1},a)) - q_2(s_t,a_t)]$$
        4. $s_t = s_{t+1}$.

Value function approximations

So far, we've worked under the assumption that the state- and action- value functions are tabular. However, in tasks with large value spaces, such as computer games, it's impossible to store all possible values in a table. Instead, we'll try to approximate the value functions. To formalize this, let's think of the tabular value functions, v_π and q_π , as actual functions with as many parameters as the number of table cells. As the state space grows, so does the number of parameters, to the point where it becomes impossible to store them. Not only that, but with a large number of states, the agent is bound to enter situations it has never seen before.

Our goal then is to find another set of functions, $\hat{v}$ and $\hat{q}$, with the following properties:

- Approximates v_π and q_π with significantly fewer parameters, compared to the tabular version
- Generalizes well enough, so they can successfully approximate previously-unseen situations

We'll denote these functions with the following:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

Where **w** is the function parameters. We can use any function for approximation, but for the purposes of this book, we'll focus on neural networks. In this case, **w** is the network weights.

So far so good, but how can we train the network? To do this, we'll treat the RL task as a supervised learning problem, where the following is true:

- The network input is the current state or state/action pair (depending on whether it estimates v or q).
- The network output is the value function approximation, $\hat{v}$ or $\hat{q}$.
- The target data (labels) is the real value function. v_π or q_π .

With these assumptions, let's define the loss function for the state- and action- value functions that we'll use for training:

$$J_v(\mathbf{w}) = \frac{1}{2}\mathbb{E}_\pi[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$
$$J_q(\mathbf{w}) = \frac{1}{2}\mathbb{E}_\pi[q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})]^2$$

It is simply the mean-squared error of the sum over all states, *s*, of the difference between the real and approximated value, with respect to the weights, **w**. $\mathbb{E}_\pi$ represents the expectation of a state distribution, which assigns a measure of importance to each state. Think of the state distribution as the amount of time spent in the *s* state relative to the other states.

Next, we can use the now-familiar **stochastic gradient descent** (**SGD**) optimization to update the network parameters. To do this, we'll need the gradient (first derivative) of the loss function with respect to the weights. We can compute it with the help of the chain rule:

$$\begin{aligned}\nabla_\mathbf{w}\ J_v(\mathbf{w})\ &= \nabla_\mathbf{w}\frac{1}{2}\mathbb{E}_\pi[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2 \\ &= -\mathbb{E}[v_\pi(s) - \hat{v}(s, \mathbf{w})]\ \nabla_\mathbf{w}\hat{v}(s, \mathbf{w}) \\ \nabla_\mathbf{w}\ J_q(\mathbf{w}) &= \nabla_\mathbf{w}\frac{1}{2}\mathbb{E}_\pi[q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})]^2 \\ &= -\mathbb{E}[q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})]\ \nabla_\mathbf{w}\hat{q}(s, a, \mathbf{w})\end{aligned}$$

Then, we can compute the weight-update delta by simply multiplying the gradient with the learning rate:

$$\triangle \mathbf{w} = -\alpha \nabla_{\mathbf{W}}\ J_{v,w}(\mathbf{w})$$
$$\mathbf{w} = \mathbf{w} + (-\alpha \nabla_{\mathbf{W}}\ J_{v,w}(\mathbf{w}))$$

But I can hear you say, "How can we do this when we don't know the true value functions? Isn't the whole point of RL to actually find v_π and q_π?" And you're be completely right. To overcome this challenge, we'll use a trick. Recall the state-value function update rule that we introduced in the *Temporal difference methods* section:

$$v(s_t) = v(s_t) + \alpha[\underbrace{r_{t+1} + \gamma v(s_{t+1})}_{\text{target}} - \underbrace{v(s_t)}_{\text{apprx.}}]$$

At the time, we observed how $r_{t+1} + \gamma v(s_{t+1})$ acts as a target value, $v(s_t)$ is the approximation, and $r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ is just the difference between the two. In fact, this is exactly what we'll use as a target in our current task. However, instead of the true value function (which we don't know), we'll use the network as an approximator. The weight update then becomes the following:

$$\mathbf{w} = \mathbf{w} - \alpha(\underbrace{r_{t+1} + \gamma\, \overbrace{\hat{v}(s_{t+1}, \mathbf{w})}^{\text{net output t+1}}}_{\text{target}} - \underbrace{\hat{v}(s_t, \mathbf{w})}_{\text{net output t}}\,)\ \nabla_{\mathbf{w}} \hat{v}(s_t, \mathbf{w})$$

We can train the network online as we let the agent interact with the environment following the steps of the TD algorithm (Sarsa, Q-learning). We'll use the stream of interaction experiences (action, reward, new state) as a training set. As the network training loss converges towards 0, the agent's behavior improves (hopefully).

The following is a step-by-step trace of the TD(0) prediction method with a value-function approximation:

1. Input the following:
    - Policy, π
    - Value function approximator, $\hat{v}$(neural net)

2. Repeat the following steps for a number of episodes:
    1. Start new episode with the initial state, $s_{t=0}$.
    2. Repeat until the terminal state is reached:
        1. Select the a_t action using the policy, π, for the current state, s.
        2. Take the action a_t, transition to new state s_{t+1}, and observe reward r_{t+1}
        3. Update the network weights:
        $$\mathbf{w} = \mathbf{w} - \alpha(r_{t+1} + \gamma\hat{v}(s_{t+1}, \mathbf{w}) - \hat{v}(s_t, \mathbf{w}))\ \nabla_{\mathbf{w}}\hat{v}(s_t, \mathbf{w})$$
        4. $s_t = s_{t+1}$.

Value approximation for Sarsa and Q-learning

We can apply the same to Sarsa, which uses a similar update rule, but we'll approximate the action-value function instead:

$$q(s_t, a_t) = q(s_t, a_t) + \alpha[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)]$$

$$\mathbf{w} = \mathbf{w} - \alpha(\underbrace{r_{t+1} + \gamma\overbrace{\hat{q}(s_{t+1}, a_{t+1}, \mathbf{w})}^{\text{net output t+1}}}_{\text{target}} - \underbrace{\hat{q}(s_t, a_t, \mathbf{w})}_{\text{net output t}})\ \nabla_{\mathbf{W}}\hat{q}(s_t, a_t, \mathbf{w})$$

The same goes for Q-learning:

$$q(s_t, a_t) = q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t)]$$

$$\mathbf{w} = \mathbf{w} - \alpha(\underbrace{r_{t+1} + \gamma \max_a \overbrace{\hat{q}(s_{t+1}, a, \mathbf{w})}^{\text{net outputs t+1}}}_{\text{target}} - \underbrace{\hat{q}(s_t, a_t, \mathbf{w})}_{\text{net output t}})\ \nabla_{\mathbf{W}}\hat{q}(s_t, a_t, \mathbf{w})$$

Improving the performance of Q-learning

In this section, we'll introduce a couple of tricks that can help to improve the agent's performance.

Fixed target Q-network

One issue with the value-function approximation in Q-learning is that we use the same network to compute both the estimation at the *t* time and the TD target value, which is based on the estimation at the *t+1* time (preceding equation). Let's say that we update the network weights at step *t* with the TD target at *t+1*. In the next iteration, we'll calculate the next TD target at step *t+2* (two) using the updated network. As a result, there is a strong correlation between the TD target and the network weights. When the weights change, so does the TD target. Think of it as a moving goalpost – as the network tries to get closer to the TD target, the target shifts and goes further away. This could lead to oscillations and unstable training. One solution to this problem is to use separate network with fixed weights, $\mathbf{w}^{\text{fixed}}$, to compute the target value.

Here's how the process works:

1. Create the fixed target network as a carbon copy of the main network. That is, a copy of the network architecture and weights.
2. Use the target network to generate the TD values for `n` iterations. Throughout the whole time, the $\mathbf{w}^{\text{fixed}}$ weights will be "frozen" – we will not perform any updates on them.
3. After `n` iterations, we'll replace the target network with another carbon copy of the latest version of the main network. Then, we can repeat the whole process.

Using a network with fixed weights will prevent the TD target value from shifting and will stabilize the training. Here is the weight update rule, including the new fixed target network:

$$\mathbf{w} = \mathbf{w} - \alpha(\underbrace{r_{t+1} + \gamma \max_a \overbrace{\hat{q}(s_{t+1}, a, \mathbf{w}^{\mathbf{fixed}})}^{\text{net outputs t+1}}}_{\text{target}} - \underbrace{\hat{q}(s_t, a_t, \mathbf{w})}_{\text{net output t}})\, \nabla_{\mathbf{W}} \hat{q}(s_t, a_t, \mathbf{w})$$

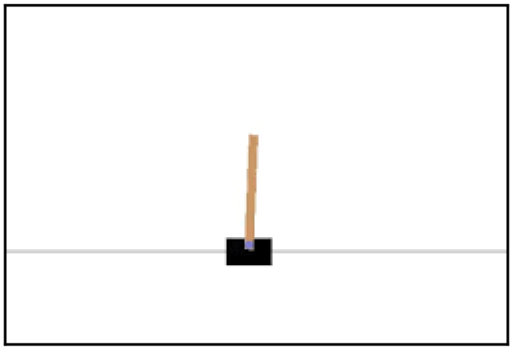
Experience replay

As we discussed in the *Value function approximation* section, we are training the network online, as the agent receives stream of experiences from the environment. But the environment is usually sequential, and consecutive experiences might not differ much. For example, imagine that the agent is a car, which is currently sliding downhill. While doing so, it receives consistent feedback that the speed increases. If we feed the network with such unified training data, there is a chance that it will start dominating all other experiences. The network might "forget" previous situations and the performance would decrease (this is a disadvantage of some neural networks). We can solve this issue with experience replay. As the environment interaction goes, we'll store a sliding window of the latest `n` interactions: `(state` s_{t-1}, `action` a_{t-1}, `reward` r_t, `state` s_t`) for t =` t_{now-n} `...` t_{now} . Instead of training the network with the latest data, we'll create one mini-batch by extracting samples from various points of the sliding window. In this way, the network will receive diversified training data and will perform much better. We can also improve experience replay by prioritizing the experiences (**prioritized experience replay**). For example, if a transition yielded a high TD error, we could use this training sample more often until it improves.

This concludes our (rather lengthy) theoretical introduction to RL. We now have enough knowledge to solve some fun RL tasks. In the next section, we'll see how to use Q-learning to play a very simple computer game.

Q-learning in action

In this section, we'll use Q-learning in combination with a simple neural network to control an agent in the cart-pole task. We'll use an ε-greedy policy and experience replay. This is a classic RL problem. The agent must balance a pole attached to the cart via a joint. At every step, the agent can move the cart left or right. It receives a reward of 1 every time step that the pole is balanced. If the pole deviates by more than 15 degrees from upright, the game ends:



The cart-pole task

To help us with this, we'll use OpenAI Gym (`https://gym.openai.com/`), which is an open source toolkit for the development and comparison of RL algorithms. It allows us to teach agents over various tasks, such as walking or playing games such as Pong, Pinball, other Atari games, and even Doom.

We can install it with `pip`:

```
pip install gym[all]
```

Next, let's start with the code.

1. First, we'll do the imports:

```
import random
from collections import deque

import gym
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

2. Then, we'll create the cart-pole environment:

```
env = gym.make('CartPole-v0')
```

The `gym.make` method creates the environment that our agent will run in. Passing in the `"CartPole-v0"` string tells the OpenAI Gym that we want this to be the cart-pole environment, represented by the `env` object. We'll use it to interact with the game. The `env.reset()` method puts the environment into its initial state, returning an array that describes it. Subsequent calls to `env.step(action)` allow us to interact with the environment, returning the new states in response to the agent's actions. Calling `env.render()` will display the current state on the screen. The environment state is an array of four floating-point values, which describe the position and angle of the cart and the pole.

3. We'll use the environment state array as input to our network. It will consist of one hidden layer with 20 nodes, a `tanh` activation function, and an output layer with two nodes. One output node will learn the expected reward for a move to the left in the current state, the other the expected reward for a move to the right.

Here is how that code looks:

```
# Build the network
input_size = env.observation_space.shape[0]

input_placeholder = tf.placeholder("float", [None, input_size])

# weights and bias of the hidden layer
weights_1 = tf.Variable(tf.truncated_normal([input_size, 20],
stddev=0.01))
bias_1 = tf.Variable(tf.constant(0.0, shape=[20]))

# weights and bias of the output layer
weights_2 = tf.Variable(tf.truncated_normal([20,
env.action_space.n], stddev=0.01))
bias_2 = tf.Variable(tf.constant(0.0, shape=[env.action_space.n]))

hidden_layer = tf.nn.tanh(tf.matmul(input_placeholder, weights_1) +
bias_1)
output_layer = tf.matmul(hidden_layer, weights_2) + bias_2

action_placeholder = tf.placeholder("float", [None, 2])
target_placeholder = tf.placeholder("float", [None])
```

Why 1 hidden layer with 20 nodes? Why use a `tanh` activation function? Picking hyperparameters is a dark art; the best answer we can give is that these values worked well for the task at hand. When selecting network architecture, we are usually interested in computation time and preventing overfitting. In RL, neither of these issues is as important. Though we care about computation time, often the bottleneck is the time spent running the game. As for overfitting, in RL we don't have train/validation/test set split. Instead, we have an environment in which an agent gets a reward. So, overfitting is not something we have to worry about (until we start to train agents that can operate across multiple environments). This is why you won't often see RL agents use regularizers. The caveat to this is that over the course of training, the distribution of our training set may change significantly as our agent improves its policy. There is always the risk that the agent may overfit on the early training samples, which can cause the learning to become more difficult later.

Would a deeper network be better? Maybe, but for tasks with such a minimal amount of complexity, more layers tend not to improve things. Running the network with extra hidden layers appears to make little difference. One hidden layer gives us the capacity we need to learn the things we want in this task.

Why did we choose tanh activation, when a sigmoid would have worked too (we have only one hidden layer)? We know that our target can be negative (for a negative expected reward). That would suggest that the range of (-1 : 1) provided by the `tanh` function might be preferable to the logistic range of (0 : 1). To judge negative rewards, the sigmoid will have to work in combination with the bias. This is a lot of conjecture and reasoning after the fact; the best answer is ultimately that this combination works very well on this task.

4. Next, let's define our `loss` function and optimizer (Adam):

```
# network estimation
q_estimation = tf.reduce_sum(tf.multiply(output_layer,
action_placeholder), reduction_indices=1)

# loss function
loss = tf.reduce_mean(tf.square(target_placeholder - q_estimation))

# Use Adam
train_operation = tf.train.AdamOptimizer().minimize(loss)

# initialize TF variables
session = tf.Session()
session.run(tf.global_variables_initializer())
```

The `q_estimation` variable will be the q-value network prediction. Multiplying `output_layer` by the `action_placeholder` tensor will return 0 for everything except for the action we took. Our `loss` is the difference between the network estimation and `target_placeholder`.

5. Then, let's define our simplified ε-greedy policy:

```
def choose_next_action(state, rand_action_prob):
    """
    Simplified e-greedy policy
    :param state: current state
    :param rand_action_prob: probability to select random action
    """

    new_action = np.zeros([env.action_space.n])

    if random.random() <= rand_action_prob:
        # choose an action randomly
        action_index = random.randrange(env.action_space.n)
    else:
        # choose an action given our state
        action_values = session.run(output_layer,
feed_dict={input_placeholder: [state]})[0]
        # we will take the highest value action
        action_index = np.argmax(action_values)

    new_action[action_index] = 1
    return new_action
```

6. Next, we'll define the `train` function, which works for a single `mini_batch`:

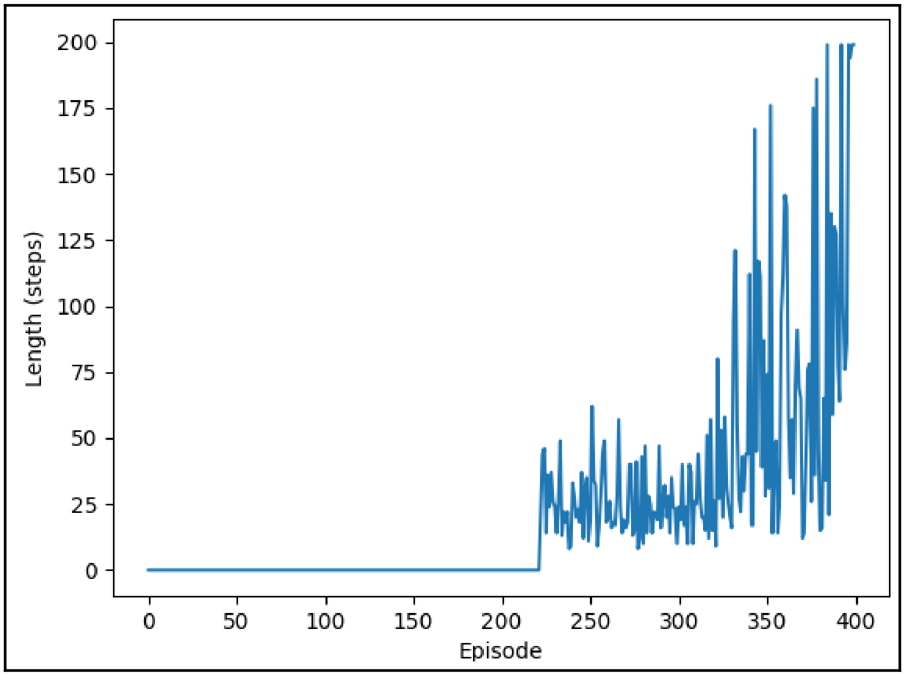
```
def train(mini_batch):
    """
    Train the network on a single minibatch
    :param mini_batch: the mini-batch
    """

    last_state, last_action, reward, current_state, terminal =
range(5)

    # get the batch variables
    previous_states = [d[last_state] for d in mini_batch]
    actions = [d[last_action] for d in mini_batch]
    rewards = [d[reward] for d in mini_batch]
    current_states = [d[current_state] for d in mini_batch]
    agents_expected_reward = []
    # this gives us the agents expected reward for each action we
might take
    agents_reward_per_action = session.run(output_layer,
feed_dict={input_placeholder: current_states})
    for i in range(len(mini_batch)):
        if mini_batch[i][terminal]:
            # this was a terminal frame so there is no future
reward...
            agents_expected_reward.append(rewards[i])
        else:
            # otherwise compute expected reward
            discount_factor = 0.9
            agents_expected_reward.append(
                rewards[i] + discount_factor *
np.max(agents_reward_per_action[i]))

    # learn that these actions in these states lead to this reward
    session.run(train_operation, feed_dict={
        input_placeholder: previous_states,
        action_placeholder: actions,
        target_placeholder: agents_expected_reward})
```

7. Then, let's define the `q_learning` function, which will put the whole thing together:

```
def q_learning():
    """The Q-learning method"""

    episode_lengths = list()

    # Experience replay buffer and definition
    observations = deque(maxlen=200000)

    # Set the first action to nothing
    last_action = np.zeros(env.action_space.n)
    last_action[1] = 1
    last_state = env.reset()

    total_reward = 0
    episode = 1

    time_step = 0

    # Initial chance to select random action
    rand_action_prob = 1.0

    while episode <= 500:
        # render the cart pole on the screen
        # comment this for faster execution
        # env.render()

        # select action following the policy
        last_action = choose_next_action(last_state,
rand_action_prob)

        # take action and receive new state and reward
        current_state, reward, terminal, info =
env.step(np.argmax(last_action))
        total_reward += reward

        if terminal:
            reward = -1.
            episode_lengths.append(time_step)

            print("Episode: %s; Steps before fail: %s; Epsilon:
%.2f reward %s" %
                  (episode, time_step, rand_action_prob,
total_reward))
            total_reward = 0

        # store the transition in previous_observations
        observations.append((last_state, last_action, reward,
current_state, terminal))

        # only train if done observing
        min_experience_replay_size = 5000
        if len(observations) > min_experience_replay_size:
            # mini-batch of 128 from the experience replay
observations
            mini_batch = random.sample(observations, 128)

            # train the network
            train(mini_batch)

            time_step += 1

        # reset the environment
        if terminal:
            last_state = env.reset()
            time_step = 0
            episode += 1
        else:
            last_state = current_state

        # gradually reduce the probability of a random action
        # starting from 1 and going to 0
```

```
            if rand_action_prob > 0 and len(observations) >
    min_experience_replay_size:
                rand_action_prob -= 1.0 / 15000

        # display episodes length
        plt.xlabel("Episode")
        plt.ylabel("Length (steps)")
        plt.plot(episode_lengths, label='Episode length')
        plt.show()
```

8. Finally, we can run the task by calling `q_learning()`. If everything goes as planned, the code will produce a chart that displays the length of each episode:



Number of steps per episode

This looks good. For the first 200 or so episodes, we wanted to fill the experience replay buffer with enough samples and no training was done. Then, we quickly reached 200 steps per episode around game 400, at which point the environment imposes a limit on the maximum episode length.

Summary

In this chapter, we introduced RL. We started with some basic paradigms and then we discussed how to represent RL as a Markov Decision Process. We talked about the core RL approaches – DP, Monte Carlo, and TD. Then, we learned about Sarsa, Q-learning, and value function approximation using neural networks. Finally, we used the OpenAI Gym to teach a simple agent to play the classic cart-pole game.

In the next chapter, we'll try to solve more advanced RL problems, such as Go and Atari games, with the help of some state-of-the-art RL algorithms, such as Monte Carlo Tree Search and Deep Q-learning.

9
Deep Reinforcement Learning for Games

In the `chapter 8`, *Reinforcement Learning Theory*, we introduced **Reinforcement Learning** (**RL**), a way to make a computer interact with an environment. In this chapter, we'll build upon that knowledge and we'll explore some more advanced RL algorithms and tasks. But don't worry, we won't create the Terminator just yet. We're aiming a little lower, so we'll just see how to teach a machine to play games such as Atari Breakout and Go.

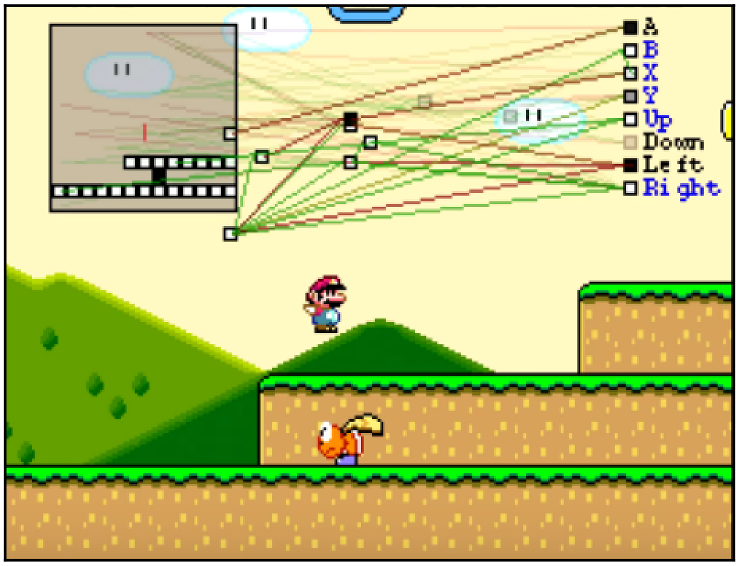
This chapter will cover the following:

- Introduction to genetic algorithms playing games
- Deep Q-learning (DQN)
- Policy gradients
- Actor-critic methods
- Monte Carlo tree search
- AlphaZero

Introduction to genetic algorithms playing games

For a long time, the best results and the bulk of the research into AIs playing video game environments were around genetic algorithms. This approach involves creating a set of modules that take parameters to control the behavior of the AI. The range of parameter values is then set by a selection of genes. A group of agents would then be created using different combinations of these genes, which would be run on the game.

The most successful set of agent's genes would be selected, then a new generation of agents would be created using combinations of the successful agent's genes. Those would again be run on the game and so on until a stopping criteria is reached, normally either a maximum number of iterations or a level of performance in the game. Occasionally, when creating a new generation, some of the genes can be mutated to create new genes. A good example of this is *MarI/O*, an AI that learned to play the classic SNES game *Super Mario World* using neural network genetic evolution:



Super Mario World with neural network genetic evolution

The big downside of these approaches is that they require a lot of time and computational power to simulate all the variations of parameters. Each member of every generation must run through the whole game until the terminal state. The technique also does not take advantage of any of the rich information in the game that a human can use. Whenever a reward or punishment is received, there is contextual information around the state and the actions taken, but genetic algorithms only use the final result of a run to determine fitness. They are not so much learning as doing trial and error. In this chapter, we'll present some better approaches using (can you guess?) deep reinforcement learning, a combination of deep networks and RL.

Deep Q-learning

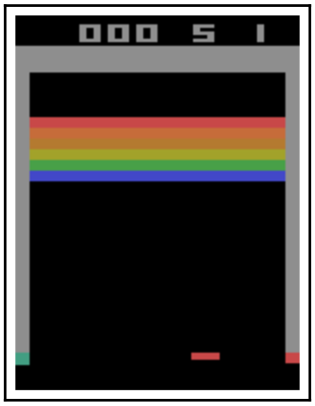
We ended `Chapter 8`, *Reinforcement Learning Theory,* with an example of an agent learning to play the cart-pole game with the help of Q-learning and a simple network with one hidden layer. The state of the cart-pole environment is described with four numerical variables: cart position and velocity, and pole angle and velocity. We used these variables as an input to the q-function approximation network and successfully trained the agent to prevent the pole from tipping over for more than 200 episode steps. But if it was a human playing the game, he or she would steer the cart based on the screen images he or she sees. That is, if we think of the human as an "agent," the environment "state" he or she would use would be the sequence of frames displayed on the screen. Compare this to just four variables our artificial agent used, and you'll see that its task was much easier than that of the human. And yet, a person wouldn't have any problem understanding what's on the screen. This is not limited to just cart pole; we can learn any game, following the same "procedure." Can we teach the agent do the same and only learn from the screen images, without any prior knowledge of the game rules? As we ask this question, you have probably already guessed there is a way to do it. In 2013, Minh et al. (all from DeepMind; `https://deepmind.com/`) released the seminal paper *Playing Atari with Deep Reinforcement Learning* (`https://arxiv.org/abs/1312.5602`). They demonstrated how to use Q-learning with a **convolutional neural network** (**CNN**) acting as a value-function approximator to play a range of Atari games. The solution described in the paper is very similar to the example we introduced at the end of the last chapter, but with two major distinctions:

- The paper uses experience replay and a CNN as q-function approximator.
- The network input is a sequence of the n latest game frames. As we know from `Chapter 4`, *Computer Vision with Convolutional networks*, the CNN input can be a grayscale or RGB image. Here, an RGB game frame is converted to grayscale and then a sequence of the latest frames is used as inputs to the network.

Playing Atari Breakout with Deep Q-learning

In this section, we'll implement an agent playing Atari Breakout (`https://en.wikipedia.org/wiki/Breakout_(video_game)`). In this game, the player can use a ball to knock down the eight rows of bricks, located at the top of the screen. The game is won when all bricks are knocked down and lost if the ball reaches the bottom of the screen. The ball can ricochet off the screen walls.

The player can prevent the ball from falling by navigating a pad (located at the bottom) left or right. Each knocked down brick carries a reward of 1:



Atari Breakout

Due to the nature of RL, this example might take a long time to train (usually multiple hours, and sometimes more than a day)

We'll solve this task using deep Q-learning with the following tricks and improvements:

- ε-greedy policy (`Chapter 8`, *Reinforcement Learning Theory*).
- Experience replay (`Chapter 8`, *Reinforcement Learning Theory*).
- Fixed q-target network (`Chapter 8`, *Reinforcement Learning Theory*).
- Our deep network will use four sequential game frames as an input, because we need multiple frames to know the ball's direction.

The code in this section is based on the *Playing Atari with Deep Reinforcement Learning* (`https://arxiv.org/abs/1312.5602`) paper and partially inspired by `https://github.com/dennybritz/reinforcement-learning/`. We'll also use some of the improvements introduced in the paper *Rainbow: Combining Improvements in Deep Reinforcement Learning* (`https://arxiv.org/abs/1710.02298`). With that short introduction, let's start!:

1. First, we'll do the imports (as usual):

```
import os
import pickle
import random
import zlib
from collections import deque
from collections import namedtuple

import gym
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

2. Next, we'll define some parameters of the RL algorithm. The constants are annotated with comments, which describe their purpose:

```
resume = True  # resume training from checpoint (if exists)
CHECKPOINT_PATH = 'deep_q_breakout_path_7'
MB_SIZE = 32  # mini batch size
ER_BUFFER_SIZE = 1000000  # experience relay (ER) buffer size
COMPRESS_ER = True  # compress episodes in the EP buffer
EXPLORE_STEPS = 1000000  # frames over which to anneal epsilon
EPSILON_START = 1.0  # starting chance of an action being random
EPSILON_END = 0.1  # final chance of an action being random
STATE_FRAMES = 4  # number of frames to store in the state
SAVE_EVERY_X_STEPS = 10000  # how often to save the model on the
disk
UPDATE_Q_NET_FREQ = 1  # how often to update the q network
UPDATE_TARGET_NET_EVERY_X_STEPS = 10000  # copy the q-net weights
to the target net
DISCOUNT_FACTOR = 0.99  # discount factor
```

3. Next, we'll define the `initialize` function, which does the following:
    - Initializes the **TensorFlow** (**TF**) session.
    - Creates the estimation and target networks `q_network` and `t_network`.
    - Defines the TF operations, which copy the weights from `q_network` to `t_network`. There is one such operation for each network parameter, defined in `t_net_updates`.
    - Initializes the Adam optimizer (as suggested by `https://arxiv.org/abs/1710.02298`).
    - Initializes the `frame_proc` routine, which transforms the RGB frames to network inputs (we'll talk about it later).
    - Restores the TF session (that is, the network and optimizer) from a previously saved checkpoint, to resume the training.

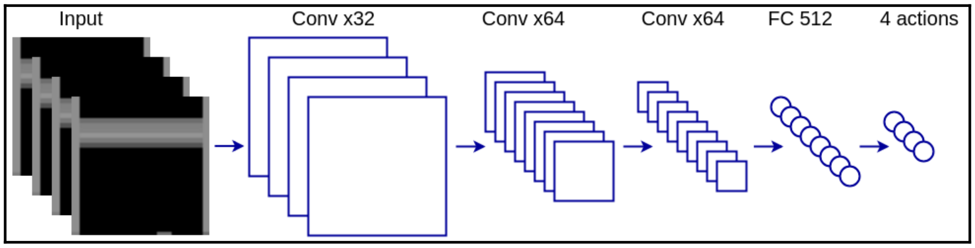
The following is the implementation:

```
def initialize():
    """Initialize the session, the networks, and the environment"""
    # Create environment
    env = gym.envs.make("BreakoutDeterministic-v4")

    tf.reset_default_graph()

    session = tf.Session()

    # Tracks the total number of training steps
    tf.Variable(0, name='global_step', trainable=False)

    # Create q- and target- networks
    q_network = build_network("q_network")
    t_network = build_network("target_network")

    # create the operations to copy the q-net weights to the t-net
    q_net_weights = [t for t in tf.trainable_variables()
                     if t.name.startswith(q_network.scope)]
    q_net_weights = sorted(q_net_weights, key=lambda v: v.name)
    t_net_weights = [t for t in tf.trainable_variables()
                     if t.name.startswith(t_network.scope)]
    t_net_weights = sorted(t_net_weights, key=lambda v: v.name)

    t_net_updates = \
        [n2_v.assign(n1_v) for n1_v, n2_v in zip(q_net_weights,
t_net_weights)]

    # pre-processor of game frames
    frame_proc = frame_preprocessor()

    optimizer = tf.train.AdamOptimizer(0.00025)
    # optimizer = tf.train.RMSPropOptimizer(0.00025, 0.99, 0.0,
1e-6)

    # training op
    train_op = optimizer.minimize(q_network.loss,
global_step=tf.train.get_global_step())

    # restore checkpoint
    saver = tf.train.Saver()

    if not os.path.exists(CHECKPOINT_PATH):
        os.mkdir(CHECKPOINT_PATH)

    checkpoint = tf.train.get_checkpoint_state(CHECKPOINT_PATH)
    if resume and checkpoint:
        session.run(tf.global_variables_initializer())
        session.run(tf.local_variables_initializer())

        print("\nRestoring checkpoint...")
        saver.restore(session, checkpoint.model_checkpoint_path)
    else:
        session.run(tf.global_variables_initializer())
        session.run(tf.local_variables_initializer())

    return session, \
           q_network, \
           t_network, \
           t_net_updates, \
           frame_proc, \
           saver, \
           train_op, \
           env
```

Note that the function returns multiple variables. Later, when we make the function call, we'll turn them into global variables. When the following code references some of them, know that they were defined here.

4. Next, we'll define the `build_network` function. We'll use it to build both the estimation and target networks. The result of the function is a `namedtuple`, which contains the inputs (placeholders) and outputs (tensors) of the network. The network itself has the following properties:
    - Three convolutional layers and two fully-connected layers with ReLU activations (as suggested by `https://arxiv.org/abs/1710.02298`).
    - It solves a regression (the difference between the target and output estimations). Therefore, take the output of the last hidden layer without any modifications (such as softmax).
    - We'll use Huber loss (`https://en.wikipedia.org/wiki/Huber_loss`), which is somewhat similar to the mean-squared-error. It allows us to perform something akin to error clipping: put the reward in the [-1,1] range.

- The output is *q* estimations for all possible environment actions, given the input state. Here, we have four actions:

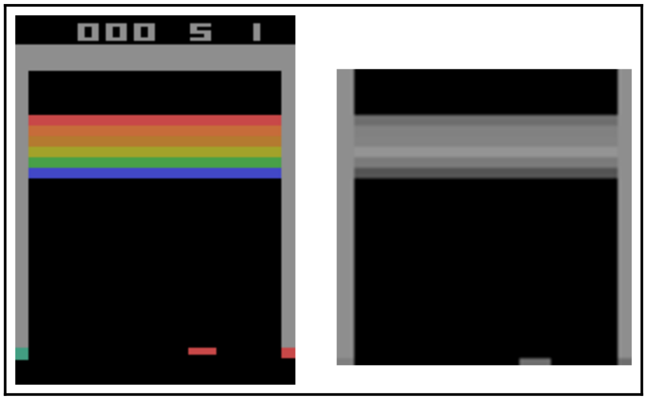


The network takes as input multiple pre-processed game frames and outputs value estimations of all actions

The following is the implementation:

```
def build_network(scope: str, input_size=84, num_actions=4):
    """Builds the network graph."""

    with tf.variable_scope(scope):
        # Our input are STATE_FRAMES grayscale frames of shape 84,
84 each
        input_placeholder = tf.placeholder(dtype=np.float32,
                                           shape=[None, input_size,
input_size, STATE_FRAMES])

        normalized_input = tf.to_float(input_placeholder) / 255.0

        # action prediction
        action_placeholder = tf.placeholder(dtype=tf.int32,
shape=[None])

        # target action
        target_placeholder = tf.placeholder(dtype=np.float32,
shape=[None])

        # Convolutional layers
        conv_1 = tf.layers.conv2d(normalized_input, 32, 8, 4,
                                  activation=tf.nn.relu)
        conv_2 = tf.layers.conv2d(conv_1, 64, 4, 2,
                                  activation=tf.nn.relu)
        conv_3 = tf.layers.conv2d(conv_2, 64, 3, 1,
                                  activation=tf.nn.relu)

        # Fully connected layers
        flattened = tf.layers.flatten(conv_3)
        fc_1 = tf.layers.dense(flattened, 512,
                                activation=tf.nn.relu)

        q_estimation = tf.layers.dense(fc_1, num_actions)

        # Get the predictions for the chosen actions only
        batch_size = tf.shape(normalized_input)[0]
        gather_indices = tf.range(batch_size) *
tf.shape(q_estimation)[1] + action_placeholder
        action_predictions = tf.gather(tf.reshape(q_estimation,
[-1]), gather_indices)

        # Calculate the loss
        loss = tf.losses.huber_loss(labels=target_placeholder,
                                    predictions=action_predictions,
reduction=tf.losses.Reduction.MEAN)

    Network = namedtuple('Network',
                         'scope '
                         'input_placeholder '
                         'action_placeholder '
                         'target_placeholder '
                         'q_estimation '
                         'action_predictions '
                         'loss ')

    return Network(scope=scope,
                   input_placeholder=input_placeholder,
                   action_placeholder=action_placeholder,
                   target_placeholder=target_placeholder,
                   q_estimation=q_estimation,
                   action_predictions=action_predictions,
                   loss=loss)
```

5. Then, we'll define the `frame_preprocessor` function. Note that it uses TF graph of operations to transform the RGB game frame to an input tensor of the network. It does so by cropping, resizing, and converting it to grayscale. We'll use the output of this operation chain as an input to our network:



An example of game frame (left) and the same frame, pre-processed as input to the network

The following is the implementation:

```
def frame_preprocessor():
    """Pre-processing the input data"""

    with tf.variable_scope("frame_processor"):
        input_placeholder = tf.placeholder(shape=[210, 160, 3],
dtype=tf.uint8)
        processed_frame =
tf.image.rgb_to_grayscale(input_placeholder)
        processed_frame =
tf.image.crop_to_bounding_box(processed_frame, 34, 0, 160, 160)
        processed_frame = tf.image.resize_images(
            processed_frame,
            [84, 84],
            method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

        processed_frame = tf.squeeze(processed_frame)

    FramePreprocessor = namedtuple('FramePreprocessor',
'input_placeholder processed_frame')

    return FramePreprocessor(
        input_placeholder=input_placeholder,
        processed_frame=processed_frame)
```

6. Next, we'll define the `choose_next_action` function, which implements the ε-greedy policy. It starts by taking the q estimations of the `net` network, given the current `state`. Then, it modifies the probability of the most likely action with the `epsilon` value. Finally, it makes a semi-random choice of the new action, by taking into account the modified probabilities. We should note that the value of `epsilon` linearly decreases, as the agent gathers gathers more experience (this is implemented outside the function). Following is the implementation:

```
def choose_next_action(state, net, epsilon):
    """Epsilon-greedy policy"""

    # choose an action given our last state
    tmp = np.ones(env.action_space.n, dtype=float) * epsilon /
env.action_space.n
    q_estimations = session.run(net.q_estimation,
        feed_dict={net.input_placeholder: np.reshape(state, (1,) +
state.shape)})[0]

    tmp[np.argmax(q_estimations)] += (1.0 - epsilon)

    new_action = np.random.choice(np.arange(len(tmp)), p=tmp)

    return new_action
```

7. Then, we'll implement the `populate_experience_replay_buffer` function. It will generate the initial experience replay buffer, before the actual training has started. The function runs multiple game episodes. During an episode, the agent follows the ε-greedy policy we just defined in `choose_next_action`. The episode steps come in the shape of game frames, which are combined in groups of four (the `STATE_FRAMES` parameter), and then stored in the buffer variable (which is of type `deque`). When an episode is over, we reset the environment and start a new episode, and we repeat this until the buffer is filled. We can choose to compress the states, before saving them in the buffer (the `COMPRESS_ER` constant). This option is selected by default, as it reduces the memory consumption and doesn't have significant impact on performance. Following is the implementation:

```
def populate_experience_replay_buffer(buffer: deque,
initial_buffer_size: int):
 """Initial population of the experience replay buffer"""

 # Initialize epsilon based on the current step
 epsilon_step = (EPSILON_START - EPSILON_END) / EXPLORE_STEPS
 epsilon = max(EPSILON_END,
 EPSILON_START -
 session.run(tf.train.get_global_step()) * epsilon_step)

 # Populate the replay memory with initial experience
 state = env.reset()
 state = session.run(frame_proc.processed_frame,
 feed_dict={frame_proc.input_placeholder: state})

 state = np.stack([state] * STATE_FRAMES, axis=2)

 for i in range(initial_buffer_size):

 # Sample next state with the q_network
 action = choose_next_action(state, q_network, epsilon)

 # Perform one action step
 next_state, reward, terminal, info = env.step(action)
 next_state = session.run(frame_proc.processed_frame,
 feed_dict={frame_proc.input_placeholder: next_state})

 # Stack the game frames in a single array
 next_state = np.append(state[:, :, 1:], np.expand_dims(next_state,
2), axis=2)

 # Store the experience in ER
 if COMPRESS_ER:
 buffer.append(
 zlib.compress(
 pickle.dumps((state, action, reward, next_state, terminal), 2),
2))
 else:
 buffer.append((state, action, reward, next_state, terminal))

 # Set next state as current
 if terminal:
 state = env.reset()
 state = session.run(frame_proc.processed_frame,
 feed_dict={frame_proc.input_placeholder: state})

 state = np.stack([state] * STATE_FRAMES, axis=2)
 else:
 state = next_state

 print("\rExperience replay buffer: {} / {} initial ({}
total)".format(
 len(buffer), initial_buffer_size, buffer.maxlen), end="")
```

8. Next, we'll implement the `deep_q_learning` function, which is the centerpiece of our program. As the name suggests, it runs the Q-learning algorithm. Although the function is long, we have done our best to provide enough comments to make it understandable. Nevertheless, let's discuss some of the more important moments. After some initializations, we start the main loop. One iteration of the loop represents one step in an episode. For each step, we do the following:
    - Compute the new, reduced value of `epsilon`, which decreases linearly with each iteration. The parameters of the decrease are defined with the configuration constants.
    - If necessary, synchronize the q and target networks, by copying the weights from the q network to the target network.
    - Select a new action, following the ε-greedy policy, according to the new `epsilon` and the current `state`.
    - Send the action to the environment `env` and receive `next_state` and `reward`.
    - Store the `(state, action, reward, next_state, terminal)` tuple in the `observations` experience replay buffer. Note that we also store whether the state is terminal (game over) or not.
    - Sample one `mini_batch` of experiences from the experience replay buffer.
    - Then, sample the estimations of the next actions, `q_values_next` using the target network,`t_network`.
    - Compute the estimated discounted returns `targets_batch` of the next actions by taking into account whether the state is `terminal` or not.
    - Perform one gradient descent step. DeepMind suggests to do one gradient update every four steps of the environment (that is, four frames). We can do this with the `UPDATE_Q_NET_FREQ` constant, but in this example we've chosen to update on every frame: `UPDATE_Q_NET_FREQ=1`.
    - If the state is `terminal` (game over), we save the progress, generate chart, reset the environment, and start again (within the same loop).

Finally, here is the function itself:

```
def deep_q_learning():
    """The Q-learning training process"""

    # build experience replay
    observations = deque(maxlen=ER_BUFFER_SIZE)

    print("Populating replay memory...")
    populate_experience_replay_buffer(observations, 100000)

    # initialize statistics
    stats = namedtuple('Stats', 'rewards lengths')(rewards=list(),
lengths=list())
    global_time = session.run(tf.train.get_global_step())
    time = 0

    episode = 1

    episode_reward = 0
    global_reward = 0

    # Start the training with an initial state
    state = env.reset()
    state = session.run(frame_proc.processed_frame,
                        feed_dict={frame_proc.input_placeholder:
state})
    state = np.stack([state] * STATE_FRAMES, axis=2)

    while True:
        # env.render()

        # Initialize epsilon based on the current step
        epsilon_step = (EPSILON_START - EPSILON_END) /
EXPLORE_STEPS
        epsilon = max(EPSILON_END, EPSILON_START - (global_time -
1) * epsilon_step)

        # Copy q-net weights to the target-net
        if global_time % UPDATE_TARGET_NET_EVERY_X_STEPS == 0:
            session.run(t_net_updates)
            print("\nCopied model parameters to target network.")

        # Sample next action
        action = choose_next_action(state, q_network, epsilon)

        # Perform one step with the selected action
        next_state, reward, terminal, info = env.step(action)
        # This is how we pre-process
        next_state = session.run(frame_proc.processed_frame,
feed_dict={frame_proc.input_placeholder: next_state})

        # Stack the game frames in a single array
        next_state = np.append(state[:, :, 1:],
np.expand_dims(next_state, 2), axis=2)

        # Store the experience in ER
        if COMPRESS_ER:
            observations.append(
                zlib.compress(pickle.dumps((state, action, reward,
next_state, terminal), 2), 2))
        else:
            observations.append((state, action, reward, next_state,
terminal))

        # Sample a mini-batch from the experience replay memory
        mini_batch = random.sample(observations, MB_SIZE)
        if COMPRESS_ER:
            mini_batch = [pickle.loads(zlib.decompress(comp_item))
for comp_item in mini_batch]

        states_batch, action_batch, reward_batch,
next_states_batch, terminal_batch = \
            map(np.array, zip(*mini_batch))

        if global_time % UPDATE_Q_NET_FREQ == 0:
            # Compute next q values using the target network
            q_values_next = session.run(t_network.q_estimation,
                feed_dict={t_network.input_placeholder:
next_states_batch})

 # Calculate q values and targets
 targets_batch = reward_batch + \
 np.invert(terminal_batch).astype(np.float32) * \
 DISCOUNT_FACTOR * \
 np.amax(q_values_next, axis=1)

 # Perform gradient descent update
 states_batch = np.array(states_batch)

            _, loss = session.run([train_op, q_network.loss],
                                  feed_dict={
                                      q_network.input_placeholder:
states_batch,
                                      q_network.action_placeholder:
action_batch,
                                         q_network.target_placeholder:
targets_batch})

        episode_reward += reward
        global_reward += reward
        time += 1
        global_time += 1

        print("\rEpisode {}: "
              "time {:5}; "
              "reward {}; "
              "epsilon: {:.4f}; "
              "loss: {:.6f}; "
              "@ global step {} "
              "with total reward {}".format(
            episode,
            time,
            episode_reward,
            epsilon,
            loss,
            global_time,
            global_reward), end="")

        if terminal:
            # Episode end

            print()

            stats.rewards.append(int(episode_reward))
            stats.lengths.append(time)

            time = 0
            episode_reward = 0
            episode += 1

            state = env.reset()
            state = session.run(frame_proc.processed_frame,
feed_dict={frame_proc.input_placeholder: state})
            state = np.stack([state] * STATE_FRAMES, axis=2)
        else:
            # Set next state as current
            state = next_state

        # save checkpoints for later
        if global_time % SAVE_EVERY_X_STEPS == 0:
            saver.save(session, CHECKPOINT_PATH + '/network',
                       global_step=tf.train.get_global_step())
```

```
                # plot the results and save the figure
                plot_stats(stats)

                fig_file = CHECKPOINT_PATH + '/stats.png'
                if os.path.isfile(fig_file):
                    os.remove(fig_file)

                plt.savefig(fig_file)
                plt.close()

                # save the stats
                with open(CHECKPOINT_PATH + '/stats.arr', 'wb') as f:
                    pickle.dump((stats.rewards, stats.lengths), f)
```

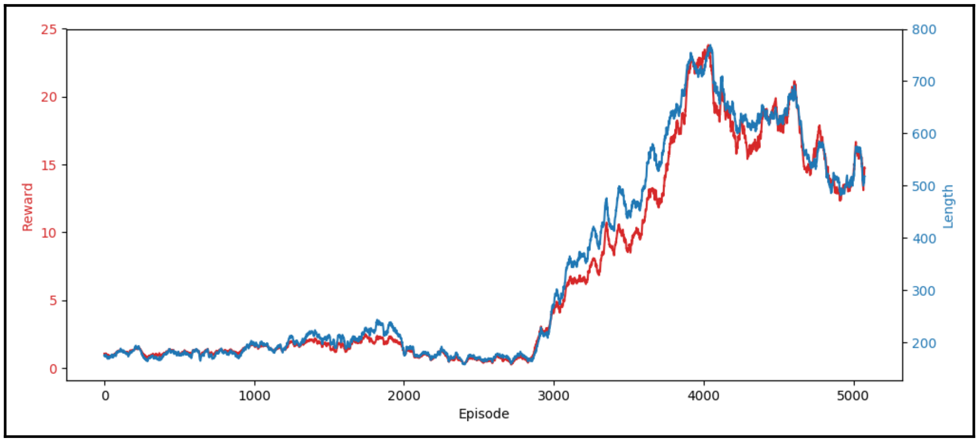
9. As a dessert, let's implement the `plot_stats` function, which simply plots the moving average of episode lengths and rewards:

```
def plot_stats(stats):
    """Plot the stats"""
    plt.figure()

    plt.xlabel("Episode")

    # plot the rewards
    # rolling mean of 50
    cumsum = np.cumsum(np.insert(stats.rewards, 0, 0))
    rewards = (cumsum[50:] - cumsum[:-50]) / float(50)

    fig, ax1 = plt.subplots()

    color = 'tab:red'

    ax1.set_ylabel('Reward', color=color)
    ax1.plot(rewards, color=color)
    ax1.tick_params(axis='y', labelcolor=color)

    # plot the episode lengths
    # rolling mean of 50
    cumsum = np.cumsum(np.insert(stats.lengths, 0, 0))
    lengths = (cumsum[50:] - cumsum[:-50]) / float(50)

    ax2 = ax1.twinx()

    color = 'tab:blue'
    ax2.set_ylabel('Length', color=color)
    ax2.plot(lengths, color=color)
    ax2.tick_params(axis='y', labelcolor=color)
```

10. Finally, we can run the whole thing:

```
if __name__ == '__main__':
    session, q_network, t_network, t_net_updates, frame_proc,
saver, train_op, env = \
        initialize()
    deep_q_learning()
```

If everything goes alright, in a few hours we'll see how the average length and reward of the episodes starts to increase during training. In the following chart, we can see how the episode length and reward change with training:



The reward and episode length increase as the training episodes increase

At one point in the training, the reward per episode goes as high as 25. If we omit the averaging, we'll see individual episodes with a reward larger than 40. That is, the ball has been able to knock out more than 40 bricks before the game ends. Although this is not an earth shattering result, it clearly shows that the agent has learned to interact with the environment in a non-random way.

We can improve upon our result with double Q-learning, which we introduced in `Chapter 8`, *Reinforcement Learning Theory*. Since we already use **deep Q-learning** (**DQN**), the new abbreviation will become DQN. Recall that in DQN we have two approximation networks. We use one of them to compute the next action *q* values and the other to actually select the best action from these values.

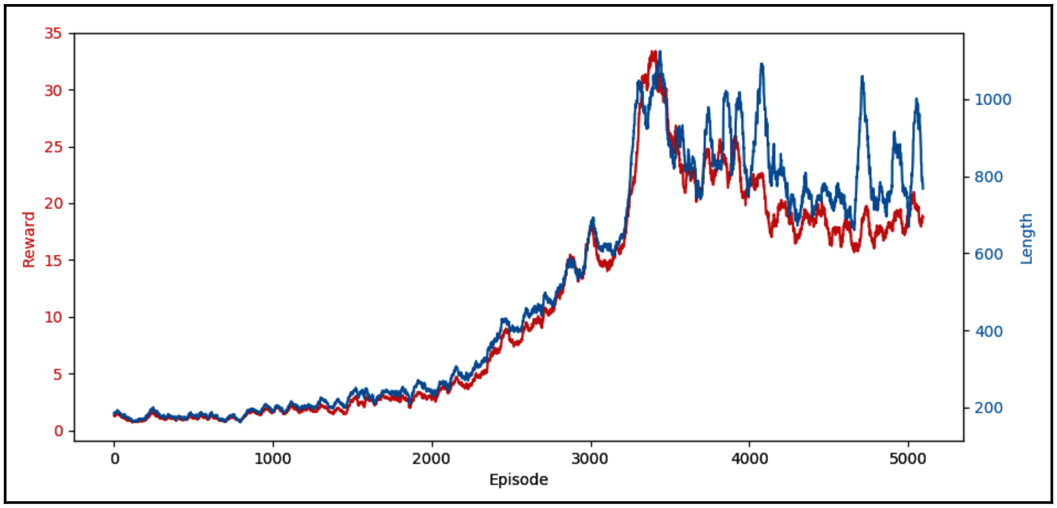
In our example, we already have two networks, `q_network` and `t_network`, and we can put them to additional use in the double Q-learning scenario. That is, we'll use `t_network` to compute the next action q values as before. However, we'll select the best action with `q_network`. In practice, we'll can do this by removing the following code from the `deep_q_learning` function:

```
              # Calculate q values and targets
              targets_batch = reward_batch + \
                              np.invert(terminal_batch).astype(np.float32) *
   \
                              DISCOUNT_FACTOR * \
                              np.amax(q_values_next, axis=1)

              # Perform gradient descent update
              states_batch = np.array(states_batch)
```

And we then replace it with this:

```
            # The best action according to the q-network
            best_actions = np.argmax(q_values_next, axis=1)

            # Next, predict the next q values with the target-network
            q_values_next_target = session.run(t_network.q_estimation,
            feed_dict={t_network.input_placeholder: next_states_batch})

            # Calculate q values and targets
            # Use the t-network estimations
            # But with the best action, selected by the q-network (Double Q-
            learning)
            targets_batch = reward_batch + \
                            np.invert(terminal_batch).astype(np.float32) * \
                            DISCOUNT_FACTOR * \
                            q_values_next_target[np.arange(MB_SIZE),
            best_actions]
```

Additionally, we'll set a new value to the `EPSILON_END = 0.01` constant. With these changes, the code will produce the following result:



The moving average result of DQN training

As we can see, the results are better compared to regular DQN. In fact, in one episode the agent managed to receive an award of 61 and another episode lasted for 2778 steps (not visible because of the moving average). We can see that in both cases, the agent peaks at one point and then the result gradually declines.

Unfortunately, these examples go to show that training a value-function approximator in an RL scenario is not easy. From the charts, we can see that the results start to diverge from random late in the training. To even see whether our agent learns anything at all, we need to wait multiple hours.

Policy gradient methods

All RL algorithms we discussed until now have tried to learn the state- or action-value functions. For example, in Q-learning we usually follow an ε-greedy policy, which has no parameters (OK, it has one parameter) and relies on the value function instead. In this section, we'll discuss something new: how to approximate the policy itself with the help of policy gradient methods. We'll follow a similar approach as in `Chapter 8`, *Reinforcement Learning Theory*, in the *Value function approximation* section.

There, we introduced a value approximation function, which is described by a set of parameters **w** (neural net weights). Here, we'll introduce a parameterized policy π_θ , which is described by a set of parameters **θ**. As with value function approximation, **θ** could be the weights of a neural network.

Recall that we use the $\pi(a|s)$ notation to describe the probability, which a stochastic (and not deterministic) policy, π ,assigns to an action, *a*, given current state *s*. We'll denote the parameterized policy with $\pi(a|s, \boldsymbol{\theta})$. That is, the policy will recommend new action based on the environment state *s*, but also on its internal "state," described by **θ**.

Let's say that we have some scalar valued function $J(\boldsymbol{\theta})$, which measures the performance of a parameterized policy π_θ with respect to its parameters **θ**. Our goal is to maximize it. A policy gradient method uses gradient ascent to update the parameters **θ** in a way that maximizes $J(\boldsymbol{\theta})$. That is, we can compute the first derivative (or gradient) of $J(\boldsymbol{\theta})$ with respect to **θ** and use it to update **θ** in a way, which will increase $J(\boldsymbol{\theta})$. We'll denote this with the following:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla \widehat{_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}}_t)$$

Where α is the learning rate, $\nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t)$ is the derivative of $J(\boldsymbol{\theta})$ with respect to **θ**, and $\nabla \widehat{_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}}_t)$ symbolizes the increasing gradient. This process is the opposite of the gradient descent we use to minimize the loss function of a neural network during training.

Approximating the policy has a few advantages over value-function approximation:

- With training, it can approach deterministic policy, whereas the ε-greedy policy in a value approximation approach always includes a random decision making component ε (even when ε is small).
- Sometimes, we may be able to approximate the policy with a simpler function, compared to the value approximation function.
- If we have prior domain knowledge of the environment, we can embed it in the policy parameters.
- we get better convergence, because the action probabilities change smoothly. In value approximation methods, a small change in the estimated action values can lead to a dramatic change in the action selection, if that change results in a different action with maximum estimation. For example, imagine a simple maze-walking robot. Let's say that at the first T-junction it encounters, it will move left. Successive iterations of Q-learning will eventually show that the right is preferable. But because the path is completely different, every other state/action q value has to be recalculated and the previous knowledge is of little value.

One disadvantage of policy-based methods is that they can converge toward a local maximum of $J(\boldsymbol{\theta})$ (and not the global one).

We know that the result of $J(\boldsymbol{\theta})$ is a scalar value, which measures the policy performance. But what exactly does performance mean? Recall that the goal of the agent is to maximize the cumulative total reward. We can intuitively see that we can measure the policy performance using the same metric. That is, the higher total reward the agent gets from following policy π_θ , the better the policy is. Then, we can define the policy performance for single episode as follows:

$$J(\boldsymbol{\theta}) = v_{\pi_\theta}(s_0)$$

Where s^0 is the initial state of the episode and $v_{\pi_\theta}(s_0)$ is the state-value function when we follow our parameterized policy π_θ . In other words, finding the gradient ∇v_{π_θ} will be the same as finding $\nabla_\theta J(\boldsymbol{\theta})$. We can find ∇v_{π_θ} with the help of policy gradient theorem, which establishes the following:

$$\nabla_\theta J(\boldsymbol{\theta}) = v_{\pi_\theta}(s_0) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta})$$

The formula has the following components:

- $\mu(s)$ is the state probability distribution. Think of it as a weight assigned to each state. Usually, the state distribution is chosen to be the time spent in that state, compared to the other states. The sum of all distributions is $\sum_s \mu(s) = 1$.
- $q_\pi(s,a)$ is the action-value function, when following the parameterized policy π_θ.
- $\nabla_\theta \pi(a|s,\boldsymbol{\theta})$ is the derivative of the parameterized policy function with respect to **θ**.
- $\propto$ means "proportional to."

We will not provide a formal proof of the theorem, but we can intuitively say that the gradient$\nabla_\theta J(\boldsymbol{\theta})$ depends on state and action distributions (that is, the environment), as well as the parameterized policy (and by extension its parameters **θ**).

Monte Carlo policy gradients with REINFORCE

REINFORCE is a Monte Carlo policy gradient method. It is Monte Carlo in the sense that it updates the policy by playing full environment episodes, in the same way as the Monte Carlo value-approximation methods we described in `Chapter 8`, *Reinforcement Learning Theory*. Once an episode finishes, REINFORCE updates the policy parameters **θ** for each step t of the episode trajectory with the following rule:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}$$

Where α is the learning rate and G_t is the total discounted reward at time *t*. But, let's discuss the last element of the equation. We divide $\nabla\pi(a|s_t, \boldsymbol{\theta})$ (the gradient of the probability of taking action a_t, given state s_t and $\boldsymbol{\theta}_t$) by the probability itself. If the gradient $\nabla\pi(a|s_t, \boldsymbol{\theta})$ is positive, we want to update **θ** in a way that will make selecting the same action more likely. Conversely, if $\nabla\pi(a|s_t, \boldsymbol{\theta})$ is negative, we want to make the selection of the same action less likely. In other words, we want to update **θ** proportional to the gradient, hence it is in the numerator. But why divide by the probability? The intuition here is that if the action has high probability, it will receive updates more often, which might skew the probabilities unfairly in its direction. We want to discourage this behavior, hence the probability is in the denominator.

The expression $\frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})}$ has a compact representation $\nabla_{\boldsymbol{\theta}} \ln \pi(a_t|s_t, \boldsymbol{\theta})$ and the two are equal (we won't provide formal proof though).

The following is the step-by-step trace of the REINFORCE algorithm:

1. The algorithm takes as input the parameterized policy $\pi(a|s, \boldsymbol{\theta})$.
2. Initialize the parameters **θ** in some arbitrary way (for example, with random values).

3. Repeat this for a number of episodes:
    1. Generate a new episode, following the policy $\pi(a|s, \boldsymbol{\theta})$: `s0, a0, r1, s1, a1, r2, s2, a2, r3, ... aT-1, rT, sT`
    2. Iterate over each step `t` of the episode, starting from `0` and going to `T-1`:
        1. Calculate the total discounted return *G* at step t $G_t = \sum_{j=t}^{T} \gamma^{j-t} r_{j+1}$, where r_j is the reward at episode step *j* and γ is the discount factor.
        2. Update the parameters $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}$.

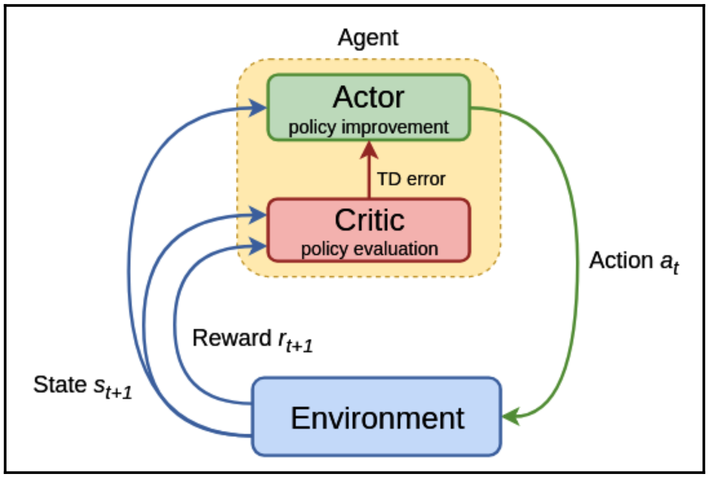
Policy gradients with actor–critic

Actor-critic (**AC**) is a family of policy gradient algorithms similar to the **temporal difference** (**TD**) methods (`Chapter 8`, *Reinforcement Learning Theory*). That is, unlike Monte Carlo, an AC method doesn't have to play whole episodes to update the policy parameters **θ**. AC has two components:

- The actor, which is the parameterized policy $\pi(a|s, \boldsymbol{\theta})$. The actor (agent) will use the policy to make decisions on what action to take next.
- The critic, which is the state- or action value function approximation $\hat{v}$ or $\hat{q}$ (we introduced this notation in `Chapter 8`, *Reinforcement Learning Theory*). The critic will use the TD error as a feedback to the actor's actions.

AC is a hybrid between policy- and value-based methods, since it tries to learn both the policy and the value function.

The following is a diagram of AC:



Actor-Critic method diagram

The execution of AC follows a similar pattern to REINFORCE, but instead of playing full episodes, we'll update the policy parameters **θ** after each episode step. Because of this, we won't have access to the total discounted return, G_t. Fear not, as we'll replace it with the value function approximation $\hat{v}$ (as in TD(0)) or $\hat{q}$ (as in SARSA and Q-learning). You can also think of AC in the opposite way, as a TD algorithm where we use parameterized instead of an ε-greedy policy. This will introduce an extra step in our algorithm; we'll have to learn an additional set of parameters **w** for either $\hat{q}(s, a, \mathbf{w})$ or $\hat{v}(s, \mathbf{w})$.

First, let's see how AC works with state-value function approximation $\hat{v}$. We'll start with the weight update rules for **w** and **θ**:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta \underbrace{\hat{v}(s_t, \mathbf{w})}_{\text{net output t}} \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}$$

$$\mathbf{w} = \mathbf{w} - \alpha_{\mathbf{w}} \Big(\underbrace{r_{t+1} + \gamma \overbrace{\hat{v}(s_{t+1}, \mathbf{w})}^{\text{net output t+1}}}_{\text{target}} - \underbrace{\hat{v}(s_t, \mathbf{w})}_{\text{net output t}} \Big) \nabla_{\mathbf{w}} \hat{v}(s_t, \mathbf{w})$$

Where $\alpha_{\mathbf{w}}$ and α_θ are the learning rates. We want to update **w** in a way, which will minimize the TD error. On the other hand, the goal of the **θ** update is to maximize the return.

The following is a step-by-step trace of the AC algorithm with $\hat{v}$:

1. Input the following:
    1. Value-function estimator $\hat{v}(s, \mathbf{w})$ (neural net)
    2. Parameterized policy $\pi(a|s, \boldsymbol{\theta})$
2. Repeat for a number of episodes:
    1. Start a new episode with initial state $s_{t=0}$.
    2. Repeat until the terminal state is reached:
        1. Select action a_t, following the policy $\pi(a_t|s_t, \boldsymbol{\theta})$ for the current state s_t.
        2. Take action a_t, transition to new state s_{t+1}, and observe reward r_{t+1}
        3. Update the parameters:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta \hat{v}(s_t, \mathbf{w}) \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}$$

$$\mathbf{w} = \mathbf{w} - \alpha_{\mathbf{w}} (r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}) - \hat{v}(s_t, \mathbf{w}))\ \nabla_{\mathbf{w}} \hat{v}(s_t, \mathbf{w})$$

        4. Set the current state to s_{t+1}: $s_t = s_{t+1}$

Next, let's talk about AC with action-value approximation $\hat{q}$. The weight update rules become as follows:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta \underbrace{\hat{q}(s_t, a_t, \mathbf{w})}_{\text{net output t}} \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}$$

$$\mathbf{w} = \mathbf{w} - \alpha_{\mathbf{w}} (\underbrace{r_{t+1} + \gamma \overbrace{\hat{q}(s_{t+1}, a_{t+1}, \mathbf{w})}^{\text{net output t+1}}}_{\text{target}} - \underbrace{\hat{q}(s_t, a_t, \mathbf{w})}_{\text{net output t}})\ \nabla_{\mathbf{W}} \hat{q}(s_t, a_t, \mathbf{w})$$

The following is a step-by-step trace of the AC algorithm with $\hat{q}$:

1. Input the following:
    1. Value function estimator $\hat{q}(s, a, \mathbf{w})$ (neural net)
    2. Parameterized policy $\pi(a|s, \boldsymbol{\theta})$
2. Repeat for a number of episodes:
    1. Start new episode with initial state/action pair $s_{t=0}$, $a_{t=0}$.
    2. Repeat until the terminal state is reached:
        1. Take the action, a_t, transition to new state, s_{t+1}, and observe reward r_{t+1}
        2. Select next action a_{t+1}, following the policy $\pi(a_{t+1}|s_{t+1}, \boldsymbol{\theta})$.
        3. Update the parameters:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta \hat{q}(s_t, a_t, \mathbf{w}) \frac{\nabla_{\boldsymbol{\theta}} \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})}$$

$$\mathbf{w} = \mathbf{w} - \alpha_{\mathbf{w}}(r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}) - \hat{q}(s_t, a_t, \mathbf{w}))\, \nabla_{\mathbf{W}} \hat{q}(s_t, a_t, \mathbf{w})$$

        4. $s_t = s_{t+1}$, $a_t = a_{t+1}$

Actor-Critic with advantage

One of the drawbacks of AC (and policy-based methods in general) is the high variance of $\nabla_\theta J(\boldsymbol{\theta})$. To understand this, note that we update the policy parameters **θ** by observing the rewards of multiple episodes. Let's focus on a single episode. The agent starts at initial state *s* and then takes a series of actions following the policy π_θ . These actions lead to new states and their corresponding rewards. When the terminal state is reached, the episode has accumulated some total reward. We'll use these rewards to update the policy parameters **θ** either online (AC) or once at the end of the episode (REINFORCE). Next, let's imagine that once the episode is finished, the agent will start another episode with the same initial state, *s*. It would make sense that the new episode will have the same trajectory as the previous one. However, this might not be the case.

At some episode step, a stochastic policy might dictate the agent to take a different action compared to before. Not only that, but a stochastic environment might present a different state, even if the agent takes the same action as before. This effect is especially pronounced, because it can happen at any one of the many episode steps. Once that happens, the remaining trajectory of the episode could be totally different than before, which could lead to totally different rewards. Therefore, even a minor change in the policy or the environment can lead to completely different outcomes, which is what we call high variance. We can intuitively say that this unpredictability is not good for the learning process.

Since we took so much time to explain this problem, you might have guessed that we won't leave it open; we'll introduce a solution. The solution is to subtract a (preferably) constant baseline value from the rewards of each episode. Let's try to explain this with an example. Imagine that we have the same situation with two episodes starting from the same initial state, but with different trajectories. Let's also imagine that $\nabla_\theta J(\boldsymbol{\theta})_1 = 0.7$ for the first episode and $\nabla_\theta J(\boldsymbol{\theta})_2 = 0.3$ for the second. Next, let's say that the total reward of the first episode is 200 and for the second is 190. In this case, the update rule for the first episode (say in REINFORCE) will include $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta * 0.7 * 200... = \boldsymbol{\theta} + \alpha_\theta * 140...$. On the other hand, the update rule for the second one will include $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta * 0.3 * 190... = \boldsymbol{\theta} + \alpha_\theta * 57...$. As we can see, the weight updates will differ by a lot. However, we can mitigate this problem by subtracting a constant value from both rewards. For example, if this constant is 180, we'll have `0.7 * (200 - 180) = 14` and `0.3 * (190 - 180) = 3` respectively. Although the results are still different, they are much closer than before.

We can implement this in practice with the so-called **advantage function**, where we use the state-value function$v$as baseline. The following is the advantage function when used with the action values:

$$A(s, a) = q(s, a) - v(s)$$

However, we can decompose $q(s, a)$ as a sum of two components:

- The immediate reward r_{t+1}, we get when we take action a_t and transition from s_t->s_{t+1}
- The discounted state-value function of the new state s_{t+1}

Therefore, we can transform the advantage formula to this:

$$A(s_t, a_t) = q(s_t, a_t) - v(s_t) = \underbrace{\overbrace{r_{t+1} + \gamma v(s_{t+1})}^{\text{TD target}} - v(s)}_{\text{TD error}}$$

This is just the TD error for the state-value function. The AC method with advantage is abbreviated as A2C.

One of the successful applications of A2C is by the OpenAI Five algorithm for playing Dota 2. This is a multiplayer online battle game, where 2 teams of 5 players (heroes) play against each other. The goal of each team is to destroy the opposing team's "Ancient" - a large structure located in the team's base. The game is highly complex: the episodes can last for 45 minutes on average, the heroes can only partially observe their surrounding environment, (represented by a large map), and each hero can take dozens of actions. In OpenAI Five, the heroes of one of the team are controlled by a combination of five LSTM networks. The networks are trained with a A2C-based algorithm called Proximal Policy Optimization (PPO, `https://arxiv.org/abs/1707.06347`). The performance of the algorithm was tested with a game (best-of-three games) against a team of five of the best Dota 2 human players. Although OpenAI Five ultimate lost 2 of the games, it was still an impressive achievement.

Playing cart pole with A2C

In this section, we'll implement an agent that tries to play the cart pole game with the help of A2C. We'll do this with the familiar tools: the OpenAI Gym and TensorFlow. Recall that the state of the cart-pole environment is described by the position and angle of the cart and the pole. We'll use feedforward networks with one hidden layer for both the actor and the critic. Let's start!:

1. First, we'll do the imports:

```
from collections import namedtuple

import gym
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

2. Next, we'll create the environment:

```
env = gym.make('CartPole-v0')
```

3. Then, we'll add some of the hyperparameters, which describe the training. We'll take the `INPUT_SIZE` and `ACTIONS_COUNT` from the environment. Additionally, we'll create one training mini-batch for the actor from three episodes:

```
DISCOUNT_FACTOR = 0.9
LEARN_RATE_ACTOR = 0.01
LEARN_RATE_CRITIC = 0.01
TRAIN_ACTOR_EVERY_X_EPISODES = 3
INPUT_SIZE = env.observation_space.shape[0]
ACTIONS_COUNT = env.action_space.n
```

4. Next, we'll create the TF session:

```
session = tf.Session()
```

5. Then, we'll define the `build_actor` function, which will create the parameterized policy (actor) network. It has one hidden layer with 20 neurons, Tanh activation, and a two-neuron softmax output. The output represents the probability to take each of the two possible actions (left or right):

```
def build_actor():
    """Actor network definition"""

    input_placeholder = tf.placeholder("float", [None, INPUT_SIZE])

    # hidden layer definition
    hidden_weights = tf.Variable(tf.truncated_normal([INPUT_SIZE,
20], stddev=0.01))
    hidden_bias = tf.Variable(tf.constant(0.0, shape=[20]))
    hidden_layer = tf.nn.tanh(
        tf.matmul(input_placeholder, hidden_weights) + hidden_bias)

    # output layer definition
    output_weights = tf.Variable(tf.truncated_normal([20,
ACTIONS_COUNT], stddev=0.01))
    output_bias = tf.Variable(tf.constant(0.1,
shape=[ACTIONS_COUNT]))
    output_layer = tf.nn.softmax(
        tf.matmul(hidden_layer, output_weights) + output_bias)

    action_placeholder = tf.placeholder("float", [None,
ACTIONS_COUNT])
    advantage_placeholder = tf.placeholder("float", [None, 1])
    # training
    policy_gradient = tf.reduce_mean(advantage_placeholder
                                     * action_placeholder
                                     * tf.log(output_layer))

    train_op = tf.train.AdamOptimizer(LEARN_RATE_ACTOR).minimize(-
policy_gradient)

    return Actor(train_op=train_op,
                 input_placeholder=input_placeholder,
                 action_placeholder=action_placeholder,
                 advantage_placeholder=advantage_placeholder,
                 output=output_layer)
```

Note that the result of the function is a named tuple, `Actor`, which is defined as follows:

```
Actor = namedtuple("Actor",
                   ["train_op",
                    "input_placeholder",
                    "action_placeholder",
                    "advantage_placeholder",
                    "output"])
```

6. Next, let's define the critic network. It has one hidden layer with 20 neurons and a single-neuron regression output for the state value:

```
def build_critic():
    """Critic network definition"""

    input_placeholder = tf.placeholder("float", [None, INPUT_SIZE])

    # hidden layer
    hidden_weights = tf.Variable(tf.truncated_normal([INPUT_SIZE,
20], stddev=0.01))
    hidden_bias = tf.Variable(tf.constant(0.0, shape=[20]))
    hidden_layer = tf.nn.tanh(
        tf.matmul(input_placeholder, hidden_weights) + hidden_bias)

    # output layer
    output_weights = tf.Variable(tf.truncated_normal([20, 1],
stddev=0.01))
    output_bias = tf.Variable(tf.constant(0.0, shape=[1]))
    output_layer = tf.matmul(hidden_layer, output_weights) +
output_bias

    target_placeholder = tf.placeholder("float", [None, 1])
    # cost and training
    cost = tf.reduce_mean(tf.square(target_placeholder -
output_layer))
    train_op =
tf.train.AdamOptimizer(LEARN_RATE_CRITIC).minimize(cost)

    return Critic(train_op=train_op,
                  cost=cost,
                  input_placeholder=input_placeholder,
                  target_placeholder=target_placeholder,
                  output=output_layer)
```

As with the actor, the result is a named tuple, `Critic`, which is defined as follows:

```
Critic = namedtuple("Critic",
                    ["train_op", "cost", "input_placeholder",
"target_placeholder", "output"])
```

7. Then, let's define the `choose_action` method. It generates next action probabilities with the actor network, and then makes a random decision based on them:

```
def choose_next_action(actor: Actor, state):
    """Actor selects next action"""

    probability_of_actions = session.run(actor.output,
feed_dict={actor.input_placeholder: [state]})[0]
    try:
        move = np.random.multinomial(1, probability_of_actions)
    except ValueError:
        # Sometimes because of rounding errors we end up with
        # action probabilities sum greater than 1.
        # In this case we need to reduce it slightly to be valid
        move = np.random.multinomial(1,
            probability_of_actions / (sum(probability_of_actions) +
1e-6))

    return move
```

8. Next, we'll implement the `a2c` function, which is the centerpiece of our program. It does the following:
    1. Builds the `actor` and `critic` networks and initialize the environment `env`.
    2. Starts the training by playing episodes and training the `actor` and `critic` networks. At each training step, we'll do the following:
        1. We'll collect the trajectory of an episode in the lists `episode_states`, `episode_rewards`, and `episode_actions` lists. Once the episode is finished, we'll use them to generate one training mini-batch for the `critic` network and we'll perform one training step with said mini-batch. Note that although we wait for the episode to finish to do one training step, this is only for convenience and doesn't change the nature of the A2C algorithm. That is, unlike REINFORCE, we still compute the `state_values` and `advantages` at each episode step, as if we didn't know the full trajectory.
        2. We'll also collect the combined trajectories of several episodes (`TRAIN_ACTOR_EVERY_X_EPISODES`) in the `batch_states`, `batch_advantages`, and `batch_actions` lists. We'll use them to create a single training mini-batch for the actor network.
        3. We'll stop the training once we have 10 consecutive episodes with maximum lengths. Let's hope that our A2c is smart enough to do this, otherwise we'll end up in an infinite loop.
    3. Finally, we'll display a chart with the episode lengths, averaged over 10 episodes.

The following is the implementation:

```
def a2c():
    """A2C implementation"""

    actor = build_actor()
    critic = build_critic()

    session.run(tf.initialize_all_variables())

    time = 0

    last_state = env.reset()

    # Trajectory of the current episode
    episode_states, episode_rewards, episode_actions = [], [], []

    # A combination of multiple episode trajectories for one mini-
batch
    batch_states, batch_advantages, batch_actions = [], [], []

    episode_lengths = list()

    while True:
        # env.render()

        # The actor (policy) selects the next action
        last_action = choose_next_action(actor, last_state)
        current_state, reward, terminal, info =
env.step(np.argmax(last_action))

        if terminal:
            reward = -.10
        else:
            reward = 0.1

        episode_states.append(last_state)
        episode_rewards.append(reward)
        episode_actions.append(last_action)

        # We wait for the terminal state
        # Then create one training mini-batch of all episode steps
        # We do this for convenience, but this is still online
method
        if terminal:
            episode_lengths.append(time)
            print("Episode: {} reward
{}".format(len(episode_lengths), time))
```

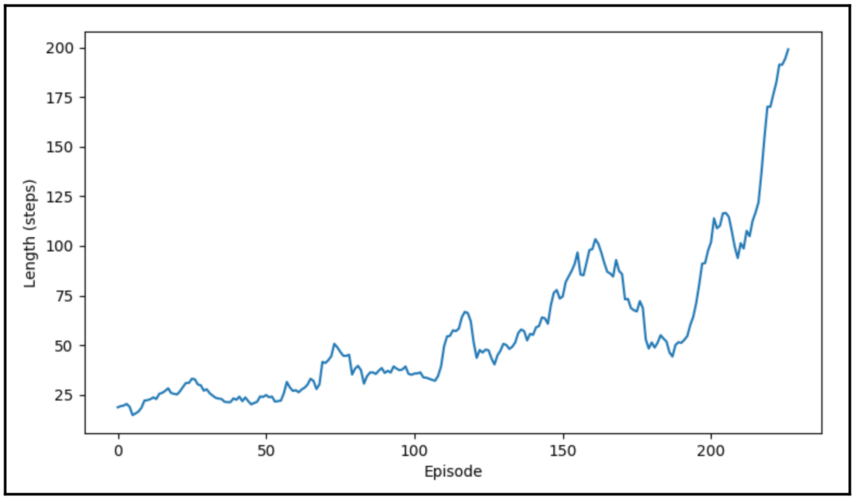
```
            # Stop when the last 10 episodes have maximum length
            if len(episode_lengths) > 10 \
                    and sum(episode_lengths[-10:]) / 10 ==
env._max_episode_steps - 1:
                break

            # get temporal difference values for critic for each
step
            cumulative_reward = 0
            for i in reversed(range(len(episode_states))):
                cumulative_reward = episode_rewards[i] + \
                                    DISCOUNT_FACTOR *
cumulative_reward
                episode_rewards[i] = [cumulative_reward]

            # estimate the state value for each state of the
episode
            state_values = session.run(critic.output,
                feed_dict={critic.input_placeholder:
episode_states})

            # calculate the advantage function for each state of
the episode
            advantages = list()

            for i in range(len(episode_states) - 1):
                advantages.append([episode_rewards[i][0] +
                                   DISCOUNT_FACTOR * state_values[i
+ 1][0]
                                   - state_values[i][0]])

            advantages.append([episode_rewards[-1][0] -
state_values[-1][0]])

            # train the critic (policy) over all steps of the
episode
            session.run([critic.train_op], {
                critic.input_placeholder: episode_states,
                critic.target_placeholder: episode_rewards})

            # add the current episode to the mini-batch
            batch_states.extend(episode_states)
            batch_actions.extend(episode_actions)
            batch_advantages.extend(advantages)

            # train the actor (state-value estimation)
            if len(episode_lengths) % TRAIN_ACTOR_EVERY_X_EPISODES
== 0:
```

```
                # standardize the data using z-standardization
                batch_advantages = np.array(batch_advantages)
                normalized_rewards = batch_advantages -
np.mean(batch_advantages)
                normalized_rewards /= np.std(normalized_rewards)

                # train the actor (policy)
                session.run(actor.train_op, feed_dict={
                    actor.input_placeholder: batch_states,
                    actor.action_placeholder: batch_actions,
                    actor.advantage_placeholder:
normalized_rewards})

                # reset batch trajectories
                batch_states, batch_actions, batch_advantages = [],
[], []

            time = 0

            # reset episode trajectory
            episode_states, episode_rewards, episode_actions = [],
[], []

            # start new episode
            last_state = env.reset()
        else:
            # if not terminal state, then continue
            last_state = current_state
            time += 1

    # display episodes length with moving average 10
    cumsum = np.cumsum(np.insert(episode_lengths, 0, 0))
    episode_lengths = (cumsum[10:] - cumsum[:-10]) / float(10)

    plt.xlabel("Episode")
    plt.ylabel("Length (steps)")
    plt.plot(episode_lengths, label='Episode length')
    plt.show()
```

9. Finally, we can run the whole thing with this:

```
a2c()
```

If everything goes as planned, the program will produce the following chart in a short amount of training time:



Training results of A2C on the cart-pole task

This task is fairly easy and we achieved the maximum episode length in around 200 episodes.

Model-based methods

RL methods such as Monte Carlo, SARSA, Q-learning, or Actor-Critic are model-free. The main goal of the agent is to learn an (imperfect) estimation of either the true value function (MC, SARSA, Q-learning) or the optimal policy (AC). As the learning goes on, the agent needs to have a way to explore the environment in order to collect experiences for its training. Usually, this happens with trial and error. For example, an ε-greedy policy will take random actions at certain times, just for the sake of environment exploration.

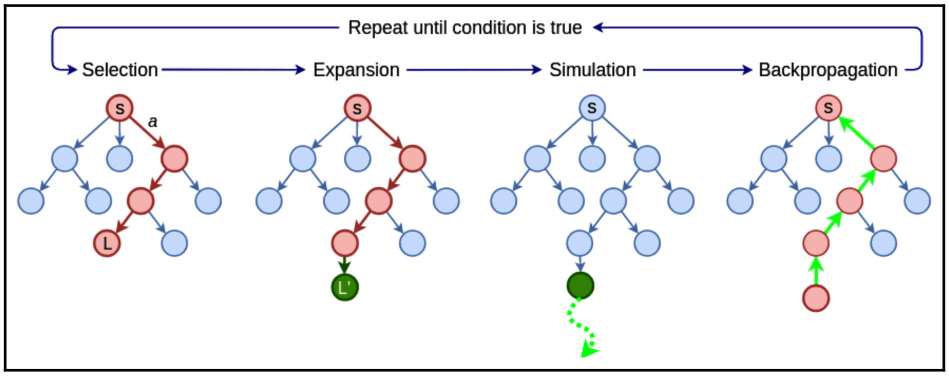
In this section, we'll introduce model-based RL methods, where the agent won't follow the trial-and-error approach when it takes new actions. Instead, it will plan the new action with the help of a model of the environment. The model will try to simulate how the environment will react to a given action. Then, the agent will make its decision based on the simulation result.

Next, we'll learn about one of the most successful model-based methods, called Monte Carlo Tree Search.

Monte Carlo Tree Search

In **Monte Carlo Tree Search** (**MCTS**), the environment model is represented by a search tree. Let's say that the agent is at some state, *s*. Our immediate goal is to select the next action (and our main goal is to maximize the total reward). To do this, we'll create a new search tree with a single node (root): the state *s*. Then, we'll gradually build it node by node by playing simulated episodes. The edges of the tree will represent actions and the nodes will represent states where the agent ends up. In the process of tree building (that is, playing simulations), we'll assign some performance value over each action (edge). Once we finish building it, we'll be able to select the action (starting from the root node, *s*) with the best performance value. In this section, we'll work in a tabular (fully known) environment.

To better understand this process, let's assume that we have already built part of the tree, and we are looking to expand it. The following is a diagram of the expansion of the tree with a new node (action/state):



One MCTS sequence

The process has four steps:

1. **Selection**: We'll start from the root node *s*, and we'll recursively select child nodes until we reach a leaf, *L*. How do we choose which child node to select at each step of the recursion? We'll do this with a special greedy **tree policy**, which will make the selection based on the performance value associated with each action. We'll also maintain how many times we have selected each action throughout the tree building process.

2. **Expansion**: If the leaf *L* is not terminal, we'll add one or more children to *L* by selecting some new action(s) and transitioning to the resulting state(s), *L'*. The actions can be selected randomly.
3. **Simulation**: Starting from *L'*, the agent will continue taking actions until it reaches a terminal state. However, during the simulation it will not consult the search tree, since it has not been built for this trajectory yet. Instead, it will follow a special **rollout policy**. This step is very similar to the Monte Carlo method we introduced in `chapter 8`, *Reinforcement Learning Theory*.
4. **Backpropagation**: The simulation episode from step 3 has generated some total reward. In this step, we'll propagate the reward back to the tree and we'll update the performance values of the actions until we reach the root node. We will only update the path we generate during the selection step.

We'll repeat these four steps until some condition is true. For example, we can stop after a certain timeout. Once the tree is ready, we'll select the next action, starting from the root node, *s*, and transitioning to the next state *s'*. We can repeat the same process for *s'*, but instead of building the tree from scratch, we can start with a subtree with root *s'* of the previous state (root node *s*).

Next, let's focus on the selection step. Although fancier, this is still an RL problem, which means that we'll face the exploration/exploitation dilemma. That is, if we always choose the best action, we might miss out on some path of the tree with lower estimated return, but higher actual return. We can balance between the two with the help of a formula called **Upper Confidence Bounds for Trees (UCT)**. We'll use it to compute the performance values for the state/action pairs in the search tree. Let's assume that we are in the process of building the search tree and we have already played a number of simulations. Then, the UCT formula for action *a*, starting from state *s* in the search tree, is this:

$$u(s,a) = \overline{r_{sa}} + c\frac{\sqrt{\ln N_s}}{n_{sa}}$$

This formula has the following components:

- $\overline{r_{sa}}$ is the average reward received for all previous simulations, which included the edge representing the *(s, a)* state/action pair. This part of the formula represents exploitation: the higher the average reward is, the more likely it is to select the action.
- N_s is the total number of times that state *s* has been visited.

- n_{sa} is the number of simulations that included the action *a*. This number is smaller than N_s, because every simulation that includes *(s, a)* will also include the state *s*. But not every simulation that includes s will include *(s, a)*.
- $\frac{\sqrt{\ln N_s}}{n_{sa}}$ will be higher for actions that participated in fewer simulations, because n_{sa} is in the denominator. This component of the formula represents the exploration.
- *c* is the exploration parameter. It describes the ratio between exploration and exploitation.

Playing board games with AlphaZero

MCTS with UCT is the base for a series of breakthroughs developed by DeepMind. These include the Go-playing AlphaGo, its improved version AlphaGo Zero, and finally AlphaZero (`https://arxiv.org/abs/1712.01815`), an improvement on AlphaGo Zero for playing multiple games, such as Chess and Shogi. Let's discuss AlphaZero. For the sake of simplicity, we'll assume that we want to teach the agent to play chess. Each state of the environment will be one configuration of the board (the positions of the pieces). By taking a turn (moving a piece), the players transition the environment from one state to another.

At the center of the algorithm is a neural network, which takes as input the current state of the board and has two outputs (the network weights are denoted with θ):

- $v_\theta(s) \in [-1, 1]$ is the scalar state-value approximation. The [-1, 1] range symbolizes the chance of victory for the current player at the end of the episode (the end of one game). The value is 1 if the player wins and -1 otherwise.
- $p_\theta(s)$ are the probabilities to take each action, given a current state *s*. In other words, this is the network's policy estimation.

Next, let's take a look at the MCTS part of AlphaZero, which uses a modified version of UCT:

$$u(s,a) = q(s,a) + c\, p_\theta(s,a)\frac{\sqrt{N_s}}{1 + n_{sa}}$$

The formula has the following components:

- $q(s,a)$ is action-value estimation, which is in tabular form and is maintained only for the state/action pairs of the search tree (and not the whole environment). It carries the same meaning as $\overline{r_{sa}}$ in the original UCT formula.
- $p_\theta(s,a)$ is the probability the network assigns to select action *a*, given state *s*.

An important feature of AlphZero is that it replaces the entire simulation step of MCTS with $v_\theta(s)$. When MCTS reaches leaf node *s*, it won't continue with the simulation until a terminal state. Instead, it will simply estimate the total reward of the episode with the $v_\theta(s)$ state-value network output. This value is then propagated back through the tree in the backpropagation step. Once the search tree is ready, we can select our next action. We'll denote the MCTS policy with $\pi(s)$. The score for each action *a*, starting from the root state, is simply:

$$\pi(s,a) = \frac{N_s}{n_{sa}}$$

Where *s* is the root node (our current state), N_s is the total number of simulations, and n_{sa} is the number of simulations that included the action *a*. We'll select the action with the highest score.

Let's observe that in AlphaZero we have two different policy estimations: the action probabilities of the network $p_\theta(s)$ and the MCTS policy $\pi(s)$. To understand the intuition behind this, let's take a look at the training of the neural network. During training, the network plays against itself (self-play); the algorithm powers both players. We'll use the trajectories of the self-play episodes as a training set. For each step of an episode, we have a training tuple, (s_t, π_t, z_t), where s_t is the board state at step *t*, π_t is the MCTS policy for all actions, and z_t is the indication of whether the player has won or lost [-1, 1]. Then, the network loss function for one episode is this:

$$J(\theta) = \sum_{t=0}^{T}((z_t - v_\theta(s_t))^2 - \pi_t \log(p_\theta(s_t))$$

The left part of the equation is just the mean-squared-error between the predicted and the actual results. The right side is the cross-entropy loss between the action predictions of MCTS and the network outputs. Note that the "labels" for the network policy estimation $p_\theta(s)$ are actually the MCTS action probabilities $\pi(s)$ (we can assume they are more accurate), which explains the need for the two estimations.

And that's it. AlphaZero is simpler than AlphaGo, and at the same time it easily defeated the previous state-of-the art models in the games of chess, Shogi, and Go (the previous best in Go was, in fact, AlphaGo).

If you are interested in trying AlphaZero, you can find a general implementation for any board game in this repo: `https://github.com/suragnair/alpha-zero-general`.

Summary

In this chapter, we introduced some advanced RL techniques, starting with deep Q-learning. Then, we used DQN to teach an agent to play the Atari Breakout game with moderate success. Next, we introduced policy-based RL methods, which approximate the optimal policy instead of the true value functions. Then, we used A2C to teach an agent how to play the cart pole game. Finally, we introduced model-based RL methods and MCTS in particular.

In the next chapter, we'll explore how to apply deep learning in the challenging and at the same time exciting area of autonomous vehicles.

10
Deep Learning in Autonomous Vehicles

Let's think about how **autonomous vehicles** (**AVs**) would affect our lives. For one thing, instead of focusing our attention on driving, we'll be able to do something else during our trip. Catering to the needs of such travelers could probably spawn a whole industry in itself. But that's just a side effect. If we can be more productive or just relax during our travels, it is likely that we'll start traveling more. Not to mention the benefits for people with limited ability to drive themselves. Making such an essential and basic commodity as transportation more accessible has the potential to transform our lives. And that's just the effect on us as individuals. AVs can have profound effects on the economy too, starting from delivery services and going to just-in-time manufacturing. In short, making AVs work is a very high-stakes game. No wonder, then, that in recent years the research in this area has transferred from the academic world to the real economy. Companies from Waymo, Uber, and NVIDIA to virtually all major vehicle manufacturers are rushing to develop AVs.

However, we are not there just yet. One of the reasons is that self-driving is a complex task, composed of multiple sub-problems, each a major task in its own right. To navigate successfully, the vehicle "brain" needs an accurate 3D model of the environment. The way to construct such a model is to fuse the signals coming from multiple sensors. Once we have the model, we still need to solve the actual driving task. Think about the many unexpected and unique situations a driver has to overcome without crashing. But even if we create a driving policy, it needs to be accurate almost 100% of the time. Imagine that our AV will successfully stop at 99 out of 100 red traffic lights. 99% accuracy is a great success for any other **machine learning**(**ML**) task. Not so for autonomous driving, where even a single mistake can lead to a crash.

In this chapter, we'll explore the applications of deep learning in AVs. We'll discuss how to use deep networks to help the vehicle make sense of its surrounding environment. We'll also see how to use them in actually controlling the vehicle.

This chapter will cover the following:

- Brief history of AV research
- AV introduction
- Components of an AV system
- Imitation driving policy
- Driving policy with ChauffeurNet
- Deep learning (DL) in the cloud

Brief history of AV research

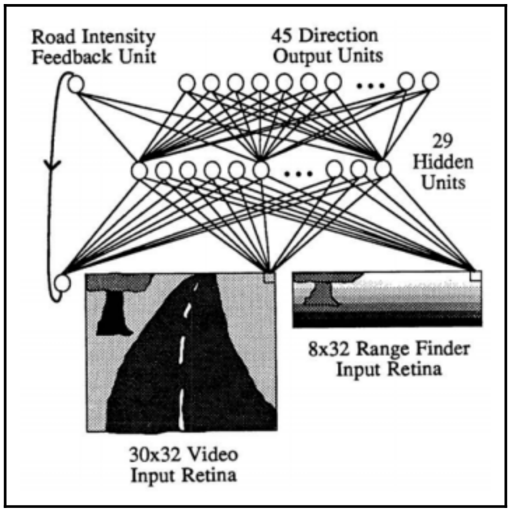
The first serious attempt to implement self-driving cars began in the 1980s in Europe and the USA. Since the mid 2000s, progress has rapidly accelerated. The following is a timeline of some AV research historic points and milestones:

- The first major effort in the area was the Eureka Prometheus Project (`https://en.wikipedia.org/wiki/Eureka_Prometheus_Project`), which lasted from 1987 to 1995. It culminated in 1995, when an autonomous Mercedes-Benz S-Class took a 1,600 km trip from Munich to Copenhagen and back using computer vision. At some points, the car achieved speeds of up to 175 km/h on the German Autobahn (fun fact: some sections of the Autobahn don't have speed restrictions). The car was able to overtake other cars on its own. The average distance between human interventions was 9 km, and at one point it drove 158 km without interventions.
- In 1989, Dean Pomerleau from Carnegie Mellon University published *ALVINN: An Autonomous Land Vehicle in a Neural Network* (`https://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf`), a pioneering paper on the use of neural networks for AVs. This work is especially interesting, as it applied many of the topics we've discussed in this book in AVs nearly 30 years ago.

Let us look at the most important properties of ALVINN:

- It uses a simple neural network to decide the steering angle of a vehicle (it doesn't control the acceleration and the brakes).
- The network is fully connected with one input, one hidden layer, and one output layer.

- The input consists of the following:
    - A 30 x 32 single-color image (they used the blue channel from an RGB image) from a forward-facing camera mounted on the vehicle.
    - An 8 x 32 image from a laser range finder. This is simply a grid, where each cell contains the distance to the nearest obstacle, covered by that cell in the field of view.
    - One scalar input, which indicates the road intensity, that is, whether the road is lighter or darker than the non-road in the image from the camera. This values comes recursively from the network output.
- A single fully-connected hidden layer with 29 neurons.
- A fully-connected output layer with 46 neurons. 45 of those neurons represent the curvature of the road, in a way that resembles one-hot encoding. That is, if the middle neuron has the highest activation, then the road is straight. Conversely, the left and right neurons represent increasing road curvature. The final output unit indicates the road intensity.
- The network was trained for 40 epochs on a dataset of 1,200 images:



The network architecture of ALVINN

There is also an interesting video, showing ALVINN driving a military vehicle in 1992: `https://www.youtube.com/watch?v=ilP4aPDTBPE`.

- The DARPA Grand Challenge (`https://en.wikipedia.org/wiki/DARPA_Grand_Challenge`) was organized in 2004, 2005, and 2007. In the first year, the participating teams' AVs had to navigate a 240 km route in the Mojave Desert. The best performing AV managed just 11.78 km of that route, before getting hung up on a rock. In 2005, the teams had to overcome a 212 km off-road course in California and Nevada. This time, five vehicles managed to drive the whole route. The 2007 challenge was to navigate a mock urban environment, built in an air force base. The total route length was 89 km and the participants had to obey the traffic rules. Six vehicles finished the whole course.
- In 2009, Google started developing self-driving technology. This effort led to the creation of Alphabet's (Google's parent company) subsidiary Waymo (`https://waymo.com/`). In December 2018, they launched the first commercial on-demand ride hailing service with AVs in Phoenix, Arizona.
- Mobileye (`https://www.mobileye.com/`) uses deep neural networks to provide driver-assistance systems (for example, lane keeping assistance). The company has developed a series of **system-on-chip** (**SOC**) devices, specifically optimized to run neural networks with low energy consumption, required for automotive use. Its products are used by many of the major vehicle manufacturers. In 2017, Mobileye was acquired by Intel for $15.3 billion. Since then BMW, Intel, Fiat-Chrysler, and the automotive supplier Delphi have cooperated on joint development of self-driving technology.
- In 2016, General Motors acquired Cruise Automation (`https://getcruise.com/`), a developer of self-driving technology, for more than $500 million (the exact figure is unknown). Since then, Cruise has tested and demonstrated multiple AV prototypes, driving in San Francisco. In October 2018, it was announced that Honda will also participate in the venture by investing $750 million in return for a 5.7% stake.
- Audi's Autonomous Intelligence Driving subsidiary has more than 150 employees developing and testing AV prototypes on the streets of Munich.

AV introduction

When we talk about AVs, we usually imagine fully driverless vehicles. But in reality, we have cars, which require a driver, but still provide some automated features.

The **Society of Automotive Engineers** (**SAE**) has developed a scale of six levels of automation:

- **Level 0**: The driver handles the steering, acceleration, and braking of the vehicle. The features at this level can only provide warnings and immediate assistance to the driver's actions. Examples of features of this level include the following:
    - A lane departure warning simply warns the driver when the vehicle has crossed one of the lane markings.
    - A blind spot warning warns the driver when another vehicle is located in the blind spot area of the car (the area immediately left or right of the rear end of the vehicle).
- **Level 1**: Features that provide either steering or acceleration/braking assistance to the driver. The most popular such features in vehicles today are these:
    - **Lane keeping assist** (**LKA**): The vehicle can detect the lane markings and use the steering to keep itself centered in the lane.
    - **Adaptive cruise control** (**ACC**): The vehicle can detect other vehicles and use brakes and acceleration to maintain a preset speed or reduce it, depending on the circumstances.
    - **Automatic emergency braking** (**AEB**): The vehicle can stop automatically if it detects an obstacle and the driver doesn't react.
- **Level 2**: Features that provide both steering and brake/acceleration assistance to the driver. One such feature is a combination between LKA and adaptive cruise control. At this level, the car can return control to the driver without advance warning at any moment. Therefore, he or she has to maintain constant focus on the road situation. For example, if the lane markings suddenly disappear, the LKA system can prompt the driver to take control of the steering immediately.
- **Level 3**: This is the first level where we can talk about real autonomy. It is similar to level 2 in the sense that the car can drive itself under certain limited conditions and it can prompt the driver to take control. However, this is guaranteed to happen in advance with sufficient time to allow an inattentive person to familiarize themselves with the road conditions. For example, imagine that the car drives itself on the highway, but the cloud-connected navigation obtains information about construction works on the road ahead. The driver will be prompted to take control well in advance of reaching the construction area.
- **Level 4**: Vehicles at this level are fully autonomous in a wider range of situations, compared to level-3. For example, a locally geofenced (that is, limited to a certain region) taxi service could be at level 4. There is no requirement for the driver to take control. Instead, if the vehicle goes outside this region, it should be able to safely abort the trip.
- **Level 5**: Full autonomy under all circumstances. The steering wheel is optional.

All commercially available vehicles today have features at level 2 at most. The only exception (according to the manufacturer) is the 2018 Audi A8, which has a level 3 feature called AI Traffic Jam Pilot. It takes charge of driving in slow-moving traffic at up to 60 km/h on highways and multi-lane roads with a physical barrier separating the two directions of traffic. The driver can be prompted to take control with 10 seconds of advance warning.

Components of an AV system

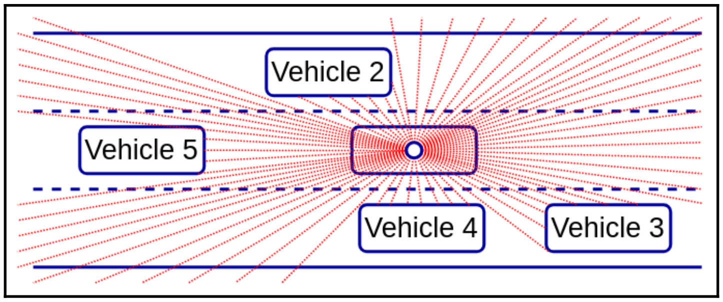
In this section, we'll discuss the building blocks of an AV system.

Sensors

For any automation feature to work, the vehicle needs a good perception of its surrounding environment. The first step in building a good environment model is the vehicle sensors. The following is a list of the most important sensors:

- **Camera**: Its images are used to detect the road surface, lane markings, pedestrians, cyclists, other vehicles, and so on. An important camera property (besides resolution) in the automotive context is the field of view. It measures how much of the observable world the camera sees at a given moment. For example, with a 180^o field of view, it can see everything in front of it and nothing behind. With 360^o, it can see both front and back (full observation). Different types of camera systems exist:
    - Mono camera: Uses a single forward-facing camera, usually mounted on the top of the windshield. Most automation features rely on this type of camera to work. A typical field of view for the mono camera is 125^o.
    - Stereo camera: A system of two forward-facing cameras, slightly removed from each other. The distance between the cameras allows them to capture the same picture from a slightly different angle and combine them in a 3D image (in the same way we use our eyes). A stereo system can measure the distance to some of the objects in the image, while a mono camera relies only on heuristics to do this.
    - Some vehicles have a system of four cameras (front, back, left, and right), which can can assemble a 360^o surrounding view of the environment.
    - Night vision cameras.

- **Radar**: A system that uses a transmitter to emit electromagnetic waves (in the radio or microwave spectrum) in different directions. When the waves reach an object, they are usually reflected, some of them in the direction of the radar itself. The radar can detect them with a special receiver antenna. Since we know that radio waves travel at the speed of light, we can calculate the distance to the reflected object by measuring how much time has passed between emitting and receiving the signal. We can also calculate the speed of an object (for example, another vehicle) by measuring the difference between the frequencies of the outgoing and incoming waves (Doppler effect). The "image" of the radar is noisier, narrower, and with lower resolution, compared to a camera image. For example, a long-range radar can detect objects at a distance of 160m, but in a narrow 12° field of view. The radar can detect other vehicles and pedestrians, but it won't be able to detect the road surface or lane markings. It is usually used for ACC and AEB, while the LKA system uses a camera. Most vehicles have one or two front-facing radars and, on rare occasions, a rear-facing radar.
- **Lidar** (**light detection and ranging**): This sensor is somewhat similar to the radar, but instead of radio waves, it emits a laser in the near-infrared spectrum. Because of this, one emitted pulse can accurately measure the distance to a single point. Lidar emits multiple signals very fast in a pattern, which creates a 3D point cloud of the environment (for example, the sensor can rotate very fast). Following is an illustration of how a vehicle would see the world with a lidar:



An illustration of how a vehicle sees the world through lidar

The data from multiple sensors can be merged into a single environment model with a process called sensor fusion. Sensor fusion is usually implemented with Kalman filters (`https://en.wikipedia.org/wiki/Kalman_filter`).

Deep learning and sensors

Now that we have an idea what sensors the vehicle uses, let's see how to apply deep learning to the raw sensor data. First, we'll do this for the camera. In `Chapter 5`, *Advanced Computer Vision*, we discussed how to use CNNs in two advanced vision tasks: object detection and semantic segmentation. To recap, object detection creates a bounding box around different classes of objects detected in the image. Semantic segmentation assigns a class label to every pixel of the image. We can use segmentation to detect the exact shape of the road surface and the lane markings on the camera image. We can use object detection to classify and localize the objects of interest in the environment. These include other vehicles, pedestrians, bicyclists, traffic signs, traffic lights, and so on.

Next, let's focus on lidar. We can use 3D CNNs (`Chapter 4`, *Computer Vision with Convolutional Networks*) for object detection and segmentation of the lidar point cloud data. This is similar to the way we use a 2D CNN for camera input. An example of these techniques is *Vehicle Detection from 3D Lidar Using Fully Convolutional Network* (`https://arxiv.org/abs/1608.07916`).

Vehicle localization

Localization is the process of determining the exact position of the vehicle on the map. Why is this important? Companies such as HERE (`https://www.here.com/`) specialize in creating extremely accurate road maps, where the entire area of the road surface is known within a few centimeters. Therefore, if we know the exact position of the vehicle on the road, it won't be hard to calculate the optimal trajectory. One obvious solution is to use GPS. However, GPS can be accurate to within 1-2 meters under perfect conditions. In areas with high-rise buildings or mountains, the accuracy can suffer, because the GPS receiver won't be able to get a signal from a sufficient number of satellites. One way to solve this problem is with **simultaneous localization and mapping** (**SLAM**) algorithms. These algorithms are beyond the scope of this book and we encourage you to do your own research on the topic.

Planning

Planning (or driving policy) is the process of calculating the vehicle trajectory and speed. Although we might have an accurate map and exact location of the vehicle, we still need to keep in mind the dynamics of the environment. The car is surrounded by other moving vehicles, pedestrians, traffic lights, and so on. What happens if the vehicle in front stops suddenly? Or if it's moving too slow? Our AV has to make the decision to overtake and then execute the maneuver. This is an area where ML and DL in particular can be especially useful.

One obstacle in AV research is that building an AV and obtaining the necessary permits to test it is very expensive and time consuming. Thankfully, we can still train our algorithms with the help of AV simulators.

Some of the most popular simulators are these:

- Microsoft AirSim, built on the Unreal Engine (`https://github.com/Microsoft/AirSim/`)
- CARLA, built on the Unreal Engine (`https://github.com/carla-simulator/carla`)
- Udacity's Self-Driving Car Simulator, built with Unity (`https://github.com/udacity/self-driving-car-sim`)
- OpenAI Gym's `CarRacing-v0` environment (we'll see an example in the next section, *Imitation driving policy*)

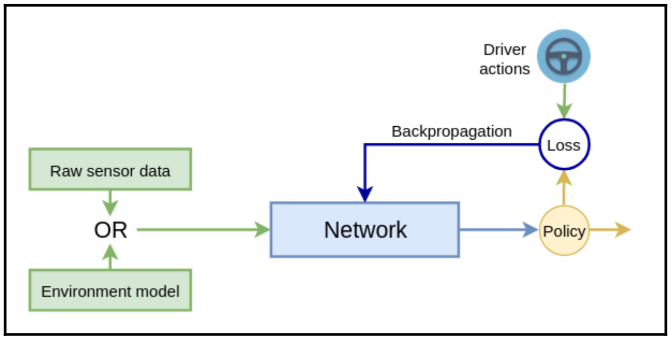
Imitation driving policy

In the section *Components of an AV system* we outlined several modules, necessary for a self-driving system. In this section we'll discuss how to implement one of them - the driving policy - with the help of DL. One way to do this is with RL, where the car is the agent and the environment is, well, the environment. Another popular approach is **imitation learning**, where the model (network) learns to imitate the actions of an expert (human). Let's see the properties of imitation learning in the AV scenario:

- We'll use a type of imitation learning, known as **behavioral cloning**. This simply means that we'll train our network in a supervised way. Alternatively, we have imitation learning in RL scenario, which is known as Inverse RL.
- The output of the network is the driving policy, represented by desired steering angle and/or acceleration or breaking. For example, we can have one regression output neuron for the steering angle and one neuron for acceleration or braking (as we cannot have both at the same time).
- The network input can be either:
    - Raw sensor data. For example, an image from the forward-facing camera. AV systems, where a single model starts from raw sensor inputs and outputs driving policy, are referred to as **end-to-end**.

    - An intermediate environment model created with the help of sensor fusion. In this case we can combine the data from different sensors to produce a top-down (birds eye) 2D view of the environment, similar to the lidar image in section *Sensors*. This approach has several advantages over the end-to-end models. First, instead of using the sensor data to create the top-down image, we can produce it with a simulator. In this way, it will be easier to collect training data, as we won't have to drive the real car. Even more important is that we'll be able to simulate situations, which rarely occur in the real world. For example, our AV have to avoid crashes at any cost, yet a real world training data will have very few, if any, crashes. If we only use real sensor data, one of the most important driving situations will be severely underrepresented.
- We'll create the training dataset with the help of the expert. We'll let them drive the vehicle manually. They could do this in the real world or in the simulator. At each step of the journey, we'll record:
    - The current state of the environment. This could be the raw sensor data or the top-down view representation. We'll use the current state as input to the model.
    - The actions of the expert in the current state of the environment (steering angle and breaking/acceleration). This will be the target data for the network. During training we'll simply minimize the error between the network predictions and the driver actions using the familiar gradient descent. In this way, we'll teach the network to imitate the driver.

Following is an illustration of the behavioral cloning scenario:

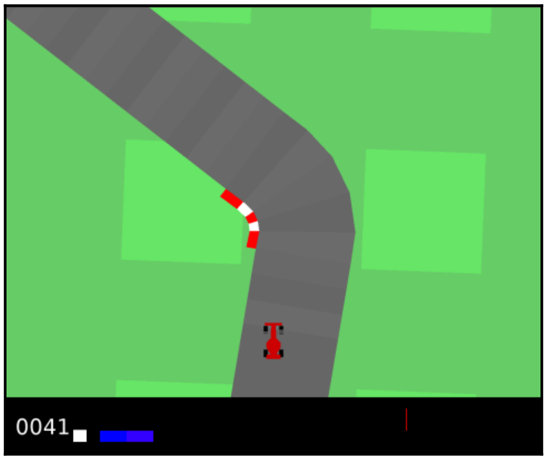


Behavioral cloning scenario.

In fact, we already mentioned behavioral cloning end-to-end system, when we discussed ALVINN (section *Brief history of AV research*). More recently Bojarski et al. (`https://arxiv.org/abs/1604.07316`) introduced similar system, which uses a CNN with 5 convolutional layers instead of a fully-connected network. In their experiment, the images of a forward-facing camera on the vehicle are fed as input to the CNN. The output of the CNN is a single scalar value, which represents the desired steering angle of the car. The network doesn't control acceleration and breaking. To build the training dataset, the collected about 72 hours of real-world driving videos. During the evaluation, the car was able to drive itself 98% of the time in a suburban area (excluding lane changes and turns from one road to another). Additionally, it managed to drive without intervention for 16 km on a multi-lane divided highway.

Behavioral cloning with PyTorch

In this section, we'll implement something fun, a behavioral cloning example with PyTorch. We'll do this with the help of the `CarRacing-v0` OpenAI Gym environment, displayed as follows:



In the CarRacing-v0 environment, the agent is a racing car. A birds-eye view is used the whole time

TIP

This example contains multiple Python files. In this section, we'll mention the most important parts. The full source code lives at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/tree/master/ch10/imitation_learning`.

The goal is for the red racing car to drive around the track as quickly as it can without sliding out of the road surface. We can control the car with four actions: accelerate, brake, and turn left and right. The input for each action is continuous. For example, we can specify full throttle with the value 1.0 and half throttle with the value 0.5 (the same goes for the other controls). For the sake of simplicity, we'll assume that we can only specify two discrete action values: 0 for no action and 1 for full action. Since originally this is an RL environment, the agent will receive an award at each step as it progresses along the track. However, we'll not use them, since the agent will learn directly from our actions. We'll perform the following steps:

1. Create a training dataset by driving the car around the track ourselves (we'll control it with the keyboard arrows). In other words, we'll be the expert, the agent tries to imitate. At every step of the episode, we'll record the current game frame (state) and the currently pressed keys, and we'll store them in a file. The full code for this step is available at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/keyboard_agent.py`. All you have to do is run the file and the game will start. As you play, the episodes will be recorded (once per five episodes) in the `imitation_learning/data/data.gzip` file. If you want to start over, you can simply delete it. You can exit the game by pressing *Escape* and pause with the spacebar. You can also start a new episode by pressing *Enter*. In this case, the current episode will be discarded and its sequence will not be stored. We would advise you to play at least 20 episodes for a sufficient size of the training dataset. It would be good to use the brake more often, because otherwise the dataset will become too imbalanced. In normal play, acceleration is used much more frequently than the brake or the steering. Alternatively, if you don't want to play, the GitHub repository already includes an existing data file.
2. Train a CNN in a supervised manner using the dataset we just generated. The input will be a single game frame (as opposed to the DQN scenario, where we had four frames). The target (labels) will be the action recorded for the human operator. If you want to omit this step, the repository already has a trained PyTorch network located at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/data/model.pt`.
3. Let the CNN agent play by using the network output to determine the next action to send to the environment. You can do this by simply running the `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/nn_agent.py` file. If you haven't performed any of the previous two steps, this file will use the existing agent.

With that introduction, let's start (the following source code is located at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/train.py`).

First, we'll create the training dataset in several steps:

1. The `read_data` function reads `imitation_learning/data/data.gzip` in two `numpy` arrays: one for the game frames and the other for the keyboard combinations associated with them.
2. The environment accepts actions, composed of a three-element array, where the following are true:
    1. The first element has a value in the range [-1, 1] and represents the steering angle (-1 for right, 1 for left).
    2. The second element is the [0, 1] range and represents the throttle.
    3. The third element is in the [0, 1] range and represents the brake power.
3. We'll use the seven most common key combinations: `[0, 0, 0]` for no action (the car is coasting), `[0, 1, 0]` for acceleration, `[0, 0, 1]` for brake, `[-1, 0, 0]` for left, `[-1, 0, 1]` for a combination of left and brake, `[1, 0, 0]` for right, and `[1, 0, 1]` for the right and brake combination. We have deliberately prevented the simultaneous use of acceleration and left or right, as the car becomes very unstable. The rest of the combinations are implausible. `read_data` will convert these arrays to a single class label from 0 to 6. In this way, we'll simply solve a classification problem with seven classes.
4. The `read_data` function will also balance the dataset. As we mentioned, acceleration is the most common key combination, while some of the others, such as brake, are the rarest. Therefore, we'll remove some of the acceleration samples and we'll multiply some of the braking (and left/right + brake). However, the author did this in a heuristic way by trying multiple combinations of deletion/multiplication ratios and selected the ones that work best. If you record your own dataset, your driving style may differ and you may want to modify these ratios.
5. Once we have the `numpy` arrays, we'll convert them to PyTorch `DataLoader` instances with the `create_datasets` function. These classes simply allow us to extract the data in mini-batches and apply data augmentation.

The following is the implementation:

```
def create_datasets():
    """Create training and validation datasets"""

    class TensorDatasetTransforms(torch.utils.data.TensorDataset):
        """
        Helper class to allow transformations
        by default TensorDataset doesn't support them
        """

        def __init__(self, x, y):
            super().__init__(x, y)

        def __getitem__(self, index):
            tensor = data_transform(self.tensors[0][index])
            return (tensor,) + tuple(t[index] for t in self.tensors[1:])

    x, y = read_data()
    x = np.moveaxis(x, 3, 1)  # channel first (torch requirement)

    # train dataset
    x_train = x[:int(len(x) * TRAIN_VAL_SPLIT)]
    y_train = y[:int(len(y) * TRAIN_VAL_SPLIT)]

    train_set = TensorDatasetTransforms(
        torch.tensor(x_train),
        torch.tensor(y_train))

    train_loader = torch.utils.data.DataLoader(train_set,
                                               batch_size=BATCH_SIZE,
                                               shuffle=True,
                                               num_workers=2)

    # test dataset
    x_val, y_val = x[int(len(x_train)):], y[int(len(y_train)):]

    val_set = TensorDatasetTransforms(
        torch.tensor(x_val),
        torch.tensor(y_val))

    val_loader = torch.utils.data.DataLoader(val_set,
                                             batch_size=BATCH_SIZE,
                                             shuffle=False,
                                             num_workers=2)

    return train_loader, val_loader
```

In the preceding code, we have implemented the `TensorDatasetTransforms` helper class to be able to apply the `data_transform` transformations over the input image (defined in `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/util.py`). Before feeding the image to the network, we'll convert it to grayscale, we'll normalize the color values in the [0, 1] range, and we'll crop the bottom part of the frame (the black rectangle, which shows the rewards and other information).

The following is the implementation:

```
data_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Grayscale(1),
    transforms.Pad((12, 12, 12, 0)),
    transforms.CenterCrop(84),
    transforms.ToTensor(),
    transforms.Normalize((0,), (1,)),
])
```

Next, we'll define our CNN, which is very similar to the network we used in the double q-learning example of `Chapter 9`, *Deep Reinforcement Learning for Games*. It has the following properties:

1. A single input 84x84 slice.
2. Three convolutional layers with striding for downsampling.
3. ELU activations.
4. Two fully-connected layers.
5. Seven output neurons (one for each neuron).
6. Batch normalization and dropout, applied after each layer (even the convolutional) to prevent overfitting. Overfitting is not a problem in RL tasks, but it is a real issue in supervised learning. Our problem is particularly exaggerated, because we cannot use any meaningful data augmentation techniques. For example, imagine that we randomly flipped the image horizontally. In this case, we would have to also alter the label to reverse the steering value. Therefore, we'll rely on regularization as much as we can.

The following is the network implementation:

```
def build_network():
    """Build the torch network"""

    class Flatten(nn.Module):
        """
        Helper class to flatten the tensor
```

```
            between the last conv and first fc layer
            """

            def forward(self, x):
                return x.view(x.size()[0], -1)

        # Same network as with the DQN example
        model = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, 8, 4),
            torch.nn.BatchNorm2d(32),
            torch.nn.ELU(),
            torch.nn.Dropout2d(0.5),
            torch.nn.Conv2d(32, 64, 4, 2),
            torch.nn.BatchNorm2d(64),
            torch.nn.ELU(),
            torch.nn.Dropout2d(0.5),
            torch.nn.Conv2d(64, 64, 3, 1),
            torch.nn.ELU(),
            Flatten(),
            torch.nn.BatchNorm1d(64 * 7 * 7),
            torch.nn.Dropout(),
            torch.nn.Linear(64 * 7 * 7, 120),
            torch.nn.ELU(),
            torch.nn.BatchNorm1d(120),
            torch.nn.Dropout(),
            torch.nn.Linear(120, len(available_actions)),
        )

        return model
```

Next, let's implement the training itself with the help of the `train` function. It takes the network and the `cuda` device as parameters. We'll use cross-entropy loss and the Adam optimizer (the usual combination for classification tasks). The function simply iterates `EPOCHS` times and calls the `train_epoch` and `test` functions for each epoch. The following is the implementation:

```
    def train(model, device):
        """
        Training main method
        :param model: the network
        :param device: the cuda device
        """

        loss_function = nn.CrossEntropyLoss()

        optimizer = optim.Adam(model.parameters())
    train_loader, val_order = create_datasets()  # read datasets

    # train
    for epoch in range(EPOCHS):
        print('Epoch {}/{}'.format(epoch + 1, EPOCHS))

        train_epoch(model,
                    device,
                    loss_function,
                    optimizer,
                    train_loader)

        test(model, device, loss_function, val_order)

        # save model
        model_path = os.path.join(DATA_DIR, MODEL_FILE)
        torch.save(model.state_dict(), model_path)
```

Then, we'll implement the `train_epoch` for a single epoch training. This function iterates over all mini-batches and performs forward and backward passes for each one. The following is the implementation:

```
def train_epoch(model, device, loss_function, optimizer, data_loader):
    """Train for a single epoch"""

    # set model to training mode
    model.train()

    current_loss = 0.0
    current_acc = 0

    # iterate over the training data
    for i, (inputs, labels) in enumerate(data_loader):
        # send the input/labels to the GPU
        inputs = inputs.to(device)
        labels = labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        with torch.set_grad_enabled(True):
            # forward
            outputs = model(inputs)
            _, predictions = torch.max(outputs, 1)
            loss = loss_function(outputs, labels)

            # backward
            loss.backward()
```

```
                optimizer.step()

            # statistics
            current_loss += loss.item() * inputs.size(0)
            current_acc += torch.sum(predictions == labels.data)

        total_loss = current_loss / len(data_loader.dataset)
        total_acc = current_acc.double() / len(data_loader.dataset)

        print('Train Loss: {:.4f}; Accuracy: {:.4f}'.format(total_loss,
    total_acc))
```

TIP The `train_epoch` and `test` functions are similar to the ones we implemented for the transfer learning code example of `Chapter 5`, *Advanced Computer Vision*. To avoid repetition, we won't implement the `test` function here, although it's available in the GitHub repository.

We'll run the training for around 100 epochs, but you can shorten this to 20 or 30 epochs for rapid experiments. One epoch usually takes less than a minute using the default training set.

Next, let's implement the `nn_agent_play` function, which allows the agent to play the game (defined in `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/nn_agent.py`). The function will start the `env` environment with an initial state (game frame). We'll use it as an input to the network. Then, we'll convert the softmax network output from one-hot encoding to an array-based action and we'll send it to the environment to make the next step. We'll repeat these steps until the episode ends. `nn_agent_play` also allows the user to exit by pressing *Esc*. Note that we still use the same `data_transform` transformations as we did for the training.

The following is the implementation:

```
def nn_agent_play(model, device):
    """
    Let the agent play
    :param model: the network
    :param device: the cuda device
    """

    env = gym.make('CarRacing-v0')

    # use ESC to exit
    global human_wants_exit
    human_wants_exit = False

    def key_press(key, mod):
        """Capture ESC key"""
        global human_wants_exit
        if key == 0xff1b:  # escape
            human_wants_exit = True

    # initialize environment
    state = env.reset()
    env.unwrapped.viewer.window.on_key_press = key_press

    while 1:
        env.render()

        state = np.moveaxis(state, 2, 0)  # channel first image

        # numpy to tensor
        state = torch.from_numpy(np.flip(state, axis=0).copy())
        state = data_transform(state)  # apply transformations
        state = state.unsqueeze(0)  # add additional dimension
        state = state.to(device)  # transfer to GPU

        # forward
        with torch.set_grad_enabled(False):
            outputs = model(state)

        normalized = torch.nn.functional.softmax(outputs, dim=1)

        # translate from net output to env action
        max_action = np.argmax(normalized.cpu().numpy()[0])
        action = available_actions[max_action]

        # adjust brake power
        if action[2] != 0:
            action[2] = 0.3

        state, _, terminal, _ = env.step(action)  # one step

        if terminal:
            state = env.reset()

        if human_wants_exit:
            env.close()
            return
```

Finally, we can run the whole thing. The full code for this is available at `https://github.com/ivan-vasilev/Python-Deep-Learning-SE/blob/master/ch10/imitation_learning/main.py` . The following snippet builds and restores (if available) the network, runs the training, and evaluates the network:

```
# create cuda device
dev = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# create the network
model = build_network()

# if true, try to restore the network from the data file
restore = False
if restore:
    model_path = os.path.join(DATA_DIR, MODEL_FILE)
    model.load_state_dict(torch.load(model_path))

# set the model to evaluation (and not training) mode
model.eval()

# transfer to the gpu
model = model.to(dev)

# train
train(model, dev)

# agent play
nn_agent_play(model, dev)
```

Although we cannot show the agent in action here, you can easily do so by following the instructions in this section. Still, we can say that it learns well and is able to make full laps of the racing track on a regular basis (but not always). Interestingly, the network's driving style strongly resembles the style of the operator who generated the dataset. The example also goes to show that we shouldn't underestimate supervised learning. We were able to create a decently performing agent with a small dataset and in a relatively short training time. Had we approached this as an RL problem, it would have taken much longer to reach similar results (admittedly, without the need for labeled data).

Driving policy with ChauffeurNet

In this section, we'll discuss a recent paper called *ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst* (`https://arxiv.org/abs/1812.03079`). It was released in December 2018 by Waymo, one of the leaders in the AV space. The following are some of the properties of the ChauffeurNet model:

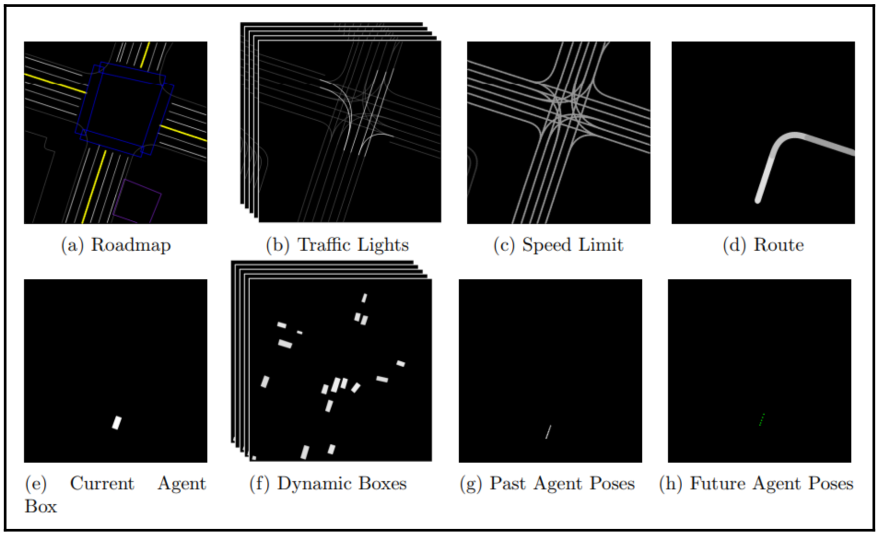
- It is a combination of two interconnected networks. The first is a CNN called FeatureNet, which extracts features from the environment. These features are fed as inputs to a second, recurrent network called AgentRNN, which them to determine the driving policy.
- It uses imitation supervised learning similarly to the algorithms we described in the *Imitation driving policy* section. The training set is generated based on records of real-world driving episodes. ChauffeurNet can handle complex driving situations such as lane changes, traffic lights, traffic signs, changing from one street to another, and so on.

This paper is published by Waymo on arxiv.org and is used here for referential purposes only.
Waymo and arxiv.org are not affiliated, and do not endorse this book, or the authors with Packt.

Model inputs and outputs

Unlike the end-to-end approach, which uses the raw sensor data (for example, camera images), here we'll use the so-called middle-level input. This is a series of top-down (birds-eye) view 400 × 400 images, similar to the images of the `CarRacing-v0` environment but much more complex. One moment of time is represented by multiple images, where each one contains different elements of the environment.

We can see an example of a ChauffeurNet input/output combination in the following diagram:



ChauffeurNet inputs (source: `https://arxiv.org/abs/1812.03079`)

Let's discuss the input/output elements in alphabetical order:

- (a) is a precise representation of the road map. It is an RGB image, which uses different colors to represent various road features such as lanes, cross-walks, traffic signs, and curbs.
- (b) is a temporal sequence of grayscale images of the traffic lights. Unlike the features of (a), the traffic lights are dynamic; that is, they can be green, red, or yellow at different times. In order to properly convey their dynamics, the algorithm uses a series of images, displaying the state of the traffic lights for each lane at each of the past T_{scene} seconds up to the current moment. The gray color of the lines in each image represents the state of each traffic light, where the brightest color is red, intermediate is for yellow, and the darkest is green or unknown.
- (c) is a grayscale image with the known speed limit for each lane. Different color intensities represent different speed limits.

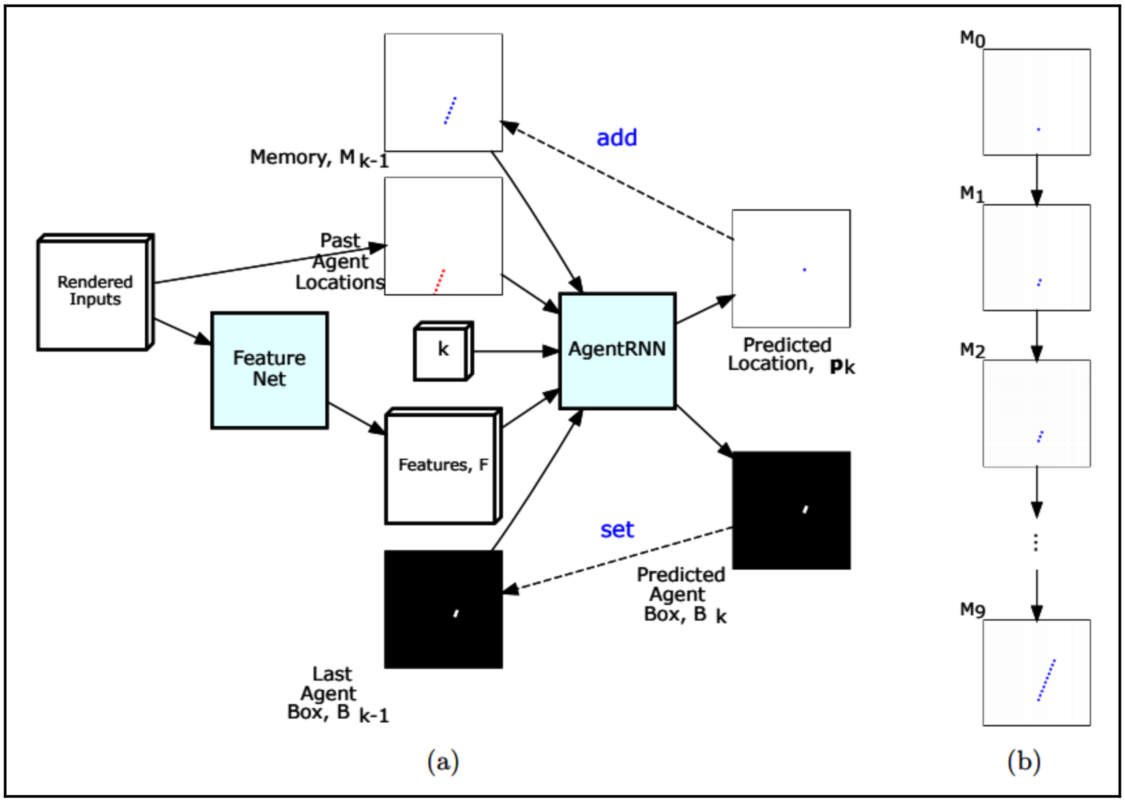
- (d) is the intended route between the start and the destination. Think of it as the directions generated by Google Maps.
- (e) is a grayscale image, which represents the current location of the agent (displayed as a white box).
- (f) is a temporal sequence of grayscale images, which represent the dynamic elements of the environment (displayed as boxes). These could be other vehicles, pedestrians, or cyclists. As these objects change locations over time, the algorithm conveys their trajectories with a series of snapshot images, representing their positions over the last T_{scene} seconds. This works in the same way as the traffic lights (b).
- (g) is a single grayscale image for the agent trajectory of the past T_{pose} seconds until the current moment. The agent locations are displayed as a series of points on the image. Note that we display them in a single image, and not with a temporal sequence like the other dynamic elements.
- (h) is the algorithm output, a series of points that represent the desired locations of the agent's future trajectory. These points carry the same meaning as the past trajectory (g). The newly generated trajectory is fed to the control module of the vehicle, which tries its best to execute it via the vehicle controls (steering, acceleration, and brakes). The future location output at time *t+1* is generated by using the past trajectory (g) up to the current moment *t*. Once we have *t+1*, we can add it to the past trajectory (g) and we can use it to generate the next location at step *t*+2 in a recurrent manner:

$$p_{t+\delta t} = ChauffeurNet(I, p_t)$$

Where *I* are the input images, p_t is the agent position at time *t*, and δt is a 0.2s time delta. The value of δt is arbitrary chosen by the authors of the paper.

Model architecture

The following is the ChauffeurNet model architecture:



(a) ChauffeurNet architecture and (b) the memory updates over the iterations (source: `https://arxiv.org/abs/1812.03079`)

First, we have FeatureNet (in the preceding diagram, (a)). Its inputs are the middle-level top-down images we defined in the *Model inputs and outputs* section. The output of FeatureNet is a feature vector, *F*, which represents the synthesized network understanding of the current environment. This vector serves as one of the inputs to the recurrent network AgentRNN. Let's say that we want to predict the next point of the agent's trajectory (step *k*). Then, AgentRNN has the following outputs:

- p_k is the predicted next point of the driving trajectory at the step *k*. p_k is added to an additive memory, *M*, of the past predictions (p_k, p_{k-1}, ..., p_0) at each step (preceding diagram, (b)). *M* is represented by the input image (g) we defined in section *Model inputs and outputs* section.
- B_k is the predicted bounding box of the agent at the next step *k*.
- Two additional outputs (not displayed in the diagram): θ_k for the heading (or orientation) of the agent and s_k for the desired speed. p_k, θ_k, *and* s_k fully describe the agent in the environment.

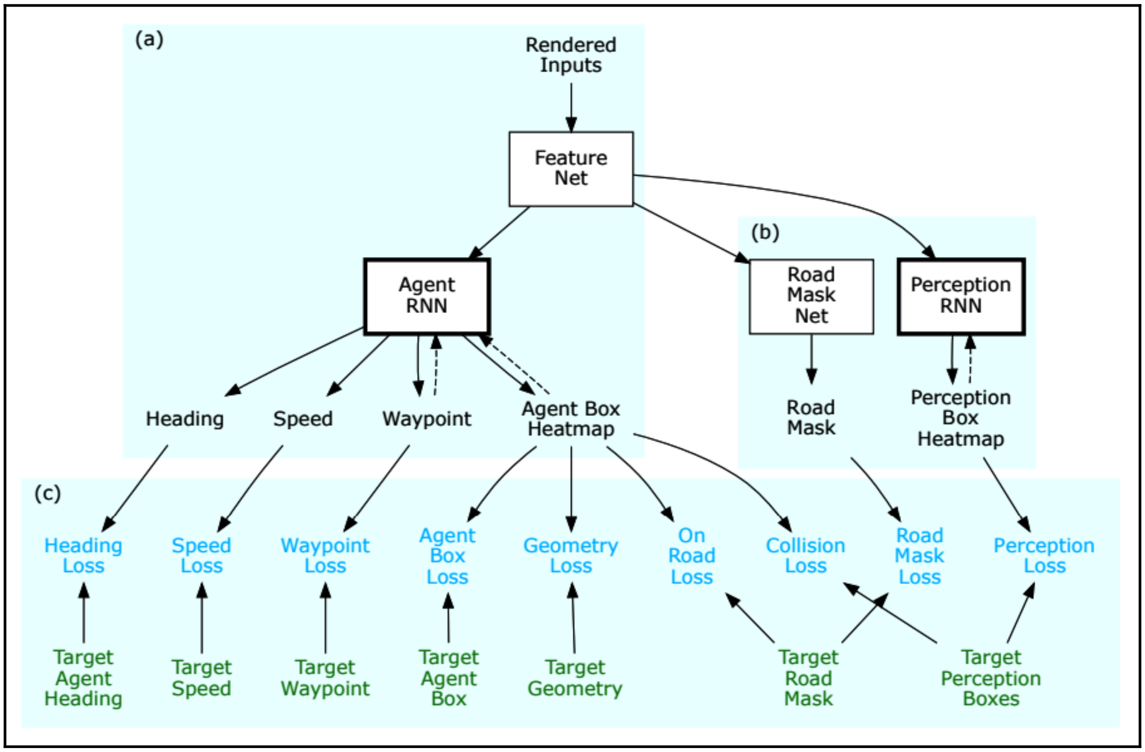
The outputs p_k and B_k are fed back recursively as inputs to AgentRNN for the next step, *k+1*. The formula for the AgentRNN output is as follows:

$$p_k, B_k = AgentRNN(k, F, M_{k-1}, B_{k-1})$$

Training

ChauffeurNet was trained with 30 million expert driving examples, using imitation supervised learning. The middle-level, top-down input allows to use different sources of training data with ease. On one hand, it can be generated from real-world driving with a fusion between the vehicle sensor inputs (cameras, lidar) and mapping data such as streets, traffic lights, traffic signs, and so on. On the other hand, we can generate images of the same format with a simulated environment. As we mentioned in section *Imitiation driving policy*, this allows us to simulate situations that occur rarely in the real world, such as emergency braking or even crashes. To help the agent learn about such situations, the authors of the paper explicitly synthesized multiple rare scenarios using simulation.

The following are the components of the ChauffeurNet training process:



ChauffeurNet training components: (a) the model itself, (b) the additional networks, and (c) the losses (source: https://arxiv.org/abs/1812.03079)

We are already familiar with the ChauffeurNet model itself (a). Let's focus on the two additional networks involved in the process (b). These are the following:

- The Road Mask network predicts a mask with the exact area of the road surface over the current input images.
- PerceptionRNN attempts to predict the future locations of every other dynamic object in the environment (vehicles, cyclists, pedestrians, and so on).

These networks don't participate in the final vehicle control and are used only during training. The intuition to use them is that the FeatureNet network will learn better representations if it receives feedback from the tree tasks (AgentRNN, Road Mask Net, and PerpcetionRNN), compared to simply getting feedback from AgentRNN.

Finally, let's focus on the various loss functions (c). The authors of the paper observed that the imitation learning approach works well when the driving situation does not differ significantly from the expert driving training data. However, the agent has to be prepared for many driving situations that are not part of the training, such as collisions. If the agent only relies on the training data, it will have to learn about collisions implicitly, which is not easy. To solve this problem, the paper proposes explicit loss functions for the most important situations. These include the following:

- **Waypoint loss**: The error between the ground truth and the predicted agent future position, p_k.
- **Speed loss**: The error between the ground truth and the predicted agent future speed, s_k.
- **Heading loss**: The error between the ground truth and the predicted agent future direction, θ_k.
- **Agent box loss**: The error between the ground truth and the predicted agent bounding box, B_k.
- **Geometry loss**: Force the agent to explicitly follow the target trajectory, independent of the speed profile.
- **On-road loss**: Force the agent to navigate only over the road surface area and avoid the non-road areas of the environment. This loss will increase if the predicted bounding box of the agent overlaps with the non-road area of the image, predicted by the road mask network.
- **Collision loss**: Explicitly force the agent to avoid collisions. This loss will increase if the agent's predicted bounding box overlaps with the bounding boxes of any of the other dynamic objects of the environment.

ChauffeurNet performed well in various real-world driving situations. You can see some of the results here: `https://medium.com/waymo/learning-to-drive-beyond-pure-imitation-465499f8bcb2`.

DL in the Cloud

In this chapter, we are discussing a serious topic, AVs and how to apply DL techniques in them. Let's see how to approach this task in practice. First, let's observe that in deep networks (as with most ML algorithms), we have two phases—training and inference. In most production environments, the network is trained once, and then used only in inference mode to solve tasks. If we obtain additional training data during the course of events, we can eventually train the network again (for example, using transfer learning). Then, we can embed the new model in the production environment until we need to retrain it again and so on. The alternative to this is incremental learning, having the model (network) constantly learn from new data, as it comes from the environment.

Although this approach is tempting, it has a few disadvantages, which are as follows:

- As the training is a non-deterministic process, we cannot guarantee whether it won't actually worsen the network performance. For one thing, the network might start to overfit. Alternatively, it can learn from the new data at the expense of forgetting the old one. In other words, training the network online in a production environment can be risky.
- The training process is more computationally intensive compared to inference. It involves forward and backward passes and weight updates, whereas the inference has only a forward pass. Besides taking more computational time, we need additional memory to store the activations of each layer in the forward pass, but also the gradients of the backward pass and the weight updates. Therefore, we can use the model in inference mode with less powerful hardware, compared to training.

It makes even more sense to separate the training and inference in AVs. On one hand, every new network model deployed in the AV needs rigorous testing, to ensure that the car will behave safely. Therefore, it is better to do the training and the evaluation offline. On the other hand, if we want to mass-produce DL hardware for tens of thousands of AVs, it will be more cost effective to produce less powerful hardware, which works only for inference. The cars can simply collect environmental data as they drive, and send this data to a central data center (also referred to as cloud). Once enough new data is collected, a new version of the network model is trained in the center itself. The updated model is sent back to the AVs via **over-the-air** (**OTA**) updates. In this way, the data of all the cars can be combined in a single model, which wouldn’t impossible if each of them was learning separately from their own experience. Admittedly, if the training requires labeled data, the new data still has to be labeled manually.

Some automakers are already adopting similar approach. For example, Tesla can update their Autopilot (the set of driving assistance features they offer) OTA. And we know that the latest models of BMW, Mercedes, and Audi support OTA updates, although it is not known whether they include driving-assistance features.

In this section, we'll focus on how to run DL algorithms in the cloud. We'll do this for two reasons. First, unless we are working at a company such as Waymo, we probably don't have access to an AV to tinker with. And second, cloud-based DL can be useful for all sorts of problems and not just AVs. Thanks to the popularity of DL, many companies offer cloud services to run your models. Two of the biggest providers are, **Amazon Web Services** (**AWS**) and Google's cloud AI products (`https://cloud.google.com/products/ai/`).

Let’s talk about Amazon first. Their most popular service is called **Elastic Compute Cloud** (**EC2**, `https://aws.amazon.com/ec2/`). EC2 works with the so called **Amazon Machine Images** (**AMIs**). These are template configurations, which contain the software specifications of a virtual server, known as an instance. You can launch multiple instances of the same AMI, depending on your needs. Assuming we already have an AWS account, launching a new EC2 instance requires two main steps:

1. Select the desired AMI. Since we are interested in DL, we can select an AMI called **Deep Learning AMI** (Ubuntu). It will launch a virtual server running Ubuntu, which comes with various preinstalled DL libraries including MXNet, TensorFlow, PyTorch, Keras, Chainer, Caffe/2, Theano, and CNTK, configured with NVIDIA CUDA, cuDNN, NCCL, and Intel MKL-DNN. If our DL models work one of these libraries we can immediately start using them on the instance.
2. Choose the instance hardware configuration. AWS offers multiple instance types with different specs including the number of virtual CPUs, memory size, storage size, and network bandwidth. Crucially, some of the instance come with up to 16 NVIDIA K80 GPUs with a combined 192 GB of GPU memory. We should note that to use a GPU instance, you have to obtain permission from Amazon first. That is, the process is not fully automated.

Once an instance is launched, it behaves just like a regular computer. For example, you can access it via `ssh`, install software packages and run applications on it. Amazon will bill you, depending on how much the instance is utilized. Alternatively, we can use the so called EC2 spot instances, which are simply unutilized regular EC2 instances. They are cheaper to use, but come with a small caveat, Amazon can interrupt a spot instance at any time, but it will provide a two-minute warning before doing so. In this way, we can save our progress before the instance is interrupted.

Let's assume that you have logged in the instance via `ssh` and you're using the `bash` terminal. You can test whether the instance is working properly with the help of the source examples of this book.

All you have to do is:

1. Clone the book's GitHub repository with the following command:

    ```
    git clone https://github.com/ivan-vasilev/Python-Deep-Learning-SE/
    ```

2. Navigate to any of the source code examples.

    ```
    cd Python-Deep-Learning-SE/ch10/imitation_learning/
    ```

3. Run the training as follows:

    ```
    python3 train.py
    ```

If everything goes as planned, we should see the how the training progresses through the epochs.

Amazon offers a host of other ML cloud services. You can find out more about them here `https://aws.amazon.com/machine-learning/`.

Next, let's take a look at what Google offers. One of their ML services is the *Cloud Deep Learning VM Image* (`https://cloud.google.com/deep-learning-vm/`). It allows you to configure and launch a virtual server, similar to Amazon EC2. You can choose different hardware parameters including the number of virtual CPU cores, memory size, disk size, and the number of GPUs. Several GPU types are supported, such as NVIDIA K80, P100, and V100. At the time of writing, the DL VM supports TensorFlow, Pytorch, Chainer (`https://github.com/chainer/chainer`), and XGBoost (`https://github.com/dmlc/xgboost`). Once you've deployed your configuration, you can use it like a regular server.

Another interesting ML service from Google is *Cloud TPU* (`https://cloud.google.com/tpu/`), where **TPU** stands for **Tensor Processing Unit**. As we mentioned in `Chapter 1`, *Machine Learning: An Introduction*, these are **application-specific integrated circuits** (**ASICs**), developed by Google and optimized for fast neural network operations. You can use TPUs for your models via the Cloud VM (`https://cloud.google.com/compute/`), which is similar to the DL VM. At present, TPUs only support TensorFlow models (and Keras by extension).

Finally, we should note that using DL in the cloud is often convenient, but it could become expensive over time.

Summary

In this chapter, we explored the applications of deep learning in AVs. We started with a brief historical reference of AV research. Then, we described the components of the AV system and identified when it's appropriate to use DL techniques. Next, we introduced driving policy with behavioral cloning and Waymo's ChauffeurNet. And finally, we introduced DL in the cloud.

This chapter concludes our book. I hope you enjoyed the read!

Index