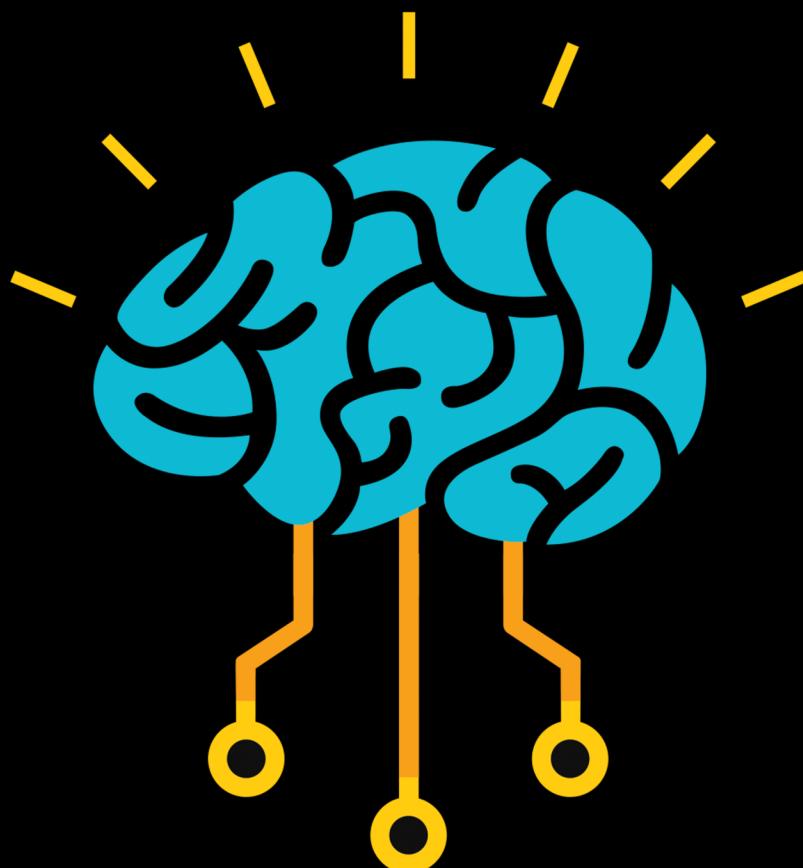


ZERO TO DEEP LEARNING

DEEP LEARNING
FOR THE REST
OF US
WITH KERAS AND TENSORFLOW



FRANCESCO MOSCONI

EDITORS: NATE MURRAY, ARI LERNER

Copyright © 2017 Fullstack.io

Chapter 1: Introduction

About the author

How to use this book

Prerequisites - Is this book right for me?

Deep Learning in the real world

Our development environment

First Deep Learning Model

Exercise 1

Exercise 2

Exercise 3

Exercise 4

Chapter 2: Data Manipulation

Many types of data

Data Exploration with Pandas

Visual data exploration

Unstructured data

Feature Engineering

Exercises

Chapter 3: Machine Learning

The purpose of Machine Learning

Different types of learning

Supervised Learning

Linear Regression

Classification

Overfitting

Cross Validation

Confusion Matrix

Feature Preprocessing

Exercises

Chapter 4: Deep Learning

Beyond linear models

Neural Network Diagrams

Activation functions

Binary classification

Multiclass classification

Conclusion

Exercises

Chapter 5: Deep Learning Internals

This is a special chapter

Derivatives

Backpropagation intuition

Learning Rate

Gradient descent

Gradient calculation in Neural Networks

The math of backpropagation

Fully Connected Backpropagation

Matrix Notation

Gradient descent

Optimizers

Initialization

Inner layer representation

Exercises

Chapter 6: Convolutional Neural Networks

Intro

Machine Learning on images with pixels

Beyond pixels as features

Convolutional Neural Networks

Beyond images

Conclusion

Exercise

Time Series and Recurrent Neural Networks

Time Series

Time series classification

Sequence Problems

Time series forecasting

Improving forecasting

Exercises

Chapter 8: Natural Language Processing and Text Data

Use cases

Text Data

Sequence generation and language modeling

Exercises

Chapter 9: Training with GPUs

Graphical Processing Units

Cloud GPU providers

GPU VS CPU training

Multiple GPUs

Conclusion

Exercise 1

Exercise 2

Chapter 10: Performance Improvement

Learning curves

Reducing Overfitting

Data augmentation

Hyperparameter optimization

Exercises

Chapter 11: Pre-trained models for images

Recognizing sports from images

Keras applications

Predict class with pre-trained Xception

Transfer Learning

Data augmentation

Bottleneck features

Train a fully connected on bottlenecks

Exercises

Learn Deep Learning from nothing all the way to using it professionally.

Francesco Mosconi, Ari Lerner, Nate Murray

Chapter 1: Introduction

Welcome to this practical introduction to Deep Learning. Whether starting our journey with Machine Learning or as a seasoned practitioner looking to add Deep Learning to our set of tools, this course will help! We will gradually start introducing data and Machine Learning problems and then dive deeper into how Neural Networks are built, trained and used. We will deal with tabular data, images, sound, text data and video games and for each of these we will build and train the appropriate Neural Network.

By the end of the course, we will be able to recognize problems that can be solved using Deep Learning, collect and organize data so that it can be used by Neural Networks, build and train models to solve a variety of tasks and finally take advantage of the cloud to train even more powerful models.

ABOUT THE AUTHOR

Francesco Mosconi is an experienced data scientist and entrepreneur. He is owner and chief data scientist at Catalit LLC, a data science consulting and training company based in San Francisco, CA.

He has over 15 years of experience working with data. He started his career with a PhD in Biophysics, publishing a paper on DNA mechanics that currently has over 100 citations. Turning his data skills to business, for a few years he helped companies grow their market through analytics and business development.

Selected by Singularity University to participate to the summer program of 2011, he moved to Silicon Valley and was co-founder and chief data officer of Spire, a Y-Combinator-backed company producing the first consumer wearable device able to measure breathing and state of mind.

After Spire, Francesco founded Catalit and he has been working as data science consultant and trainer for the past years. With Catalit, Francesco has trained engineering teams at some of the largest companies in the world, helping them to acquire skills in Machine Learning and Deep Learning.

He also started a series of workshops on Machine Learning and Deep Learning called Dataweekends, training hundred of students on the same skills.

This book extends and improves the training program of Dataweekends and it provides a practical foundation in Machine Learning and Deep Learning.

HOW TO USE THIS BOOK

This book is meant to provide a self contained introduction to Deep Learning. It assumes basic familiarity with the Python programming language and with a little bit of math.

The first chapters review core concepts in Machine Learning and explain how Deep Learning expands the field. We will build an intuition to recognize the kind of problems where Deep Learning really shines and those where other techniques could provide better results.

Chapters 4-8 present the foundation of Deep Learning. By the end of Chapter 8, we'll be able to use Fully Connected, Convolutional, and Recurrent Neural Networks on your laptop and deal with a variety of input sources including: tabular data, images, time series and text.

Chapters 9-12 build on core chapters to extend the reach of our skills both in depth and in width. We'll learn to improve the performance of our models as well as to use pre-trained models, piggybacking on the shoulders of giants. We will also talk about how to use GPUs to speed up training as well as how to serve predictions from your model.

This is a practical book. Everything we introduce is accompanied by code to experiment with and explore.

The code and notebooks accompanying the book are available through your purchase on your [gumroad library](#). In order to follow along with the book, please be sure to download and unarchive the code on your local computer.

About the exercises

Before we go on, let's spend a couple of words on exercises. They are a key part of this book and we suggest working as follows:

1. Execute the code provided with the chapter, to get a sense of what it is doing.
 - Once we have run the provided code, we suggest to start working through the exercises. We begin with easy exercises and build gradually towards more difficult ones.

If you find yourself stuck, here are some resources where you can look for help:

- look at the error message: understand which parts of it are important. The most important line in the Python error message is the last one. It tells us the error type. The other lines give us information about what caused the error to happen (backtrace), so that we can go ahead and fix it.

- the internet: try pasting part of the error message in a search engine and see what you find. It is very likely that someone has already encountered the same problem and a solution is available.
- stack overflow: this is a great knowledge base where people answer code-related questions. Very often you can just search for the specific error message you got and find the answer here.

Notation

You will find different fonts for different parts of the text:

- **bold** will be used for new terms
- *italic* will be used to emphasize parts of the text that are important
- `fixed width` will be used for code variables
- $math$ will be used for math

We may use tip blocks, like this one:

TIP: this is a tip

to indicate practical suggestions and concepts that add to the material but are not strictly core.

And we may also use math blocks, like this one:

$$x_0 + x_1 + x_2$$

for math formulas.

PREREQUISITES - IS THIS BOOK RIGHT FOR ME?

This book is written for software engineers and developers that want to approach Machine Learning and Deep Learning and understand it. When reviewing current books on the topic we found that they tend to either be very abstract and theoretical with lots of maths and formulas, or too practical, with just code and no explanation of the theory.

In this book we'll try to find a balance between the two. It is an application focused book, with working examples, coding exercises, solutions and real-world datasets. At the same time we won't shy away from the math when necessary. We'll try to keep equations to a minimum, so here are a few symbols you may encounter in the course of the book.

Sum

Sometimes we will need to indicate the sum of many quantities at once. Instead of writing the sum explicitly like this:

$$x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + \dots$$

we may use the summation symbol \sum and write it like this:

$$\sum_i x_i$$

Partial derivatives

Sometimes we will need to indicate the speed of change in a function with respect to one of its arguments. This is obtained through an operation called **partial derivative** and it's indicated with the symbol ∂ like this:

$$\frac{\partial f(x_1, x_2, \dots)}{\partial x_1}$$

It means we are looking at how much f is changing for a unit of change in x_1 when all the other variables are kept fixed (find more about on [Wikipedia](#)).

Dot product

A lot of Deep Learning relies on a few common linear algebra concepts like vectors and matrices. In particular, an operation we will frequently use is the dot product: $A \cdot B$. If you've never seen this before it may be a great time to look up on [Wikipedia](#) how it works.

In any case do not worry too much about the math. We will introduce it gradually and only when needed, so you'll have time to get used to it.

DEEP LEARNING IN THE REAL WORLD

This is a hands-on course where we learn to train Deep Learning models. Such models are omnipresent in the real world and you may have already encountered them without knowing! Both large and small companies use them to solve challenging problems. Here we will mention some of them, but we encourage you to be constantly updated with new applications coming out every day.

Image recognition

This is a very common application, consisting in determine whether or not an image contains some specific objects, features, or activities. For example, the following image shows an object detection algorithm taken from the [Google Blog](#).



The trained model is able to identify the objects in the image. Similar algorithms can be applied to identify faces, or determine diseases from a radiography, or in self driving cars, just to name a few examples.

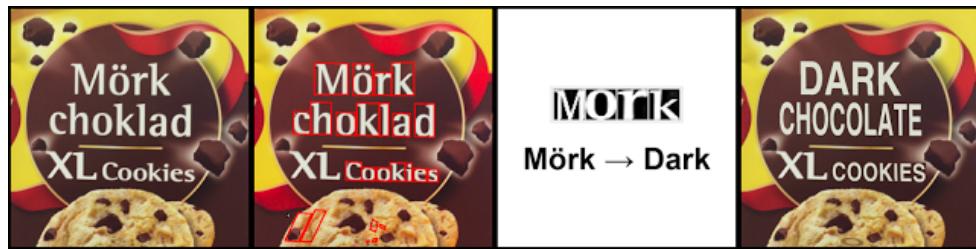
Predictive modeling

Deep Learning may be applied for predictive purposes. Algorithms can be applied to times series of a certain value over time, in order to forecast a future trend. For example the energy consumption of a region, the

temperature over an area, the price of a stock, and so on. Again, Neural Networks can be used for predicting demographics and election results or even earthquakes.

Language translation

Deep Learning can be applied for language translation. This approach uses a large artificial Neural Network to predict the likelihood of a sequence of words, typically modeling entire sentences in a single integrated model. The following image represents an example of an instant visual translation, taken from the [Google Blog](#). This algorithm combines image recognition tasks with language translation ones.



Recommender system

Recommender systems help the user finding the correct choice among the available possibilities. They are everywhere and we use them every day: when we buy a book that Amazon recommends us based on our previous history, or when we listen to that song tailored to our taste in Spotify, or when we watch with the family that movie recommended in Netflix, just to name some examples.

Automatic Image Caption Generation

Automatic image captioning is the task where, given an image, the system can generate a caption that describes the contents of the image. Once you can detect objects in photographs and generate labels for those objects, you can turn those labels into a coherent sentence description.

This is a sample of automatic image caption generation taken from [Andrej Karpathy](#) and Li Fei-Fei at Stanford University.



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"girl in pink dress is jumping in air."



"black and white dog jumps over bar."



"young girl in pink shirt is swinging on swing."

Anomaly detection

Anomaly detection is a technique used to identify unusual patterns that do not conform to expected behavior. It has many applications in business, from intrusion detection (identifying strange patterns in network traffic that could signal a hack) to system health monitoring (spotting a malignant tumor in an MRI scan), and from fraud detection in credit card transactions to fault detection in operating environments.

Automatic Game Playing

This is a task where a Deep Learning model learns how to play a computer game, training to maximize some goals. One glorious example is the DeepMind's [AlphaGo](#) algorithm developed by Google, that beat the world master at the game Go.

OUR DEVELOPMENT ENVIRONMENT

Since this is a practical course, we'll need to get some tools installed on your computer. Whether you have Linux, Mac or Windows, the software required for this course runs on all these systems, so you won't have to worry about it.

We will need to install Python and then we will also install a few libraries that allow us to perform Machine Learning and Deep Learning experiments.

Miniconda Python

Anaconda Python is a great open source distribution of Python packages geared to data science. It is prepackaged with a lot of useful tools and we encourage you to have a look at it. For this book we will not need the full Anaconda distribution, but we will just install the required packages, so that we can keep space requirements to a minimum.

TIP: if you already have Anaconda Python installed, just make sure `conda` is up to date by running `conda update conda` in a terminal window.

We can do this by installing Miniconda, which includes Python and the Anaconda package installer `conda`. Here are the steps to complete in order to install it:

1. Download [Miniconda Python 3.6](#) for your system (Windows/Mac OS X/Linux).
2. Run the installer and make sure that it completes successfully.
3. Done!

If you've completed these steps successfully you can open a command prompt (how to do this will differ depending on which OS you're using) and type `python`. This will launch the Python interpreter and should display something like the following:

```
Python 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

Congratulations! You have just installed Miniconda Python successfully!

Conda Environment

Now that we have installed Python, we need to download and install the packages required to run the code of this book. We will do this in a few simple steps.

1. Open a terminal window.

- Change directory to the folder you have downloaded from our book repository using the command `cd .`
- Run the command `conda env create` to create the environment with all the required packages.

You will see a lot of lines on your terminal. What is going on? The `conda` package installer is creating an **environment**, i.e. a special folder that contains the specific versions of each required package for the book. The environment is specified in the file `environment.yml`, that looks like this:

```
name: ztdlbook
channels:
- defaults
dependencies:
- python==3.6
- pip==9.0.3
- numpy==1.14.2
- jupyter==1.0.0
- matplotlib==2.2.2
- scikit-learn==0.19.1
- scipy==1.0.1
- pandas==0.22.0
- pillow==5.0.0
- seaborn==0.8.1
- h5py==2.7.1
- pytest==3.5.0
- pip:
- tensorflow==1.8.0
- keras==2.1.6
- jupyter_contrib_nbextensions==0.5.0
```

The package installer reads the environment file and downloads the correct versions of each package including its dependencies. This is why you see a lot more packages being downloaded.

Once the environment is created you should see a message like the following (Mac/Linux):

```
#  
# To activate this environment, use:  
# > source activate ztdlbook  
#  
# To deactivate an active environment, use:
```

```
# > source deactivate  
#
```

or the following (Windows):

```
#  
# To activate this environment, use:  
# > activate ztdlbook  
#  
# To deactivate an active environment, use:  
# > deactivate ztdlbook  
#
```

This tells you how to activate and deactivate the environment.

You can read more about conda environments [here](#).

So let's go ahead and activate the environment by typing `source activate ztdlbook` (or the Windows equivalent). If you do that, you'll notice that your command prompt changes and now displays the environment name at the beginning, within parentheses, like this:

```
(ztdlbook) <other stuff your prompt usually shows> $
```

If you see this, you can go ahead to the next step.

Jupyter notebook

Now that we have installed Python and the required packages, let's explore the code for this book. Code is provided as notebooks. Jupyter Notebooks are documents that can contain live code, equations, visualizations and explanatory text. They are very common in the data science community because they allow for easy prototyping of ideas and fast iteration. Notebook files are opened and edited through the Jupyter Notebook web application. Let's launch it!

TIP: For a complete introduction to the jupyter notebook we encourage you to have a look at the official documentation.

In the terminal, change directory to the course folder (if you haven't already) and then type:

```
jupyter notebook
```

This will start the notebook server and open a window in your default browser, and you should reach a window like this:

The screenshot shows the Jupyter Notebook dashboard. At the top, there is a logo and a 'Logout' button. Below the header, there are three tabs: 'Files' (selected), 'Running', and 'Clusters'. A message 'Select items to perform actions on them.' is displayed above a file list. The file list includes:

	Name	Last Modified
<input type="checkbox"/>	apps	2 minutes ago
<input type="checkbox"/>	assets	25 days ago
<input type="checkbox"/>	course	3 minutes ago
<input type="checkbox"/>	data	4 minutes ago
<input type="checkbox"/>	solutions	4 minutes ago
<input type="checkbox"/>	tests	30 minutes ago
<input type="checkbox"/>	environment.yml	16 minutes ago

At the top right of the file list, there are 'Upload' and 'New' buttons.

This is called the **notebook dashboard** and serves as a home page for the notebook. There are three tabs in the dashboard:

- *Files* : this tab displays the notebooks and files in the current directory. By clicking on the breadcrumbs or on sub-directories at the top of notebook list, you can navigate your file system. Additional files may be created either uploaded. To create a new notebook, click on the “New” button at the top of the list and select a kernel from the dropdown. To upload a new file, click on the “Upload” button and browse the file from your computer. By selecting a notebook file, you can perform several tasks, such as “Duplicate”, “Shutdown”, “View”, “Edit” or “Delete” it.
- *Running* : this tab displays the currently running Jupyter processes, either a Terminals or Notebooks. This tab is important to shutdown running notebook: in fact Notebooks remain running until you explicitly shut them down, and closing the notebook’s page is not sufficient.
- *Cluster* : this tab displays parallel process, provided by [IPython parallel](#) and it requires further activation, not necessary for the scope of this book.

Great! Now we are ready to start running the code, but first, a word on the exercises.

Jupyter notebook cheatsheet

Let us summarize here a few very useful commands to get you started with jupyter notebook.

- `Ctrl-ENTER` executes the currently active cell and **keeps** the cursor on the same cell
- `Shift-ENTER` executes the currently active cell and **moves** the cursor on the same cell
- `ESC` enables the Command Mode. Try it. You'll see the **border of the notebook change to Blue**. In Command Mode you can press a single key and access many commands. For example, use:
 - `A` to insert cell **above** the cursor
 - `B` to insert cell **below** the cursor
 - `DD` to delete the current cell
 - `F` to open the **find/replace** dialogue
 - `z` to undo the last command

Finally you can use `H` to access the help dialog with all the keyboard shortcuts for both command and edit mode:

Keyboard shortcuts

X

The Jupyter Notebook has two different keyboard input modes. **Edit mode** allows you to type code or text into a cell and is indicated by a green cell border. **Command mode** binds the keyboard to notebook level commands and is indicated by a grey cell border with a blue left margin.

Mac OS X modifier keys:

: Command

: Control

: Option

: Shift

: Return

: Space

: Tab

Command Mode (press `Esc` to enable)

[Edit Shortcuts](#)

: find and replace

: enter edit mode

: open the command palette

: open the command palette

: open the command palette

: run cell, select below

: run selected cells

: run cell and insert below

: extend selected cells below

: insert cell above

: insert cell below

: cut selected cells

: copy selected cells

: paste cells above

: paste cells below

: undo cell deletion

Environment check

If you have followed the instructions this far you should be running the first notebook.

The next command cell makes sure that you are using the Python executable from within the course environment and should evaluate without an error.

If you get an error, try the following:

1. Close this notebook.
2. Go to the terminal and stop jupyter notebook.
3. Make sure that you have activated the environment, you should see a prompt like:

```
(ztdlbook) $
```

4. (Optional) if you don't see that prompt activate the environment:

- mac/linux:

```
source activate ztdlbook
```

- windows:

```
activate ztdlbook
```

5. Restart jupyter notebook.

6. Re-run the next cell.

```
import os
import sys

env_name = 'ztdlbook'

p = sys.executable
try:
    assert(p.find(env_name) != -1)
    print("Congrats! You are using the correct environment.")
except Exception as ex:
    print("It seems you are not using the correct environment.\n",
          "Currently running Python from this path:\n",
          p,
          "\n",
          "Please follow the instructions above and retry.")
    raise ex
```

```
Congrats! You are using the correct environment.
```

Python 3.6

The next line checks that you're using Python 3.6.x from Anaconda and it should execute without any error.

If you get an error, go back to the previous step and make sure you created and activated the environment correctly.

```
python_version = '3.6'
distribution = 'Anaconda'

v = sys.version
try:
    assert(v.find(python_version) != -1)
    assert(v.find(distribution) != -1)
    print("Congrats! You are using the version of python.")
except Exception as ex:
    print("It seems you are not using the correct version of python.\n",
          "Currently running Python from this path:\n\n",
          v,
          "\n\n",
          "Please follow the instructions above\n",
          "and make sure you are in the activated environment.")
raise ex
```

Congrats! You are using the version of python.

Jupyter

Let's check that Jupyter is running from within the environment.

```
import jupyter
j = jupyter.__file__

try:
    assert(j.find('jupyter') != -1)
    assert(j.find(env_name) != -1)
    print("Congrats! You are using the jupyter from within the environment.")
except Exception as ex:
    print("It seems you are not using the correct version of jupyter.\n",
          "Currently running Python from this path:\n\n",
          j,
          "\n\n",
          "Please follow the instructions above\n",
```

```
"and make sure you are in the activated environment.")  
raise ex
```

Congrats! You are using the jupyter from within the environment.

Other packages

Here we will check that all the packages are installed and have the correct versions. If everything is ok you should see:

```
Using TensorFlow backend.
```

```
Houston we are go!
```

If there's any issue here please make sure you have checked the previous steps.

```
import pip  
import numpy  
import jupyter  
import matplotlib  
import sklearn  
import scipy  
import pandas  
import PIL  
import seaborn  
import h5py  
import tensorflow  
import keras  
import pytest  
  
def check_version(pkg, version):  
    try:  
        assert(pkg.__version__ == version)  
    except Exception as ex:  
        print("{} {}\\t=> {}".format(pkg.__name__, version, pkg.__version__))  
        raise ex  
  
check_version(pip, '9.0.3')  
check_version(numpy, '1.14.2')  
check_version(matplotlib, '2.2.2')  
check_version(sklearn, '0.19.1')
```

```
check_version(scipy, '1.0.1')
check_version(pandas, '0.22.0')
check_version(PIL, '5.0.0')
check_version(seaborn, '0.8.1')
check_version(h5py, '2.7.1')
check_version(pytest, '3.5.0')
check_version(tensorflow, '1.8.0')
check_version(keras, '2.1.6')

print("Houston we are go!")
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: Fu
  from ._conv import register_converters as _register_converters
```

Houston we are go!

Using TensorFlow backend.

Congratulations! You have just verified that you have correctly set up your computer to run the code in this book.

Troubleshooting installation

If for some reason you encounter errors while running the first notebook, the simplest solution is to delete the environment and start from scratch again.

To remove the environment:

- close the browser and go back to your terminal
- stop jupyter notebook (CTRL-C)
- deactivate the environment (Mac/Linux):

```
source deactivate ztdlbook
```

- deactivate the environment (Windows 10):

```
deactivate ztdlbook
```

- delete the environment:

```
conda remove -y -n ztdlbook --all
```

- restart from environment creation and make sure that each steps completes till the end.

Updating Conda

One thing you can also try is to update your conda executable. This may help if you already had Anaconda installed on your system.

```
conda update conda
```

These instructions have been tested on:

- Mac OSX Sierra 10.12.6
- Ubuntu 16.04
- Windows 10

FIRST DEEP LEARNING MODEL

Now that we are set up and ready to go, let's get our feet wet with our first simple Deep Learning model. Let's begin by separating 2 sets of points of different color in a two dimensional space.

First we are going to create these two sets of points, and then we will build a model that is able to separate them. Although it's a toy example, this is representative of many industry relevant problems where a binary prediction is requested from the model.

TIP: Binary prediction

When we talk about binary prediction, we mean identifying one type or another. In this example, we're separating between blue and red parts. True or false, 0 or 1, etc. are other outcomes of binary prediction.

Question: Can you think of any industry examples where we may want to predict a binary outcome?

Answer: detecting if an email is spam or not, detecting if a credit card transaction is legitimate or not, predicting if a user is going to buy an item or not...

The primary goal of this exercise is to see that, with a few lines of code we can sufficiently define and train a Deep Learning model. Do not worry if some of it is beyond your understanding yet, we'll walk through it and see code similar to it in the rest of the book in-depth.

In the next chapters, we will be building more complex models and we will work with more interesting datasets.

First, we are going to import a few libraries.

Numpy

At their core, Neural Networks are mathematical functions. The workhorse library used industry-wide is `numpy`. `numpy` is a Python library that contains many mathematical functions, particularly around working with arrays of numbers.

For instance, `numpy` contains functions for:

- vector math
- matrix math
- operations optimized for number arrays

While we'll use higher-level libraries such as Keras a lot in this book, being familiar with and proficient in using `numpy` is a core skill we'll need to build (and evaluate) our networks. Also, while `numpy` is a comprehensive library, there are only a few key functions that we will use over and over again. We'll cover each new function as it comes up, so let's dive in and try out a few basic operations.

Basic Operations

The first thing we need to do to use `numpy` is `import` it into our workspace:

```
import numpy as np
```

TIP: If you get an error message similar to this:

```
-----
ImportError                                     Traceback (most recent call last)
<ipython-input-1-4ee716103900> in <module>()
      1 import numpy as np
      2
----> 3 ImportError: No module named 'numpy'
```

Refer back to the installation section and make sure you have activated the environment before starting `jupyter notebook`.

TIP: Python error messages

Python error messages are generally not easy to navigate. > To understand the error, we suggest reading from the bottom up. Usually the last line is the most informative

Using `numpy`, let's create a simple 1-D array:

```
a = np.array([1, 3, 2, 4])
```

Here we've created a 1-dimensional array containing four numbers. We can evaluate `a` to see the current values:

```
a
```

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

Note that the type of `a` is `numpy.ndarray`. The documentation for this type is available here.

```
type(a)
```

```
numpy.ndarray
```

TIP: Jupyter notebook is a great interactive environment and it offers a lot of help when we want to quickly check the documentation for an object. This can be accessed by appending a question mark to any variable in our notebook.

For example, in the next cell, try typing:

```
a?
```

and execute the cell.

As you have noticed, this opens a pane in the bottom with the documentation for the object `a`. This trick can be used with any object in the notebook. Pretty awesome!

Press escape to dismiss the panel at the bottom.

Let's create two more arrays, a 2-D and a 3-D array:

```
b = np.array([[8, 5, 6, 1],
              [4, 3, 0, 7],
              [1, 3, 2, 9]])

c = np.array([[[1, 2, 3],
               [4, 3, 6]],
              [[8, 5, 1],
               [5, 2, 7]],
              [[0, 4, 5],
               [8, 9, 1]],
              [[1, 2, 6],
               [3, 7, 4]]])
```

Again we can evaluate them to check that they are indeed what we expect them to be:

```
b
```

```
array([[8, 5, 6, 1],  
       [4, 3, 0, 7],  
       [1, 3, 2, 9]])  
...  
[[1, 3, 2, 9]])  
...  
[[1, 3, 2, 9]])
```

```
c
```

```
array([[[1, 2, 3],  
        [4, 3, 6]],  
  
       [[8, 5, 1],  
        [5, 2, 7]],  
  
       [[[0, 4, 5],  
         [8, 9, 1]],  
  
       ...  
       [3, 7, 4]]])
```

In mathematical terms, we can think of the 1-D array as a *vector*, the 2-D array as a *matrix* and the 3-D array as a tensor of order 3.

TIP: What is a Tensor?

We will encounter tensors later in the book and we will give a more precise definition of them then. For the time being, we can think of a Tensor as a more general version of a matrix, which can have more than 2 indices for its elements. Another useful way to think of tensors is to imagine them as a list of matrices of equal shape.

Numpy arrays are objects, which means they have attributes and methods. A useful attribute is `shape`, which tells us the number of elements in each dimension of the array:

```
a.shape
```

```
(4, )
```

Python here tells us the object has 4 elements along the first axis. The trailing comma is needed in Python to indicate that the object is a *tuple* with only one element.

TIP: In Python, if we write `(4)`, it is interpreted as the number `4` surrounded by parentheses and the parentheses are neglected. Typing `(4,)` is interpreted as a tuple with a single element: the number `4`. If the tuple contains more than one element, like `(3, 4)` we can omit the trailing comma.

Tab tricks

Trick 1: Tab completion

In jupyter notebook we can type faster by using the `tab` key to complete any variable name that has been created previously. So, for example, if somewhere along our code we have created a variable called `verylongclumsynamevariable` and we need to use it again, we can simply start typing `ver` and hit `tab` to see the possible completions, including our long and clumsy variable.

TIP: try to use meaningful and short variable names, to make your code more readable.

Trick 2: Methods & Attributes

In jupyter notebook, we can hit `tab` after the dot `.` to know which methods and attributes are accessible for a certain object. For example try typing

```
a.
```

and then hit `tab`. You will notice that a small window pops up with all the methods and attributes available for the object `a`. This looks like:

```
In [ ]: a.|
```

- a.all
- a.any
- a.argmax
- a.argmin
- a.argpartition
- a.argsort
- a.astype
- a.base
- a.byteswap
- a.choose

This is very handy if we are looking for inspiration about what a can do.

Trick 3: Documentation pop-up

Let's go ahead and select `a.argmax` from the tab menu by hitting `enter`. This is a method, and we can quickly find how it works. Let's open a **round parenthesis** `a.argmax(` and let's hit `SHIFT+TAB+TAB` (that is `TAB` two times in a row while holding down `SHIFT`). This will open a pop-up window with the documentation of the `argmax` function.

```
a.argmax(  
    ^ x  
  
Docstring:  
a.argmax(axis=None, out=None)  
  
Return indices of the maximum values along the given axis.  
  
Refer to `numpy.argmax` for full documentation.  
  
See Also  
-----  
numpy.argmax : equivalent function  
- https://tinyurl.com/yd7w3yjz
```

Here you can read how the function works, which inputs it requires and what outputs are returned. Pretty nice!

Let's look at the shape of b :

b. shape

(3, 4)

Since `b` is a 2-dimensional array, the attribute `.shape` has two elements one for each of the 2 axes of the matrix. In this particular case we have a matrix with 3 rows and 4 columns or a 2×4 matrix.

Let's look at the shape of `c`:

```
c.shape
```

(4, 2, 3)

`c` has 3 dimensions. Notice how the last element indicates the length of the innermost axis, in fact `shape` is a tuple, whose elements are ordered from the outer list to the inner list.

TIP: Knowing how to navigate the shape of an `ndarray` is important. Most of the work we will encounter later, from being able to perform a dot product between weights and inputs in a model to correctly reshaping images when feeding them to a convolutional Neural Network.

Selection

Now that we know how to *create* arrays, we also need to know how to extract data out of them. You can access elements of an array using the square brackets. For example, we can select the first element in `a` by doing:

```
a[0]
```

1

Remember that `numpy` indices start from `0` and the element at any particular index can be found by `n-1`. For instance, the first element can be extracted by referencing the cell at `a[0]` and the second element at `a[1]`.

Uncomment the next line and select the second element of `b`.

```
# arr = np.array([4, 3, 0, 7])
# assert (second element of arr)
```

Unlike accessing arrays in, say, JavaScript, `numpy` arrays have a powerful selection notation that you can use to read data in a variety of ways.

For instance, we can use commas to *select along multiple axes*. For example, here's how we can get the first sub-element of the third element in `c`:

```
c[2, 0]
```

```
array([0, 4, 5])
```

What about selecting all the first elements along the second axis in `b`? That's achieved with the `:` operator:

```
b[:, 0]
```

```
array([8, 4, 1])
```

Since `b` is a 2-D array, this is equivalent to selecting the first column.

`:` is the delimiter of the slice syntax to select a sub-part of a sequence, like: `[begin:end]`.

For example, we can select the first 3 elements in `a` by typing:

```
a[0:2]
```

```
array([1, 3])
```

and we can select the upper left 2x2 sub-matrix in `b` as:

```
b[:1, :1]
```

```
array([[8]])
```

Try selecting the elements requested in the next few lines.

Select the second and third elements of `a`:

```
# uncomment and complete the next line
# assert ( your code here == np.array([3, 2]))
```

Select the elements from 1 to the end in `a`:

```
# uncomment and complete the next line
# assert ( your code here == np.array([3, 2, 4]))
```

Select all the elements from the beginning excluding the last one:

```
# uncomment and complete the next line
# assert ( your code here == np.array([1, 3, 2]))
```

Stride

We can also select regularly spaced elements by specifying a step size after a second `:`. For example, to select the first and third element in `a` we can type:

```
a[0:-1:2]
```

```
array([1, 2])
```

or, simply:

```
a[::2]
```

```
array([1, 2])
```

where it is implicit that we want start and end to be the first and last element in the array.

Math

We'll try to keep the math at a minimum here, but we do need to understand how the various operations work in an array.

Math operators work element-wise, meaning that the mathematical operation is performed on *all* of the elements and their corresponding element locations.

For instance, let's say we have two variables of shape `(2,)`.

```
one = np.array([1, 2])
two = np.array([3, 4])
```

Addition works here by adding `one[0]` and `two[0]` together, then adding `one[1]` and `two[1]` together:

```
one + two # array([4, 6])
```

```
3 * a
```

```
array([ 3,  9,  6, 12])
```

```
a + a
```

```
array([2, 6, 4, 8])
```

```
a * a
```

```
array([ 1,  9,  4, 16])
```

```
a / a
```

```
array([1., 1., 1., 1.])
```

```
a - a
```

```
array([0, 0, 0, 0])
```

```
a + b
```

```
array([[ 9,  8,  8,  5],  
       [ 5,  6,  2, 11],  
       [ 2,  6,  4, 13]])  
...  
      [ 2,  6,  4, 13]])  
...  
      [ 2,  6,  4, 13]])
```

```
a * b
```

```
array([[ 8, 15, 12,  4],  
       [ 4,  9,  0, 28],  
       [ 1,  9,  4, 36]])  
...  
      [ 1,  9,  4, 36]])  
...  
      [ 1,  9,  4, 36]])
```

Go ahead and play a little to make sure we understand how these work.

TIP: If you're not familiar with the difference between element-wise multiplication and dot product, checkout these two links:

- Hadamard product
- Matrix multiplication

As mentioned in the beginning, `numpy` is a very mature library that allows us to perform many operations on arrays including:

- vectorized mathematical functions
- masks and conditional selections
- matrix operations
- aggregations
- filters, grouping, selects
- random numbers
- zeros and ones

and much more.

We will introduce these different operations as needed. The curious reader is referred to [this documentation](#) for more information.

Matplotlib

Another library we will use extensively is Matplotlib. The [Matplotlib](#) library is used to plot graphs so that we can visualize our data. Visualization is an important step in Machine Learning. Throughout this book we will use different kinds of plots in many situations, including

- Visualizing the shape / distributions of our data.
- Inspecting the performance improvement of our networks as training progresses.
- Visualizing pairs of features to look for correlations.

Let's have a look at how to generate the most common plots available in `matplotlib`.

```
%matplotlib inline  
import matplotlib.pyplot as plt
```

Above we've imported `matplotlib.pyplot`, which gives us access to the `matplotlib` functionality.

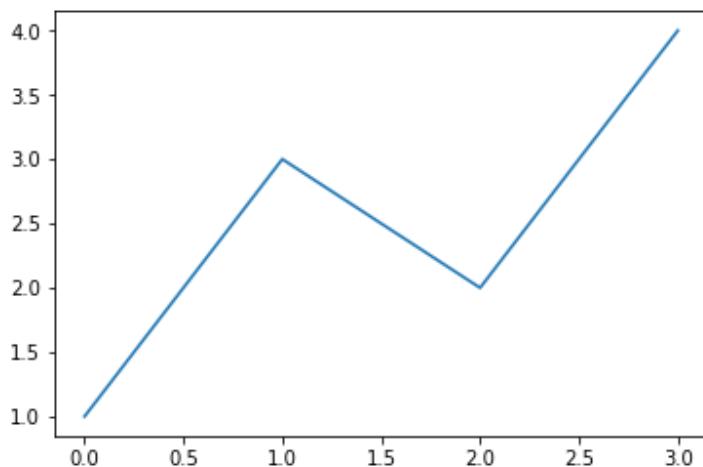
TIP: Because this course is written in Jupyter, we add the line `%matplotlib inline`, which will render our figures in-line. Find some other options [here](#).

Plot

To plot some data with a line plot, we can just call the `plot` function on that data. We can try plotting our `a` vector from above like so:

```
plt.plot(a)
```

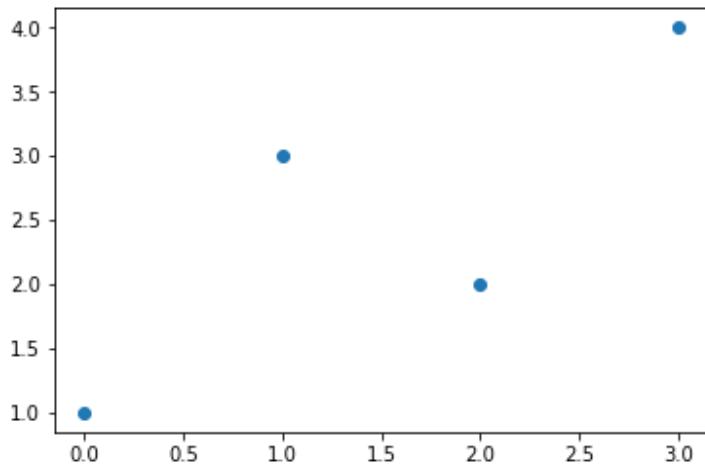
```
[<matplotlib.lines.Line2D at 0x7f18001866d8>]
```



We can also render a scatter plot, by specifying a symbol to use to plot each point.

```
plt.plot(a, 'o')
```

```
[<matplotlib.lines.Line2D at 0x7f179dc41198>]
```

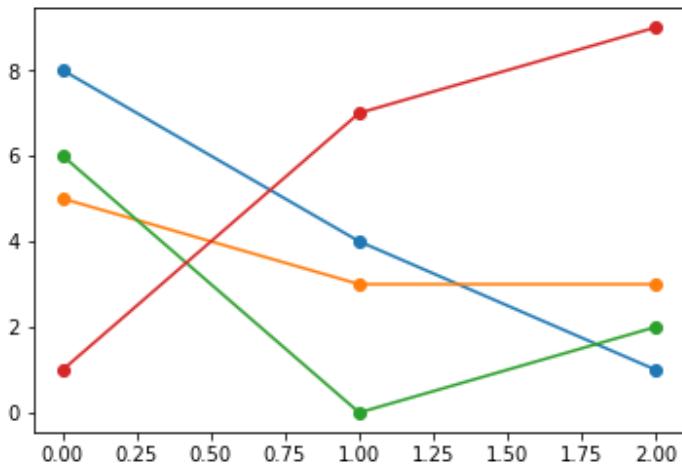


In the plot below, we show how 2-D Arrays are interpreted as *tabular data*, i.e. data arranged in a table with rows and columns. Each row is interpreted as 1 data point and each column as a coordinate for that data point.

If we plot `b` we will obtain 4 curves, one for each coordinate, with 3 points each.

```
plt.plot(b, 'o-')
```

```
[<matplotlib.lines.Line2D at 0x7f179dba4a90>,
 <matplotlib.lines.Line2D at 0x7f179dba4ba8>,
 <matplotlib.lines.Line2D at 0x7f179dba4cf8>,
 <matplotlib.lines.Line2D at 0x7f179dba4e48>]
...
<matplotlib.lines.Line2D at 0x7f179dba4e48>]
...
<matplotlib.lines.Line2D at 0x7f179dba4e48>]
```



Notice that the 4 lines in the graph are plotted in the two dimensional graph where the first line has a point at $(0, 8)$, one at $(1, 4)$, and another at $(2, 1)$, which maps to the columns of `b`.

Let's take another look at `b`:

```
b
```

```
array([[8, 5, 6, 1],
       [4, 3, 0, 7],
       [1, 3, 2, 9]])
...
[1, 3, 2, 9]])
...
[1, 3, 2, 9]])
```

`b` has the shape $(3, 4)$. If we want to plot 3 lines with 4 points each we need to swap the rows with the columns.

We do that by using the `transpose` function:

```
b.transpose()
```

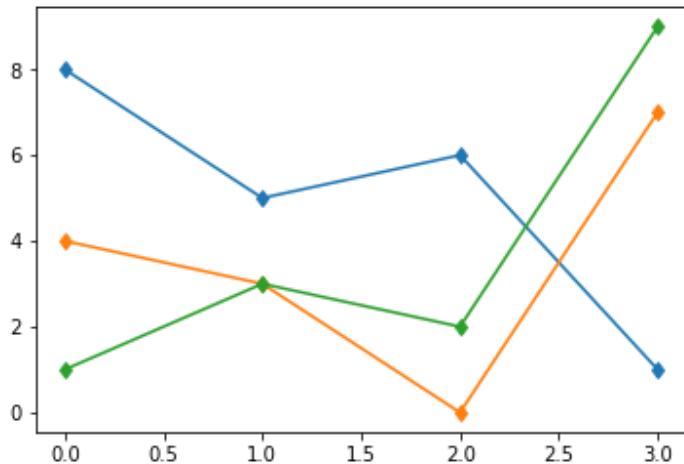
```
array([[8, 4, 1],
       [5, 3, 3],
       [6, 0, 2],
       [1, 7, 9]])
```

```
...
[1, 7, 9]])
...
[1, 7, 9]])
```

Now we can pass `b.transpose()` to `plot` and plot the 3 lines:

```
plt.plot(b.transpose(), 'd-')
```

```
[<matplotlib.lines.Line2D at 0x7f179db8e320>,
 <matplotlib.lines.Line2D at 0x7f179db8e438>,
 <matplotlib.lines.Line2D at 0x7f179db8e588>]
...
<matplotlib.lines.Line2D at 0x7f179db8e588>]
...
<matplotlib.lines.Line2D at 0x7f179db8e588>]
```



Notice how we used a special marker and also added the line between points. `matplotlib` contains a variety of functions that let us create detailed, sophisticated plots.

We don't need to understand all of the operations up-front, but for an example of the power Matplotlib provides, let's look at a more complex plot example (don't worry about understanding every one of these functions, we'll cover the ones we need later on):

```
plt.figure(figsize = (9, 6))
```

```

plt.plot(b[0], color='green', linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=12 )
plt.plot(b[1], 'D-', markersize=12 )

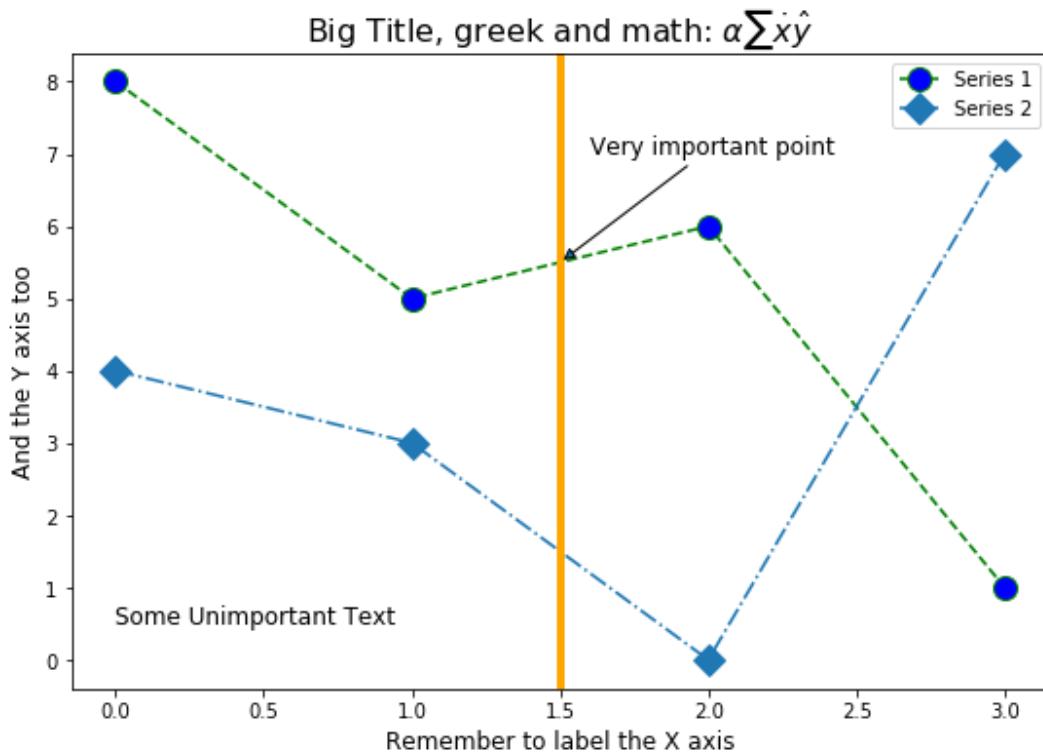
plt.xlabel('Remember to label the X axis', fontsize=12)
plt.ylabel('And the Y axis too', fontsize=12)
plt.title(r'Big Title, greek and math: $\alpha \sum \dot{x} \cdot y$', fontsize=16)

plt.axvline(1.5, color='orange', linewidth=4)
plt.annotate(xy=(1.5, 5.5), xytext=(1.6, 7),
            s="Very important point",
            arrowprops={"arrowstyle": '->'},
            fontsize=12)
plt.text(0, 0.5, "Some Unimportant Text", fontsize=12)

plt.legend(['Series 1', 'Series 2'])

```

<matplotlib.legend.Legend at 0x7f179daff978>



TIP This gallery gives more examples of what's possible with Matplotlib.

Scikit-Learn

Scikit-learn is a wonderful library for many Machine Learning algorithms in Python. We will use it here to generate some data

```
from sklearn.datasets.samples_generator import make_circles

X, y = make_circles(n_samples=1000,
                     noise=0.1,
                     factor=0.2,
                     random_state=0)
```

The `make_circles` function will generate two "rings" of data points, each with 2 coordinates. It will also generate an array of *labels*, either 0 or 1.

TIP: *label* is an important term in Machine Learning, that we'll be explained better in classification problems. Basically, it is a number indicating the class the data belongs to.

We assigned these to the variables `x` and `y`. This is a very common notation in Machine Learning.

- `x` indicates the input variable, and it is usually an array of dimension ≥ 2 with the outer index running over the various data points in the set.
- `y` indicates the output variable, and it can be an array of dimension ≥ 1 . In this case our data will belong to either one circle or to the other and therefore our output variable will be binary: either 0 or 1. In particular, the data points belonging to the inner circle will have a label of 1.

Let's take a look at the raw data we generated in `x`:

```
x
```

```
array([[ 0.24265541,  0.0383196 ],
       [ 0.04433036, -0.05667334],
       [-0.78677748, -0.75718576],
       ...,
       [ 0.0161236 , -0.00548034],
       [ 0.20624715,  0.09769677],
       [-0.19186631,  0.08916672]])
```

```
...  
[-0.19186631,  0.08916672]])
```

We can see the raw generated labels in `y`:

```
y[:10]
```

```
array([1, 1, 0, 1, 1, 0, 0, 0, 1])
```

We can also check the shape of `X` and `y` respectively:

```
X.shape
```

```
(1000, 2)
```

```
y.shape
```

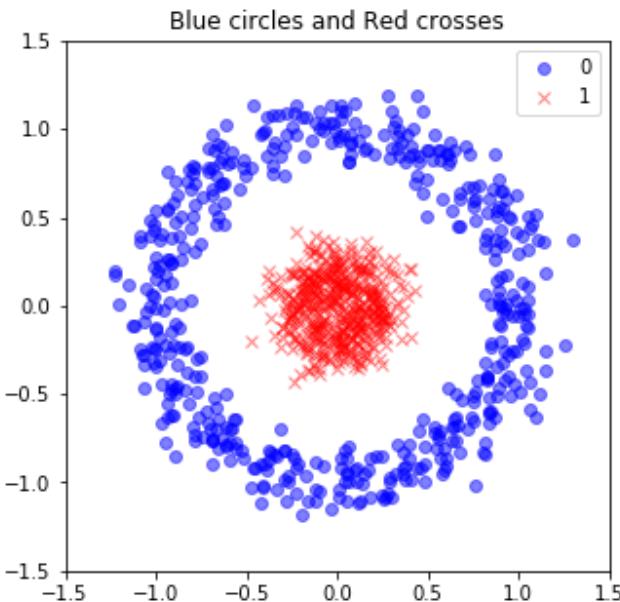
```
(1000, )
```

While we are able to investigate the individual points, it becomes a lot clearer if we plot the points visually using `matplotlib`.

Here's how we do that:

```
plt.figure(figsize=(5, 5))  
plt.plot(X[y==0, 0], X[y==0, 1], 'ob', alpha=0.5)  
plt.plot(X[y==1, 0], X[y==1, 1], 'xr', alpha=0.5)  
plt.xlim(-1.5, 1.5)  
plt.ylim(-1.5, 1.5)  
plt.legend(['0', '1'])  
plt.title("Blue circles and Red crosses")
```

```
Text(0.5, 1, 'Blue circles and Red crosses')
```



Notice that we used some transparency, controlled by the parameter `alpha`.

TIP: what does the `x[y==0, 0]` syntax do? Let's break it down:

- `x[,]` is the multiple axis selection operator, so we will be selecting along rows and columns in the 2D `x` array.
- `x[:, 0]` would select all the elements in the first column of `x`. If we interpret the 2 columns of `x` as being the coordinates along the 2 axes of the plot, we are selecting the coordinates along the horizontal axis.
- `y==0` returns a boolean array of the same length as `y`, with `True` at the locations where `y` is equal to 0 and `False` in the remaining locations. By passing this boolean array in the row selector, `numpy` will smartly choose only those rows in `x` for which the boolean condition is `True`.

Thus `x[y==0, 0]` means: select all the data points corresponding to the label `0` and for each of these select the first coordinate, then return all these in an array.

Notice also how we are using the keywords `color` and `marker` to modify the aspect of our plot.

When we look at this plot we can see that points are spread on the plane in two concentric circles, the blue dots forming a larger circle on the outside and the red crosses a smaller circle on the inside. Although this plot is artificially created, it's representative of any situations where we want to separate two classes that are not separable with a straight line.

For example in the next chapters we will try to distinguish between fake and true banknotes or between different classes of wine, and in all these cases the boundary between a class and the other will not be a straight line.

In this toy example, we want to train a simple Neural Network model to learn to separate the blue circles from the red crosses.

Time to import our Deep Learning library **Keras**!

Keras

Keras is the Deep Learning library we will use throughout the book. It's modular, well designed, and it has been integrated by both Google and Microsoft to serve as the high level API for their Deep Learning libraries (if you're not familiar with APIs, you may have a look on [Wikipedia](#)).

TIP: Do not worry about understanding every line of code of what follows. The rest of the book is dedicated to walking through how to use Keras and [Tensorflow](#) (a very powerful open-source ML library developed by Google), and so we're not going to explain every detail here.

Here we're going to demonstrate an overview of *how* to use Keras and we'll describe more details as the book progresses.

To train a model to tell the difference between red crosses and blue dots above, we have to perform the following steps:

1. Define our Neural Network structure, this is going to be our model.
2. Train the model on our data.
3. Check that the model has correctly learned to separate the red crosses from the blue dots.

TIP: If this is the first time you train a Machine Learning model, do not worry, we will repeat these steps many times throughout the book and we'll have plenty of opportunities to familiarize ourselves with them.

Let's start by importing a few libraries:

```
from keras.models import Sequential  
from keras.layers import Dense  
from keras.optimizers import SGD
```

Let's start with step one: defining a Neural Network model. The next 4 lines are all that's necessary to do just that.

Keras will interpret those lines and behind the scenes create a model in Tensorflow . In fact, we may have noticed that the above cells informed us that Keras is "Using Tensorflow backend". In fact Keras is just a high level API specification that can work with several back-ends. For this course, we will use it with the Tensorflow library as back-end.

The Neural Network below will take 2 inputs (the horizontal and vertical position of a data point in the plot above) and return a single value: the probability that the point belongs to the the "Red Crosses" in the inner circle.

Let's build it!

We start by creating an empty shell for our model. We do this using the Sequential class. This tells keras that we are planning to build our model sequentially, adding one component at a time. So we will start by declaring the model to be a sequential model and then we will proceed adding elements to the model.

TIP: Keras also offers a functional API to build models. This is a bit more complex and we will introduce it later in the book. Most of the models in this book will be built using the Sequential API.

```
model = Sequential()
```

The next step is to add components to our model. We won't explain the meaning of each of these lines now, except pointing your attention to 2 facts:

1. We are specifying the input shape of our model input_shape=(2,) in the first line below, so that our model will expect 2 input values for each data point.
2. We have one output value only which will give us the predicted probability for a point to be a blue dot or a red cross. This is specified by the number 1 in the second line below.

```
model.add(Dense(4, input_shape=(2,), activation='tanh'))  
model.add(Dense(1, activation='sigmoid'))
```

Finally we need to compile the model, which will communicate to our backend (Tensorflow) the model structure and how it will learn from examples. Again, don't worry about knowing what optimizer and loss function mean, we'll have plenty of time to understand those.

```
model.compile(SGD(lr=0.5), 'binary_crossentropy', metrics=['accuracy'])
```

Defining the model is like creating an empty box where there are no meaningful data points defined in the model. We can think of it like wiring up a circuit. To get any meaningful data points in our model, we'll need to feed some example data to the model, so that it can learn general rules to separate the red crosses from the blue dots.

This is done using the `fit` method. We'll discuss this in great detail in the chapter on Machine Learning.

```
model.fit(X, y, epochs=20)
```

```
Epoch 1/20
1000/1000 [=====] - 1s 1ms/step - loss: 0.6992 - acc: 0.5670
Epoch 2/20
1000/1000 [=====] - 0s 74us/step - loss: 0.6692 - acc: 0.7110
Epoch 3/20
1000/1000 [=====] - 0s 74us/step - loss: 0.6108 - acc: 0.8140
Epoch 4/20
1000/1000 [=====] - 0s 75us/step - loss: 0.5077 - acc: 0.8370
Epoch 5/20
...
Epoch 5/20
```

```
<keras.callbacks.History at 0x7f179c7e5898>
```

The `fit` function just ran 20 *rounds* or *passes* over our data. Each *round* is called an *epoch*. At each epoch we pass our data through the Neural Network and compare the known labels with the predictions from the network and measure how accurate our net was.

After 20 iterations the accuracy of our model is 1 or close to 1, meaning 100% (or close to 100%) of the test cases were predicted correctly. This means our prediction is spot-on.

Decision Boundary

Now that our model is trained, we can feed it with any pair of numbers and it will generate a prediction for the probability that a point situated on the 2D plane at those coordinates belongs to the group of red crosses.

In other words, now that we have a trained model, we can ask for the probability to be in the group of "red crosses" for any point in the 2D plane. This is great because we can see if it has correctly learned to draw a boundary between red crosses and blue dots. One way to calculate this is to draw a grid on the 2D plane and calculate the probability predicted by the model for any point on this grid. Let's do it!

TIP: Don't worry if you don't yet understand everything in the following code. It is important that you get the general idea.

Our data varies roughly between `-1.5` and `1.5` along both axes, so let's build a grid of equally spaced horizontal lines and vertical lines between these 2 extremes.

We will start by building 2 arrays of equally spaced points between the `-1.5` and `1.5`. The `np.linspace` function does just that.

```
hticks = np.linspace(-1.5, 1.5, 101)
vticks = np.linspace(-1.5, 1.5, 101)
```

```
hticks[:10]
```

```
array([-1.5 , -1.47, -1.44, -1.41, -1.38, -1.35, -1.32, -1.29, -1.26,
       -1.23])
...
       -1.23])
...
       -1.23])
```

Now let's build a grid with all the possible pairs of points from `hticks` and `vticks`. The function `np.meshgrid` does that.

```
aa, bb = np.meshgrid(hticks, vticks)
```

```
aa.shape
```

```
(101, 101)
```

```
aa
```

```

array([[-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
      [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
      [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
      ...,
      [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
      [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
      [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ]])

...
[-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ]])

...
[-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ]])

```

bb

```

array([[-1.5 , -1.5 , -1.5 , ..., -1.5 , -1.5 , -1.5 ],
      [-1.47, -1.47, -1.47, ..., -1.47, -1.47, -1.47],
      [-1.44, -1.44, -1.44, ..., -1.44, -1.44, -1.44],
      ...,
      [ 1.44, 1.44, 1.44, ..., 1.44, 1.44, 1.44],
      [ 1.47, 1.47, 1.47, ..., 1.47, 1.47, 1.47],
      [ 1.5 , 1.5 , 1.5 , ..., 1.5 , 1.5 , 1.5 ]])

...
[ 1.5 , 1.5 , 1.5 , ..., 1.5 , 1.5 , 1.5 ]])

...
[ 1.5 , 1.5 , 1.5 , ..., 1.5 , 1.5 , 1.5 ]])

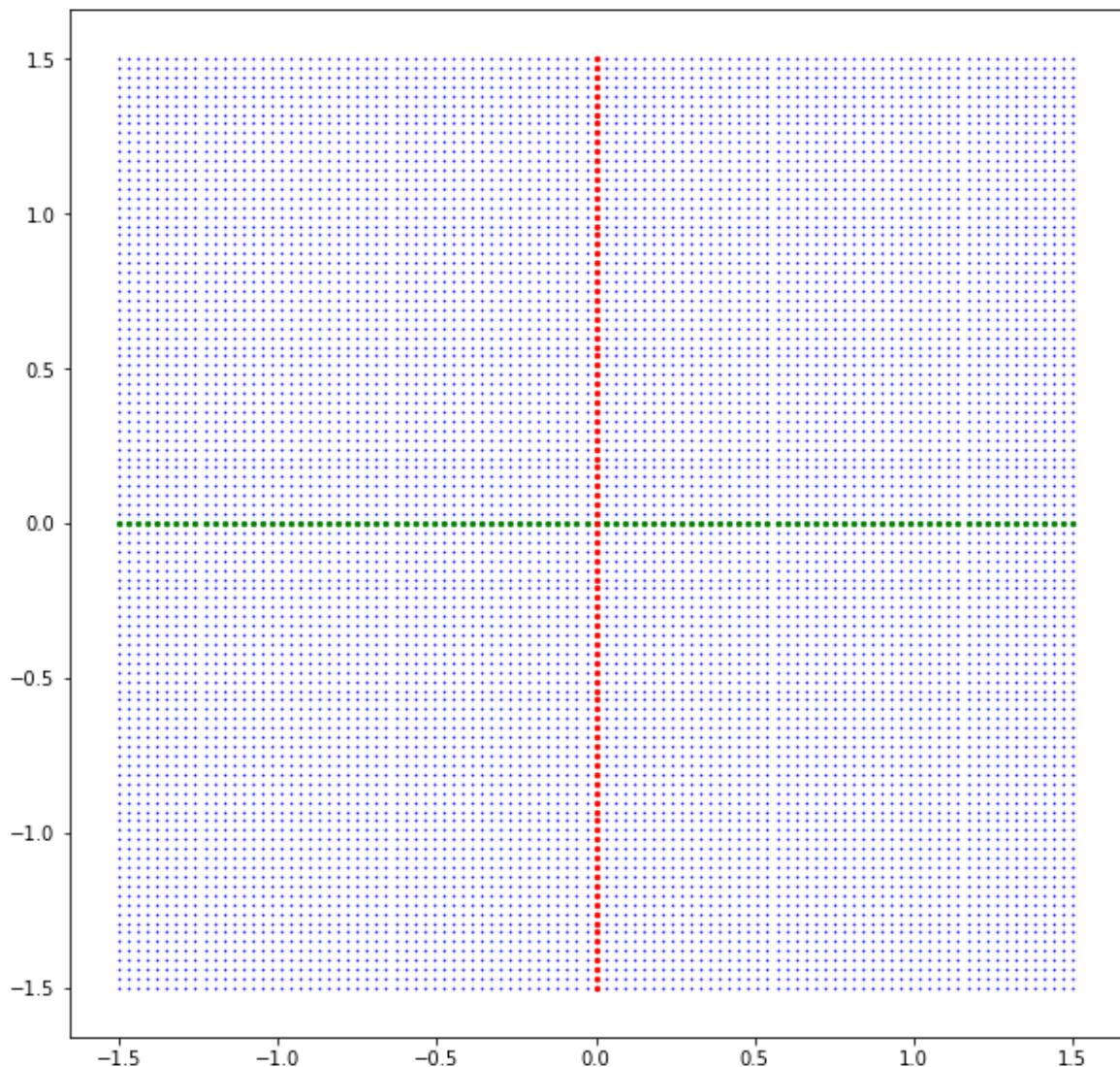
```

aa and bb contain the points of the grid, we can visualize them:

```

plt.figure(figsize=(10,10))
plt.scatter(aa, bb, s=0.3, color='blue')
# highlight one horizontal series of grid points
plt.scatter(aa[50], bb[50], s=5, color='green')
# highlight one vertical series of grid points
plt.scatter(aa[:, 50], bb[:, 50], s=5, color='red')
plt.show()

```



The model expects a pair of values for each data point, so we have to re-arrange `aa` and `bb` into a single array with 2 columns.

The `ravel` function flattens an N-dimensional array to a 1D array and the `np.c_` class will help us combine `aa` and `bb` into a single 2D array.

```
ab = np.c_[aa.ravel(), bb.ravel()]
```

We can check that the shape of the array is correct:

```
ab.shape
```

```
(10201, 2)
```

We have created an array with `10201` rows and 2 columns, these are all the points on the grid we drew above. Now we can pass it to the model and obtain a probability prediction for each point in the grid.

```
c = model.predict(ab)
```

```
c
```

```
array([[3.2427990e-05],  
       [4.1033061e-05],  
       [5.3884374e-05],  
       ...,  
       [6.4070925e-02],  
       [6.4089544e-02],  
       [6.4104326e-02]], dtype=float32)  
...  
[6.4104326e-02]], dtype=float32)  
...  
[6.4104326e-02]], dtype=float32)
```

Great! We have predictions from our model for all points on the grid, and they are all values between 0 and 1.

Let's check to make sure that they are, in fact between 0 and 1 by checking the minimum and maximum values:

```
c.min()
```

```
2.0521287e-05
```

```
c.max()
```

```
0.99104977
```

Let's reshape `c` so that it has the same shape as `aa` and `bb`. We need to do this so that we will be able to use it to control the size of each dot in the next plot

```
c.shape
```

```
(10201, 1)
```

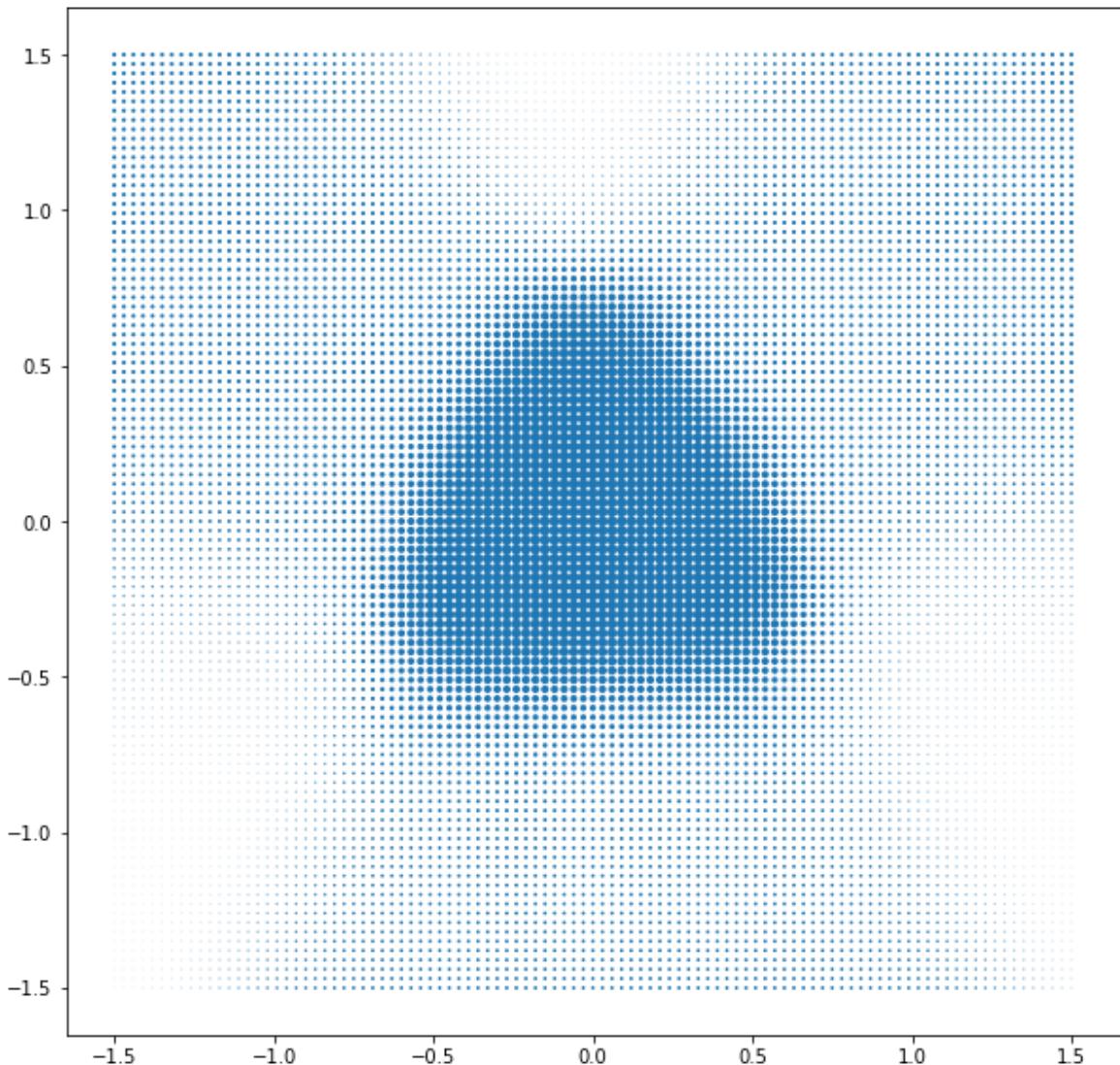
```
cc = c.reshape(aa.shape)
cc.shape
```

```
(101, 101)
```

Let's see what they look like! We will redraw the grid, making the size of each dot proportional to the probability predicted by the model that that point belongs to the group of red crosses

```
plt.figure(figsize=(10, 10))
plt.scatter(aa, bb, s=20*cc)
```

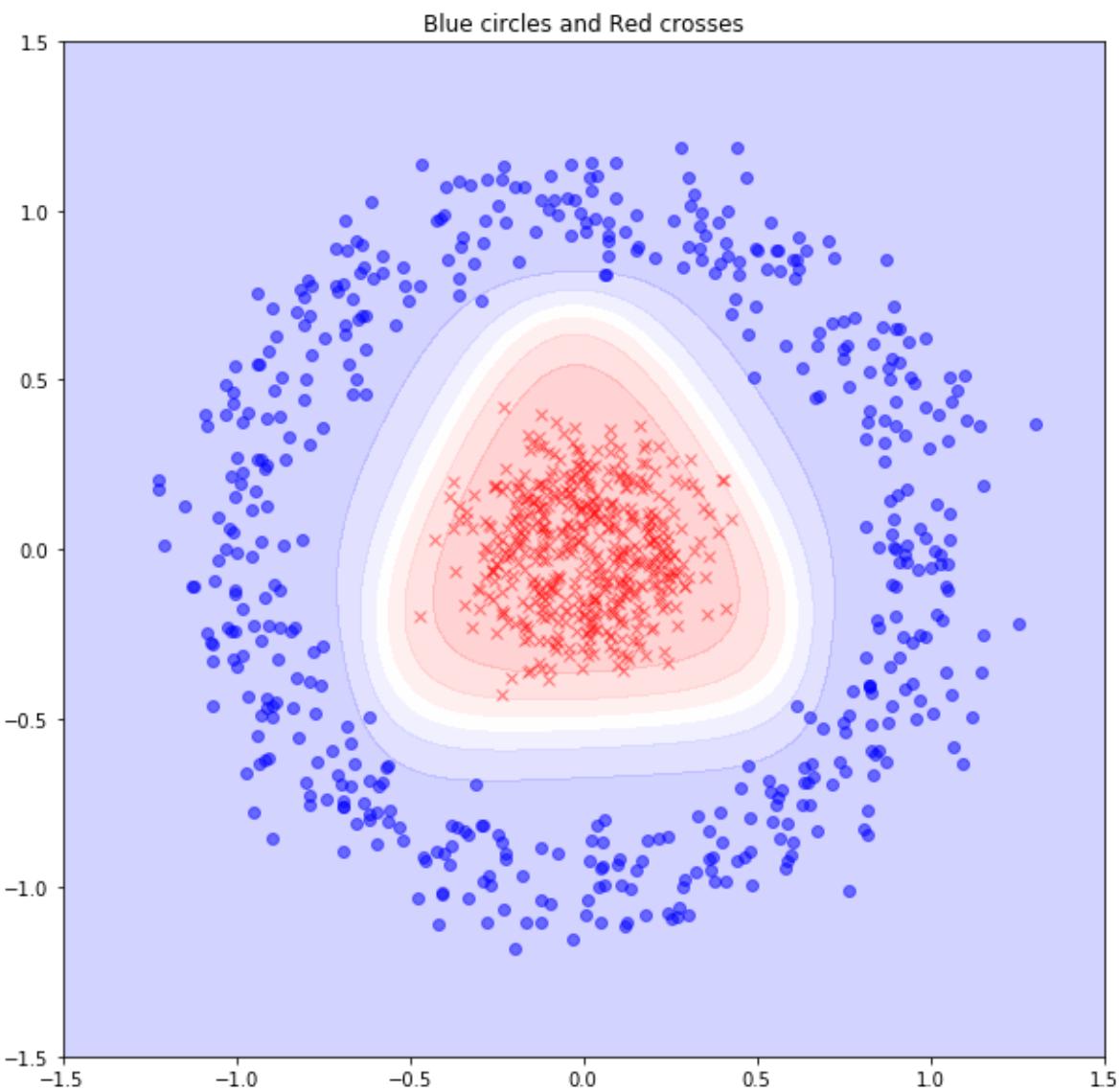
```
<matplotlib.collections.PathCollection at 0x7f17182e6358>
```



Nice! We see that a dense cloud of points with high probability is found in the central region of the plot, exactly where our red crosses are. We can draw the same data in a more appealing way using the `plt.contourf` function with appropriate colors and transparency:

```
plt.figure(figsize=(10, 10))
plt.contourf(aa, bb, cc, cmap='bwr', alpha=0.2)
plt.plot(X[y==0, 0], X[y==0, 1], 'ob', alpha=0.5)
plt.plot(X[y==1, 0], X[y==1, 1], 'xr', alpha=0.5)
plt.title("Blue circles and Red crosses")
```

```
Text(0.5,1,'Blue circles and Red crosses')
```



The last plot clearly shows the decision boundary of our model, i.e. the curve that delimits the area predicted to be red crosses VS the area predicted to be blue dots.

Our model learned to distinguish the two classes perfectly! This is really promising, although the current example was very simple.

Below are some exercises for you to practice with the commands and concepts we just introduced.

EXERCISE 1

Let's practice a little bit with `numpy` :

- generate an array of zeros with `shape=(10, 10)` , call it `a`
- set every other element of `a` to 1, both along columns and along rows, so that you obtain a nice checkerboard pattern of zeros and ones
- generate a second array to be the sequence from 5 included to 15 excluded , call it `b`
- multiply `a` times `b` in such a way that now the first row of `a` is an alternation of zeros and fives, the second row is an alternation of zeros and sixes and so on. call this new array `c` . in order to do this you will have to reshape `b` as a column array
- calculate the `mean` and the `standard deviation` of `c` along rows and along columns
- create a new array of `shape=(10, 5)` and fill it with the non-zero values of `c` , call it `d`
- add random gaussian noise to `d` , centered in zero and with standard deviation of 0.1, call thes new array `e`

EXERCISE 2

Practice plotting with `matplotlib`:

- use `plt.imshow()` to display the array `a` as an image, does it look like a checkerboard?
- display `c`, `d` and `e` using the same function, change the colormap to grayscale
- plot `e` using a line plot, assigning each row to a different data series. This should produce a plot with noisy horizontal lines. You will need to transpose the array to obtain this.
- add title, axes labels, legend and a couple of annotations

EXERCISE 3

Reuse your code:

- encapsulate the code that calculates the decision boundary in a nice function called `plot_decision_boundary` with the signature:

```
def plot_decision_boundary(model, X, y):  
    ....
```

EXERCISE 4

Practice retraining the model on different data:

- use the functions `make_blobs` and `make_moons` from scikit learn to generate new datasets with 2 classes
- plot the data to make sure you understand what has been generated
- re-train your model on each of these datasets
- display the decision boundary for each of these models

Chapter 2: Data Manipulation

This chapter is about data.

In order to do deep-learning effectively, we'll need to be able to work with data of all shapes and sizes. At the end of this section we will be able to explore data visually and do simple descriptive statistics using Python and Pandas.

MANY TYPES OF DATA

Data comes in many forms, formats, and sizes. For example, as a data scientist at a web company, a lot of our data will probably be accessible in the form of records in a database. We can think of these as very large spreadsheets, with rows and columns containing numbers.

On the other hand, if we are developing a method to **detect cancer from brain scans**, we will deal with images and video data, very often these files will be large in size (or number) and possibly in complicated formats.

If we are trying to **detect a signal for trading stocks** based on information in news articles, our data will often be millions of text documents.

If we are **translating spoken language to text**, our input data will be sound files, etc.

Traditionally Machine Learning has been fairly good at dealing with "tabular" data, while "unstructured" data such as text, sound and images, were each addressed with very complex, domain-specific techniques.

Deep Learning is particularly good at efficiently learning ways to represent such "unstructured" data, and this is one of the reasons for its enormous success. Neural net models can be used to solve a translation problem or an image classification problem, without worrying too much about the type of underlying data.

This is the first reason why Deep Learning is so popular: it can deal with many different types of data.

But before we can train models on our data, we need to gather the data and provide it to our networks in a consistent format. Let's take a look at a few different types of data and explore the tools we'll be using to process (and explore) them.

Tabular Data

The simplest data to feed to a Machine Learning model is so-called *tabular data*. It's called *tabular* because it can be represented in a table with rows and columns, very much like a spreadsheet. Let's use an example to define some common vocabulary that will be used throughout the book.

The diagram shows a box labeled "Features" with two arrows pointing down to a table row. The table has columns for Age, Gender, Annual Salary, Months in residence, Months in job, Current Debt, and Paid off credit. The fourth column is highlighted in red. The last row is also highlighted in red and contains the text "Record OR Data Point". The last column is labeled "Labels".

	Age	Gender	Annual Salary	Months in residence	Months in job	Current Debt	Paid off credit
Client 1	23	M	\$30,000	36	12	\$5,000	Yes
Client 2	30	F	\$45,000	12	12	\$1,000	Yes
Client 3	19	M	\$15,000	3	1	\$10,000	No
Client 4	Record OR Data Point				12	Labels	\$15,000 ?

A **row** in a table corresponds to a **datapoint**, and it's often referred to as a *record*. A **record** is a **list of attributes** (extracted from a data point), which are often numbers, categories, or free-form text. These attributes go by the name of *features*.

According to Bishop ¹, in Machine Learning a **feature** is an **individual measurable property** of a phenomenon being observed. In other words, **features are the properties we are using to characterize our data.**

Features can be directly measurable or they can be inferred from other features. Think, for example, of the number of times a user visited a website or the browser they used - both of these features can be directly counted. We could also create a *new feature* from existing data such as the average time *between* two user visits. The process of calculating new features is called *feature engineering*.

That said, not all the features can be as informative. Some may be completely irrelevant for what we are trying to do. For example, if we are trying to predict how likely a user is to buy our product, chances are that his/her **first name will have no predictive power**. On the other hand, **previous purchases may carry a lot of information** in terms of propensity to buy.

While traditionally a lot of emphasis has been placed on *feature engineering* (extracting or inventing "good" features) and *feature selection* (keeping only the "good" features), Deep Learning solves this problem by automatically figuring out the important features and building higher order combinations of simple features deeper in the network.

This is another reason why Deep Learning is so popular: it automates the complicated process of feature engineering.

1: Bishop, Christopher (2006). *Pattern recognition and Machine Learning*. Berlin: Springer. ISBN 0-387-31073-8.

DATA EXPLORATION WITH PANDAS

When building a predictive model, it's often helpful to get some quick facts about our dataset. We may spot some very evident problems with the data that we may want to address. This first phase is called *data exploration* and consists of a series of questions that we want to ask:

- How big is our dataset?
- How many features do we have?
- Is any record corrupted or missing information for one or more features?
- Are the features numbers or categories?
- How is each feature distributed? Are they correlated?

We want to ask these questions early in order to decide how to proceed further without wasting time. For example, if we have too few data points we may not have enough examples to train a Machine Learning model. Our first step in that case will be to go out and gather more data. If we have missing data we need to decide what to do about it. Do we delete the records missing data or do we *impute* (create) the missing data? And if we impute the data, how do we decide how to impute it? If we have many features but only few of them are not constant, we'd better eliminate the constant features first, because they will clearly have no predictive power, and so on...

Python comes with a library that allows to address all these questions very easily, it's called *Pandas*.

Let's load it in our notebook.

```
import pandas as pd
```

Pandas is an open source library that provides high-performance, easy-to-use data structures and data analysis tools. It can load data from a multitude of sources including CSV, JSON, Excel, HTML, PDF and many others ([here](#) you may find all the types of file that can be loaded, together with a short description). Let's start by loading a `csv` file.

TIP: A comma-separated values file ([CSV](#)) stores tabular data (numbers and text) in plain text. Each line of the file is a data record, and each record consists of one or more fields, separated by commas.

```
df = pd.read_csv('..../data/titanic-train.csv')
```

This is a famous dataset containing information about passengers of the Titanic, such as their name, age, and if they survived.

`pd.read_csv` will read the CSV file and create a *Pandas DataFrame object* from it. A **DataFrame** is a labeled, 2D data-structure, much like a spreadsheet.

Now that we have imported the Titanic data into a Pandas DataFrame object, we can inspect it. Let's start by peeking into the first few records to get a feel for how DataFrames work.

`df.head()` displays the first 5 lines of the DataFrame. We can see it as a table, with column names inferred from the CSV file and an index, indicating the row it came from:

```
df.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 :
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC :
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STC 310:
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	1138

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	3734

`df.info()` summarizes the content of the DataFrame, letting us know the index range, the number and names of columns with their data type.

We also learn about missing entries. For example, notice that the `Age` column has a few `null` entries.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass          891 non-null int64
Name            891 non-null object
Sex             891 non-null object
Age             714 non-null float64
...
Age             714 non-null float64
```

`df.describe()` summarizes the numerical columns with some basic stats: count, min, max, mean, standard deviation etc.

```
df.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057

	PassengerId	Survived	Pclass	Age	SibSp	Parch
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000

This is very useful to compare the scale of different features and decide if we need to rescale some of them.

Indexing

We can access individual elements of a DataFrame. Let's see a few ways.

We can get the fourth row of the DataFrame (numerical index 3) using `df.iloc[3]`

```
df.iloc[3]
```

```
PassengerId          4
Survived             1
Pclass               1
Name     Futrelle, Mrs. Jacques Heath (Lily May Peel)
Sex                  female
Age                 35
SibSp               1
Parch               0
Ticket              113803
...
Name: 3, dtype: object
```

We can fetch elements corresponding to indices 0-4 and column 'Ticket':

```
df.loc[0:4, 'Ticket']
```

```
0      A/5 21171
1      PC 17599
2  STON/O2. 3101282
3      113803
4      373450
Name: Ticket, dtype: object
...
Name: Ticket, dtype: object
...
Name: Ticket, dtype: object
```

We can obtain the same result by selecting the first 5 elements of the column 'Ticket', with `.head()` command:

```
df['Ticket'].head()
```

```
0      A/5 21171
1      PC 17599
2  STON/O2. 3101282
3      113803
4      373450
Name: Ticket, dtype: object
...
Name: Ticket, dtype: object
...
Name: Ticket, dtype: object
```

To select multiple columns, we just pass the list of columns:

```
df[['Embarked', 'Ticket']].head()
```

	Embarked	Ticket
0	S	A/5 21171
1	C	PC 17599
2	S	STON/O2. 3101282

	Embarked	Ticket
3	S	113803
4	S	373450

Selections

Pandas is smart about indices and allows us to write expressions. For example, we can get the list of passengers with Age over 70 :

```
df[df['Age'] > 70]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
96	97	0	1	Goldschmidt, Mr. George B	male	71.0	0	0	F 1
116	117	0	3	Connors, Mr. Patrick	male	70.5	0	0	3
493	494	0	1	Artagaveytia, Mr. Ramon	male	71.0	0	0	F 1
630	631	1	1	Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	2
851	852	0	3	Svensson, Mr. Johan	male	74.0	0	0	3

To understand what this does, let's break it down.

```
df['Age'] > 70
```

```

0    False
1    False
2    False
3    False
4    False
5    False
6    False
7    False
8    False
...
Name: Age, Length: 891, dtype: bool

```

This returns a boolean series of values that are `True` when the `Age` is greater than `70` (and `False` otherwise).

Passing this series to the `[]` selects only the rows for which the boolean series is `True`. We can obtain the same result using the `query` operator.

```
df.query("Age > 70")
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
96	97	0	1	Goldschmidt, Mr. George B	male	71.0	0	0	F 1
116	117	0	3	Connors, Mr. Patrick	male	70.5	0	0	3
493	494	0	1	Artagaveytia, Mr. Ramon	male	71.0	0	0	F 1
630	631	1	1	Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	2

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
851	852	0	3	Svensson, Mr. Johan	male	74.0	0	0	3

We can use the `&` and `|` Python operators (which normally do bitwise and bitwise or, respectively) to combine conditions. For example, the next statement returns the records of passengers 11 years old and with 5 siblings/spouses.

```
df[(df['Age'] == 11) & (df['SibSp'] == 5)]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticke
59	60	0	3	Goodwin, Master. William Frederick	male	11.0	5	2	CA 2144

If we use an `or` operator, we'll have passengers that are 11 years old or passengers with 5 siblings/spouses.

```
df[(df.Age == 11) | (df.SibSp == 5)]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
59	60	0	3	Goodwin, Master. William Frederick	male	11.0	5	2	C 2
71	72	0	3	Goodwin, Miss. Lillian Amy	female	16.0	5	2	C 2

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch
386	387	0	3	Goodwin, Master. Sidney Leonard	male	1.0	5	2
480	481	0	3	Goodwin, Master. Harold Victor	male	9.0	5	2
542	543	0	3	Andersson, Miss. Sigrid Elisabeth	female	11.0	4	2
683	684	0	3	Goodwin, Mr. Charles Edward	male	14.0	5	2
731	732	0	3	Hassan, Mr. Houssein G N	male	11.0	0	0
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2

Again, we can use the `query` method to achieve the same result.

```
df.query('(Age == 11) | (SibSp == 5)')
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch
--	-------------	----------	--------	------	-----	-----	-------	-------

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
59	60	0	3	Goodwin, Master. William Frederick	male	11.0	5	2	C 2
71	72	0	3	Goodwin, Miss. Lillian Amy	female	16.0	5	2	C 2
386	387	0	3	Goodwin, Master. Sidney Leonard	male	1.0	5	2	C 2
480	481	0	3	Goodwin, Master. Harold Victor	male	9.0	5	2	C 2
542	543	0	3	Andersson, Miss. Sigrid Elisabeth	female	11.0	4	2	E
683	684	0	3	Goodwin, Mr. Charles Edward	male	14.0	5	2	C 2
731	732	0	3	Hassan, Mr. Houssein G N	male	11.0	0	0	2

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2	1

Unique Values

The `unique` method returns the unique entries. For example, we can use it to know the possible ports of embarkment and only select the unique values.

```
df['Embarked'].unique()
```

```
array(['S', 'C', 'Q', nan], dtype=object)
```

Sorting

We can sort a DataFrame by any group of columns. For example, let's sort people by `Age`, starting from the oldest using the `ascending` flag. By default, `ascending` is set to `True`, which sorts by the youngest first. To reverse the sort order, we set this value to `False`.

```
df.sort_values('Age', ascending = False).head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
630	631	1	1	Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	2
851	852	0	3	Svensson, Mr. Johan	male	74.0	0	0	3

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	
493	494	0	1	Artagaveytia, Mr. Ramon	male	71.0	0	0	F 1
96	97	0	1	Goldschmidt, Mr. George B	male	71.0	0	0	F 1
116	117	0	3	Connors, Mr. Patrick	male	70.5	0	0	3

Aggregations

Pandas also allows to perform aggregations and group-by operations like we can do in [SQL](#) and can reshuffle data into pivot-tables like a spreadsheet application. This makes it very powerful for data exploration and we strongly recommend a thorough look at its [documentation](#) if you are new to Pandas. Here we will review only a few useful commands.

`value_counts()` counts how many instances of each value are there in a series, sorting them in descending order. We can use it to know how many people survived and how many died.

```
df['Survived'].value_counts()
```

```
0    549
1    342
Name: Survived, dtype: int64
...
Name: Survived, dtype: int64
...
Name: Survived, dtype: int64
```

or how many people were travelling in each class.

```
df['Pclass'].value_counts()
```

```
3    491
1    216
2    184
Name: Pclass, dtype: int64
...
Name: Pclass, dtype: int64
...
Name: Pclass, dtype: int64
```

Like in a database, we can group data by column name and then aggregate them with some function. For example, let's count dead and alive passengers by class:

```
df.groupby(['Pclass', 'Survived'])['PassengerId'].count()
```

```
Pclass  Survived
1      0          80
        1         136
2      0          97
        1          87
3      0         372
        1         119
Name: PassengerId, dtype: int64
...
...
Name: PassengerId, dtype: int64
```

This is a very powerful tool, we can immediately see that almost 2/3 of passengers in first class survived, compared to only about 1/3 of passengers in 3rd class!

We can look at individual columns `min`, `max`, `mean` and `median`, in order to get some more information about our numerical features. For example, the next line shows that the youngest passenger was less than six-months old:

```
df['Age'].min()
```

0.42

while the oldest was eighty years old:

```
df['Age'].max()
```

80.0

The average age of the passengers was almost 30 years old:

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

29.69911764705882

While the median age was a bit younger, 28 years old:

```
df['Age'].median()
```

28.0

We can see if the mean age of survivors was different from the mean age of victims.

```
mean_age_by_survived = df.groupby('Survived')['Age'].mean()
mean_age_by_survived
```

```
Survived
0    30.626179
1    28.343690
Name: Age, dtype: float64
...
Name: Age, dtype: float64
...
Name: Age, dtype: float64
```

Although the mean age of survivors seems a bit lower, the difference between the 2 classes is not statistically significant as we can see by looking at the standard deviation.

```
std_age_by_survived = df.groupby('Survived')['Age'].std()  
std_age_by_survived
```

```
Survived  
0    14.172110  
1    14.950952  
Name: Age, dtype: float64  
...  
Name: Age, dtype: float64  
...  
Name: Age, dtype: float64
```

Merge

Pandas can perform join operations like we can do in SQL. This operation is called `merge`. For example, let's combine the 2 previous tables:

```
df1 = mean_age_by_survived.round(0).reset_index()  
df1
```

	Survived	Age
0	0	31.0

1	1	28.0
---	---	------

```
df2 = std_age_by_survived.round(0).reset_index()  
df2
```

	Survived	Age
0	0	14.0

	Survived	Age
1	1	15.0
0	0	31.0
1	1	28.0

```
df3 = pd.merge(df1, df2, on='Survived')
df3
```

	Survived	Age_x	Age_y
0	0	31.0	14.0
1	1	28.0	15.0

```
df3.columns = ['Survived', 'Average Age', 'Age Standard Deviation']
df3
```

	Survived	Average Age	Age Standard Deviation
0	0	31.0	14.0
1	1	28.0	15.0

`merge` is incredibly powerful. We recommend reading more into its functionality in Pandas documentation

Pivot Tables

Pandas has the ability to aggregate data into a pivot table, just like Microsoft Excel.

TIP: A pivot table is a table that summarizes data in another table, and is made by applying an operation such as sorting, averaging, or summing to data in the first table. A trivial example is a column of numbers as the first table, and the column average as a pivot table with only one row and column.

For example, we can create a table which holds the count of the number of people who survived (or not) per class:

```
df.pivot_table(index='Pclass',
               columns='Survived',
               values='PassengerId',
               aggfunc='count')
```

Survived	0	1
Pclass		
1	80	136
2	97	87
3	372	119

Correlations

Finally, Pandas can also calculate correlations between features, making it easier to spot redundant information or uninformative columns.

For example, let's check the correlation of a few columns with a `True` value of `Survived`. If it's true that *women and children are saved first*, we expect to see some correlation with `Age` and `Sex`, while we expect no correlation with `PassengerId`.

Since the `Sex` column is a string, we first need to create an auxiliary (extra) `IsFemale` boolean column that is set to `True` if the `Sex` is set to the string 'female'.

```
df['IsFemale'] = df['Sex'] == 'female'
```

```
correlated_with_survived = df.corr()['Survived'].sort_values()
correlated_with_survived
```

Pclass	-0.338481
Age	-0.077221
SibSp	-0.035322
PassengerId	-0.005007
Parch	0.081629

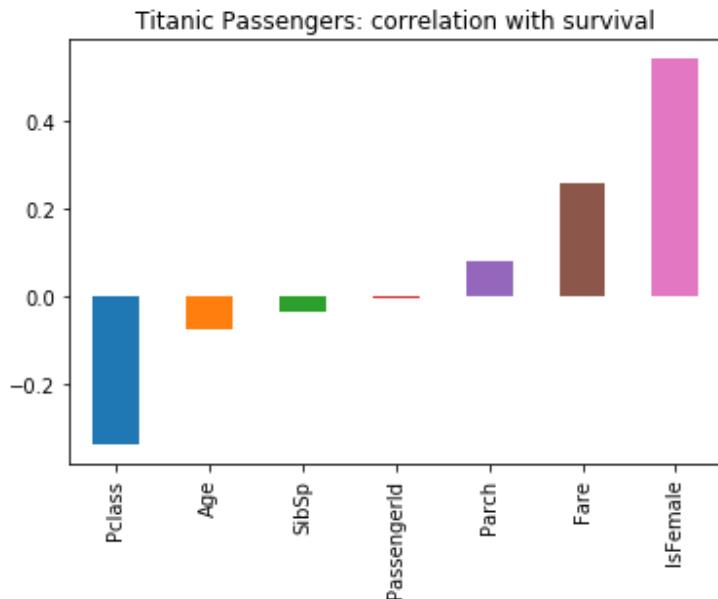
```
Fare           0.257307
IsFemale       0.543351
Survived      1.000000
Name: Survived, dtype: float64
...
Name: Survived, dtype: float64
```

Before looking at what these values mean, let's peek ahead a little and look into Pandas plotting functionality. We can use Pandas plotting functionality to display the last result visually (remember to use the `%matplotlib inline` magic to show the figures).

```
%matplotlib inline
```

```
title = 'Titanic Passengers: correlation with survival'
correlated_with_survived.iloc[:-1].plot(kind='bar', title=title)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0da47de240>
```



Let's interpret the graph above. The largest correlation with survival is being a woman. We also see that people who paid a higher fare (probably corresponding to a higher class) had a higher chance of surviving.

The attribute `Pclass` is negatively correlated, meaning the higher the class number the lower the chance of survival, which makes sense (first class passenger more likely to survive than third class).

`Age` is also negatively correlated, though mildly, meaning the younger you are the more likely you are to survive. Finally, as expected `PassengerId` has no correlation with survival.

We've barely scratched the surface of what Pandas can do in terms of data manipulation and data exploration. Do refer to the mentioned documentation for a better understanding of its capabilities.

VISUAL DATA EXPLORATION

After an initial look at the properties of our tabular dataset, it is often very useful to dig a little deeper using visualizations. Looking at a graph we may spot a trend, a particular repeating pattern, or a correlation. In fact, our visual cortex is an extremely good pattern recognizer, so it only makes sense to take advantage of it when possible.

We can represent data visually in several ways, depending on the type of data and on what we are interested in seeing.

Let's create some artificial data and visualize it in different ways.

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

We will create 3 data series:

- A stationary noisy sequence, centered around zero (`data1`).
- A sequence with larger noise, following a linearly increasing trend (`data2`).
- A sequence with where noise increases over time (`data3`).
- A sequence with somewhat intermediate noise, following a sinusoidal oscillatory pattern (`data4`).

```
N = 1000
data1 = np.random.normal(0, 0.1, N)
data2 = np.random.normal(1, 0.4, N) + np.linspace(0, 1, N)
data3 = 2 + np.random.random(N) * np.linspace(1, 5, N)
data4 = np.random.normal(3, 0.2, N) + 0.3 * np.sin(np.linspace(0, 20, N))
```

Now, let's create a `DataFrame` object composing all of our newly created data sequences:

```
data = np.vstack([data1, data2, data3, data4]).transpose()
df = pd.DataFrame(data, columns=['data1', 'data2', 'data3', 'data4'])
df.head()
```

	data1	data2	data3	data4

	data1	data2	data3	data4
0	0.091047	1.046176	2.624891	3.010013
1	0.165112	0.791210	2.304627	2.966248
2	0.052250	0.739787	2.513639	2.881766
3	0.079740	1.123321	2.257848	3.467654
4	0.015850	0.976164	2.855266	3.221133

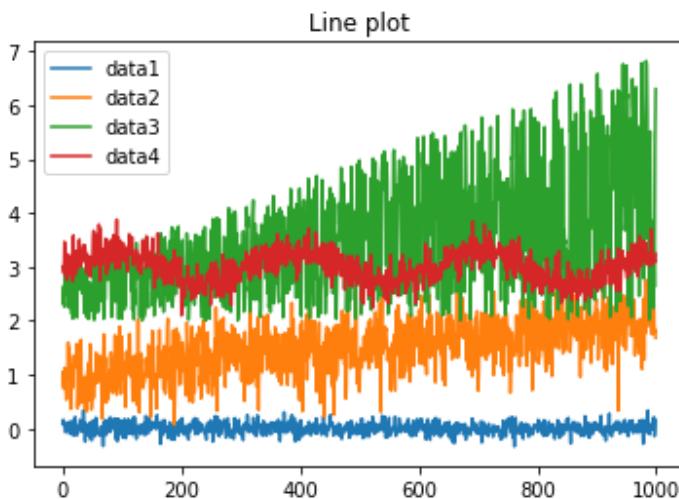
Even when we've been given a description of these four data sets, it's really hard to understand what's going on by simply looking at the table of numbers. Instead, let's look at this data visually.

Line Plot

Pandas `plot` function defaults to a line plot. This is a good choice if our data comes from an ordered series of consecutive events (for example, the outside temperature in a city over the course of a year).

A line plot represents the values of data in sequential order, and makes it easy to spot trends like growth over time or seasonal patterns.

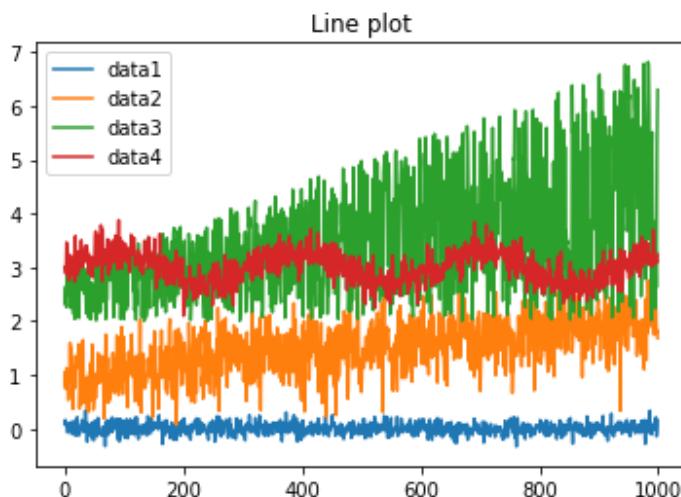
```
_ = df.plot(title='Line plot')
```



Above, we're using the `plot` method on the DataFrame. The same plot can be obtained by using `matplotlib.pyplot` (and passing in the DataFrame `df` as an argument) like this:

```
plt.plot(df)
plt.title('Line plot')
plt.legend(['data1', 'data2', 'data3', 'data4'])
```

```
<matplotlib.legend.Legend at 0x7f0da0b39c88>
```

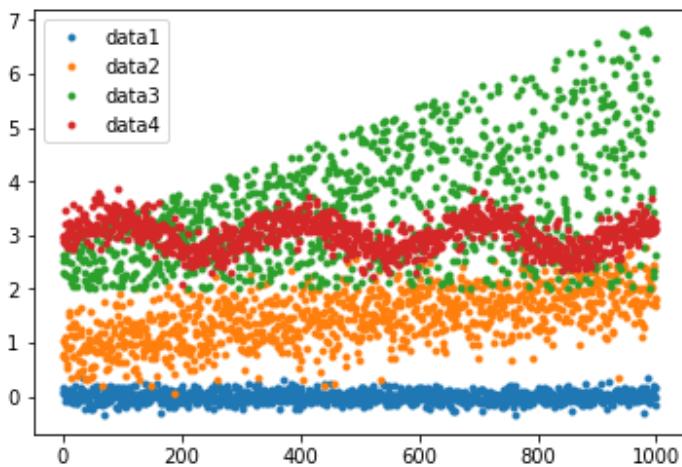


Scatter plot

If data is not ordered, and we are looking for correlations between variables, a `scatter` plot is a better choice. We can simply change the style of the line plot if we want to plot data in order:

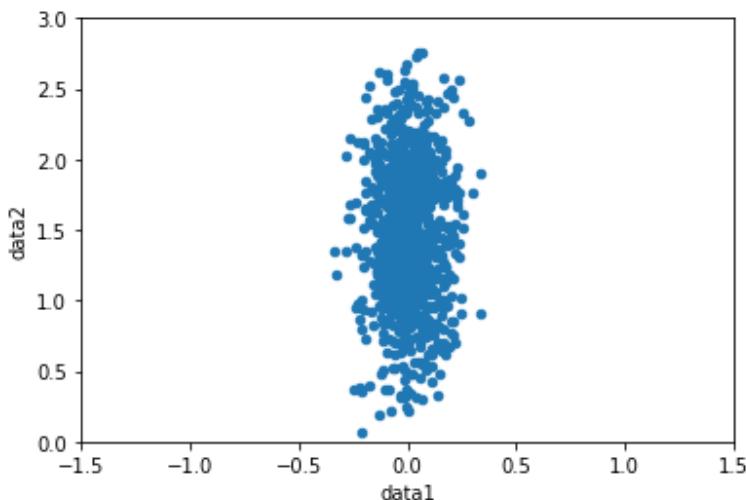
```
df.plot(style='.')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0da0b01668>
```



or we can use the `scatter` plot kind, if we want to visualize one column against another:

```
_ = df.plot(kind='scatter', x='data1', y='data2',
            xlim=(-1.5, 1.5), ylim=(0, 3))
```



In the above plot, we see that there is no correlation between `data1` and `data2` (which may be obvious because `data1` is a flat random noise).

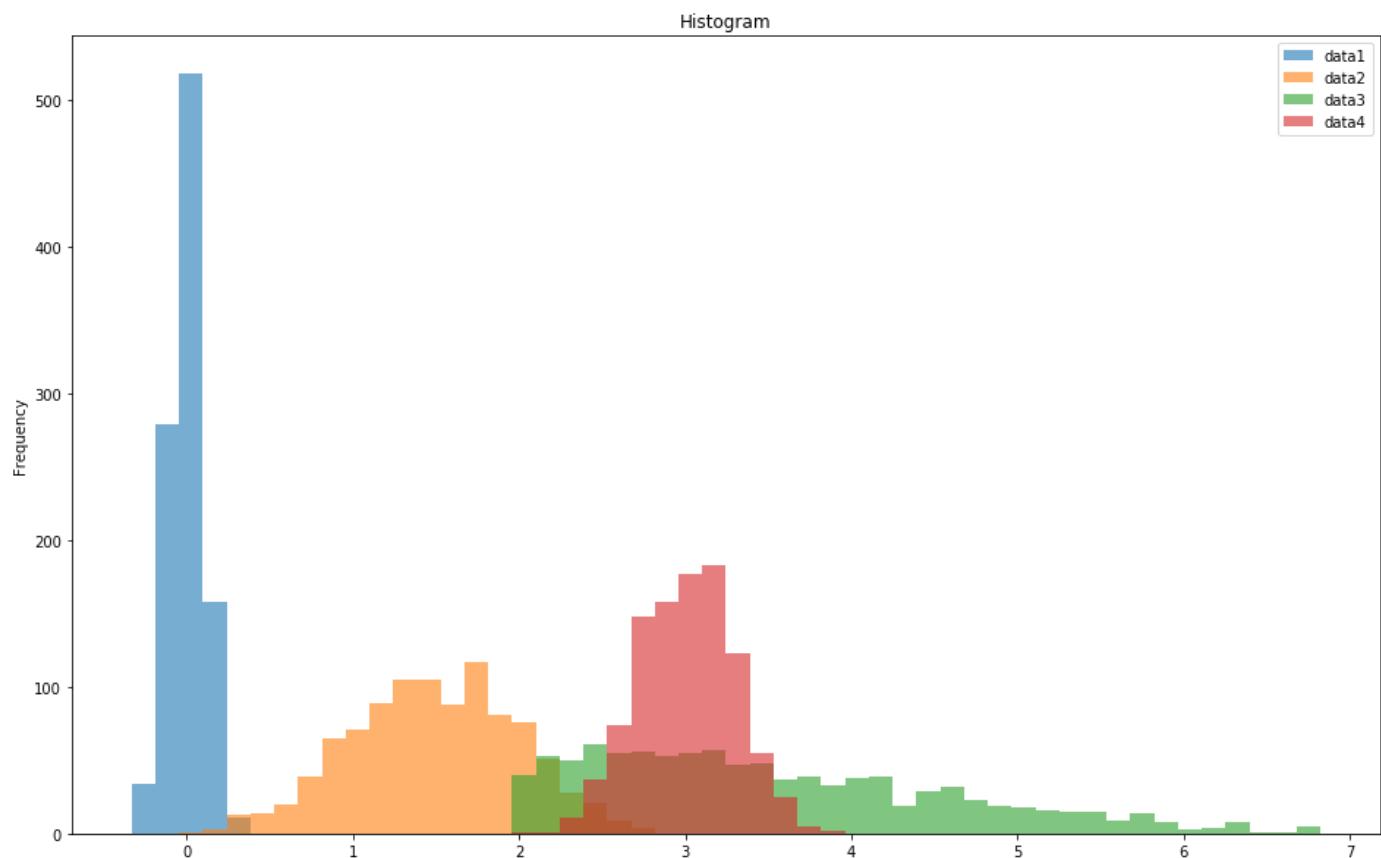
Histograms

Sometimes we are interested in knowing the **frequency of occurrence** of data, and not their order. In this case we divide the range of data into buckets and ask *how many points fall into each bucket*. This is called a **histogram**, and it represents the statistical distribution of our data.

This could look like a bell curve, or an exponential decay, or have a weird shape. By plotting the histogram of a feature we might spot the presence of distinct sub-populations in our data and decide to deal with each one separately.

```
df.plot(kind='hist',
        bins=50,
        title='Histogram',
        alpha=0.6,
        figsize=(16, 10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0da0260278>
```



Note that we lost all the temporal information contained in our data, for example the oscillations in `data4` are not visible any longer, all we see is a quite large bell-like distribution, where the sinusoidal oscillations have been summed up in the histogram.

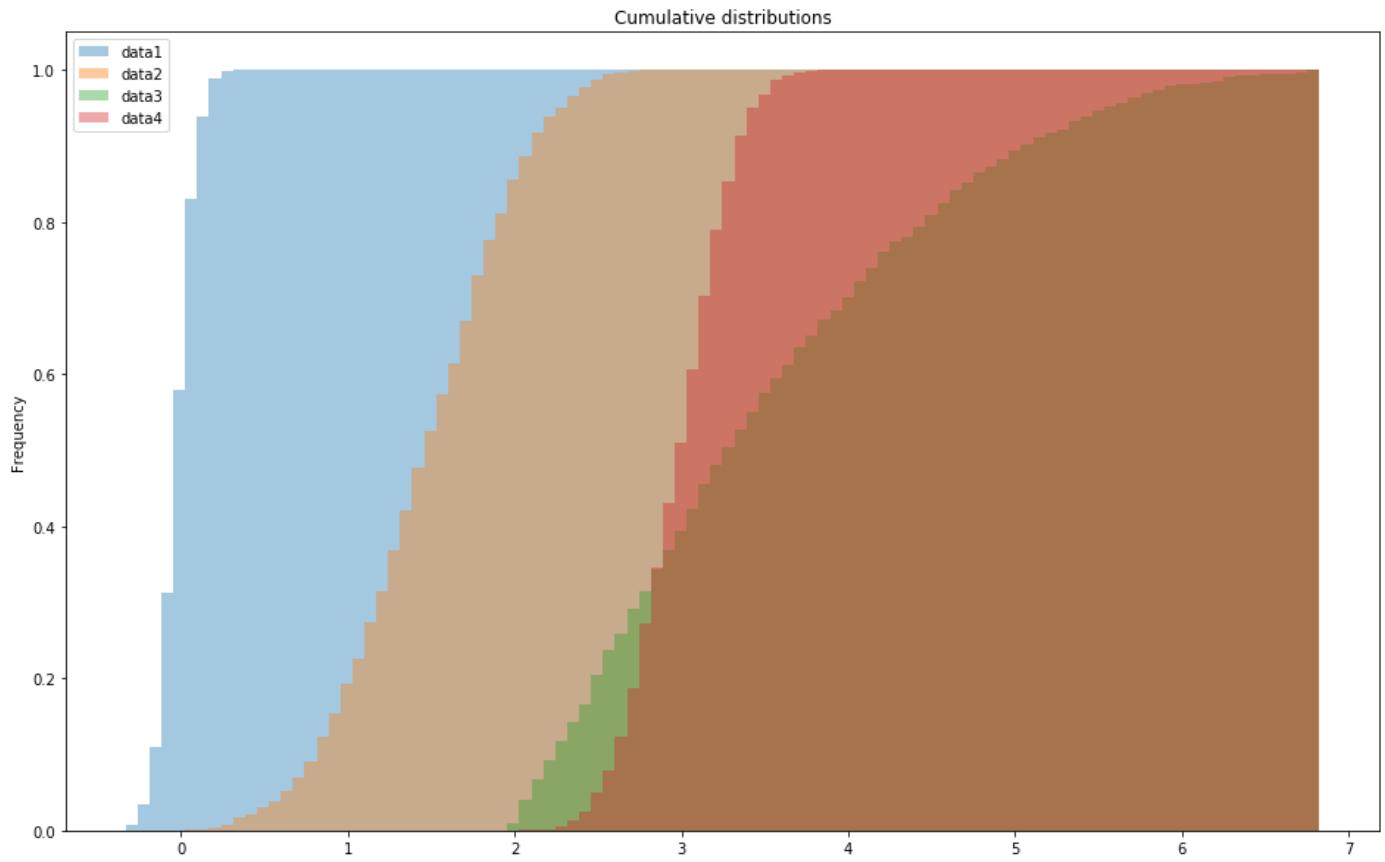
Cumulative Distribution

A close relative of a histogram is the cumulative distribution. This is useful to answer questions like: what fraction of our sample falls below a certain value?

```
df.plot(kind='hist',
        bins=100,
        title='Cumulative distributions',
        normed=True,
        cumulative=True,
        alpha=0.4,
        figsize=(16, 10))
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/matplotlib/axes/_axes.p
warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0da01bfcf8>
```



Try to answer these questions:

1. how much of `data1` falls below 2?
2. how much of `data2` falls below 1.5?

Answers:

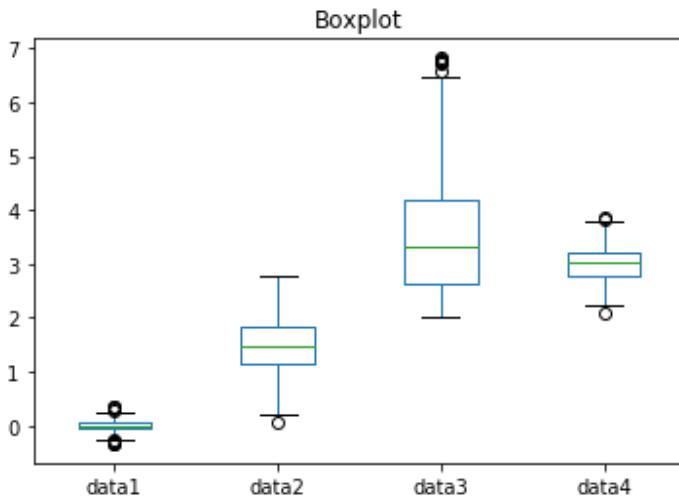
1. 100%. If you draw a vertical line that passes through 2 you will see that it crosses the cumulative distribution for `data1` at the high value of 1, which corresponds to 100%.
2. approximately 50%. This can be seen by tracing a vertical line at 1.5 and checking at what height it crosses the `data2` distribution.

Box plot

A [box plot](#) is a useful tool to compare several distributions, it is often used in biology and medicine to compare the results of an experiment with a control group. For example, in the simplest form of a clinical trial for a new drug, there will be 2 boxes, one for the population that took the drug and the other for the population that took the placebo.

```
df.plot(kind='box',
        title='Boxplot')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0d9af02e10>
```



What does this plot mean?

In loose terms, it's as if we were looking at the histogram plot from above. Each box represents the key facts about the distribution of that particular data series. Let's first get an intuition about the information it shows. Later we will give a more formal definition.

Let's start with the green horizontal line that cuts each box. It represents the position of the peak of the histogram. We can check the peak for `data1` that the line is at 0, exactly like the very sharp peak of `data1` in the histogram figure, and for `data4` the green line is roughly at 3, exactly like the peak of the red histogram in the previous picture.

The box represents the bulk of the data, i.e. it gives us an idea of how fat and centered our distribution is around the peak. We can see that the box in `data3` is not centered around the green line, reflecting the fact that the histogram in green is skewed. The whiskers give us an idea of the extension of the tails of the distribution. Again, notice how the upper whisker of `data3` extends to high values.

TIP: For the more statistically inclined readers, here are the formal definitions of the above concepts:

- The green line is the *median* of our data, i.e. the value lying at the midpoint of the distribution.
- The box around it denotes the *confidence interval* (calculated using a *gaussian approximation*). Notice how these reproduce more closely the actual size of the noise fluctuations for `data2` and `data4`.

- The whiskers above and below denote the range of data not considered **outliers**. By default they are set to be at $[Q1 - 1.5 \times IQR, Q3 + 1.5 \times IQR]$, where $Q1$ is the first quartile, $Q3$ the third quartile and IQR the interquartile range. Notice that these give us a clear indication that `data3` is not symmetric around its median.
- The dots represent data that are considered outliers.

TIP: In the previous *TIP*, we just introduced the concept of *outliers*. Outliers are data that are distant from other observations. Outliers may be due for example to variability in the measurement or they may indicate experimental errors. This is a fundamental concept in Machine Learning, and we'll have the chance to discuss it later.

Subplots

We can also combine these plots in a single figure using the `subplots` command:

```
fig, ax = plt.subplots(2, 2, figsize=(16,12))

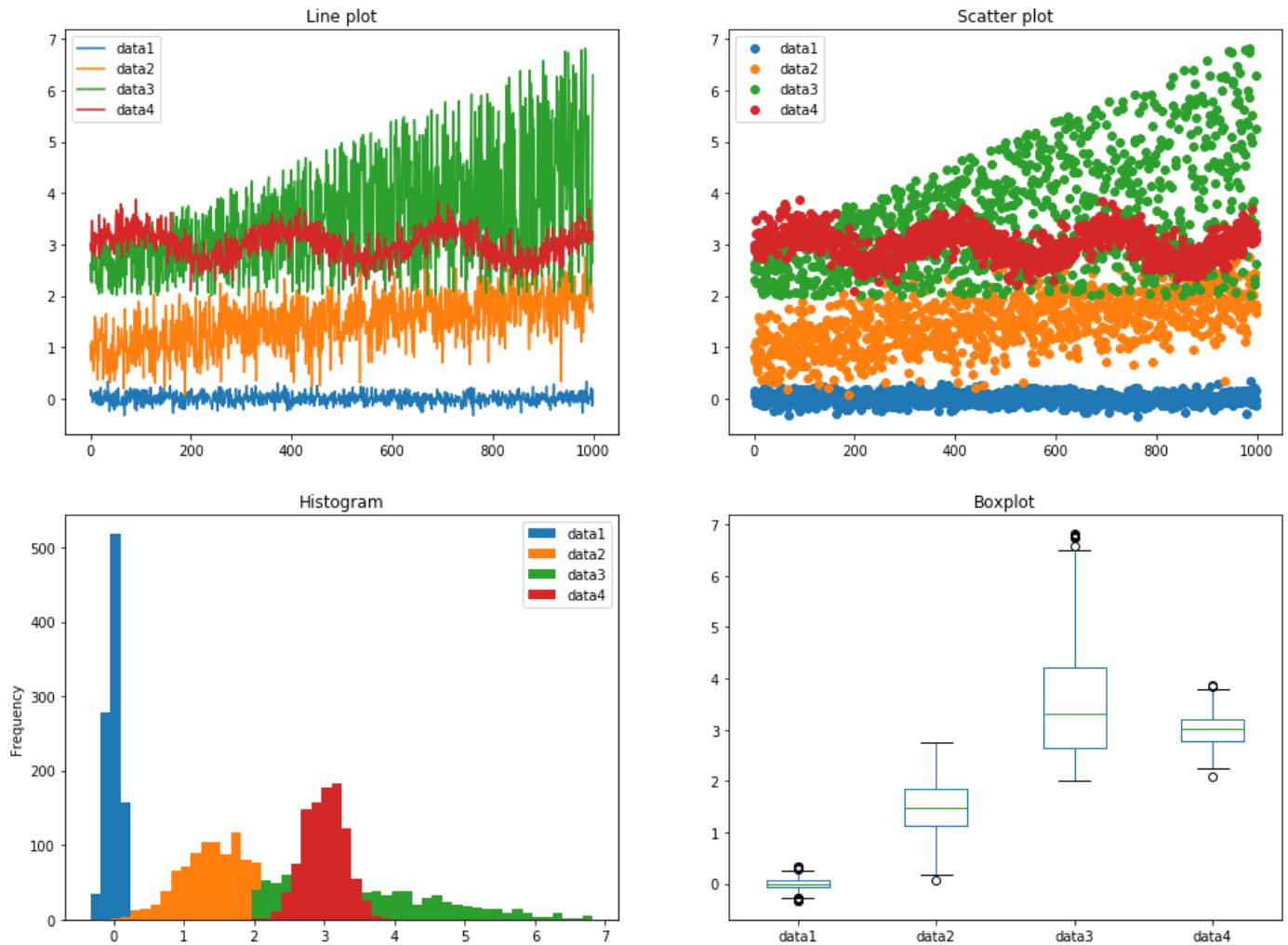
df.plot(ax=ax[0][0],
        title='Line plot')

df.plot(ax=ax[0][1],
        style='o',
        title='Scatter plot')

df.plot(ax=ax[1][0],
        kind='hist',
        bins=50,
        title='Histogram')

df.plot(ax=ax[1][1],
        kind='box',
        title='Boxplot')
```

`<matplotlib.axes._subplots.AxesSubplot at 0x7f0d9a7f6d30>`



Pie charts

Pie charts are useful to visualize fractions of a total, for example we could ask how much of `data1` is greater than 0.1:

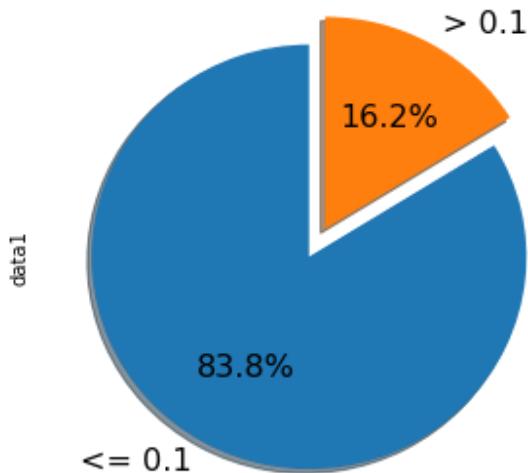
```
gt01 = df['data1'] > 0.1
piecounts = gt01.value_counts()
piecounts
```

```
False      838
True      162
Name: data1, dtype: int64
...
Name: data1, dtype: int64
```

```
...
Name: data1, dtype: int64
```

```
piecounts.plot(kind='pie',
                figsize=(5, 5),
                explode=[0, 0.15],
                labels=['<= 0.1', '> 0.1'],
                autopct='%1.1f%%',
                shadow=True,
                startangle=90,
                fontsize=16)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0d9a6dbdd8>
```



Hexbin plot

Hexbin plots are useful to look at 2-D distributions. Let's generate some new data for this plot.

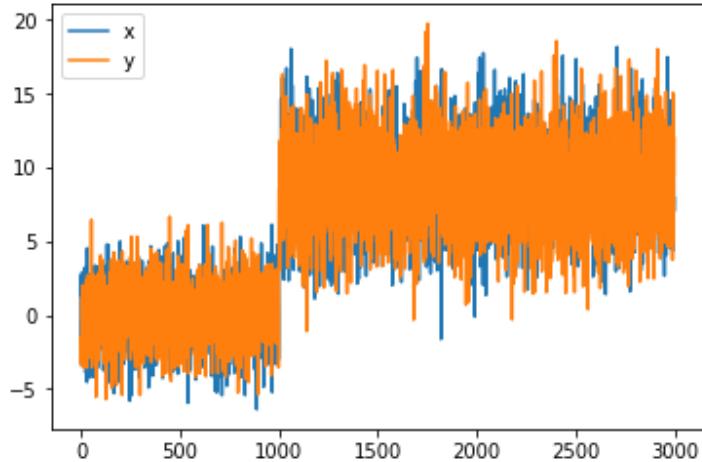
```
data = np.vstack([np.random.normal((0, 0), 2, size=(1000, 2)),
                 np.random.normal((9, 9), 3, size=(2000, 2))])
df = pd.DataFrame(data, columns=['x', 'y'])
```

```
df.head()
```

	x	y
0	-1.925066	-1.469142
1	2.650807	-3.340211
2	1.278985	-1.706053
3	2.870540	1.288486
4	-3.443489	1.330949

```
df.plot()
```

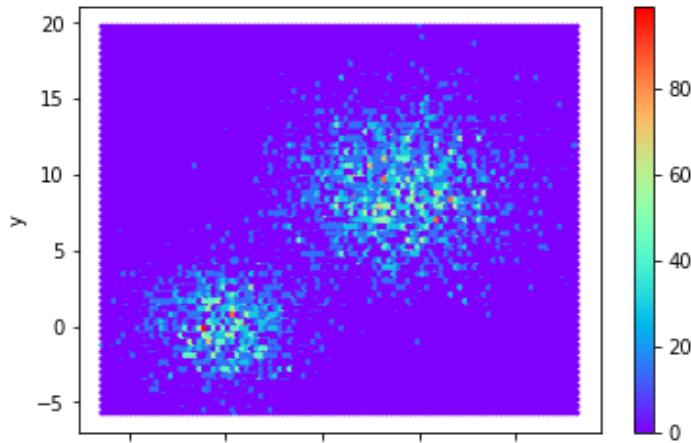
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0d9a0a9588>
```



This new data is a stack of two 2-D random sequences, the first one centered in (0, 0) and the second one centered in (9, 9). Let's see how the `hexbin` plot visualizes them.

```
df.plot(kind='hexbin', x='x', y='y', bins=100, cmap='rainbow')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0d9a5192e8>
```



The Hexbin plot is the 2-D extension of a histogram. It is created by creating a set of tiles that cover the 2-D plane, and then counting how many points end up in each tile. The color is proportional to the count. Since we created this dataset with points sampled from 2 gaussian distributions, we expect to see tiles containing more points near the centers of these two gaussians, which is what we observe above.

We encourage you to have a look at the [this gallery](#) to get some inspiration on visualizing your data. Remember that the choice of visualization is strongly tied to the kind of data and the kind of question we are asking.

UNSTRUCTURED DATA

Most often than not, data doesn't come as a nice, well-formatted table. As we mentioned earlier, we could be dealing with images, sound, text, movies, protein molecular structures, video games and many other types of data.

The beauty of Deep Learning is that it can handle most of this data and learn optimal ways to represent it for the task at hand.

Images

Let's take images for example. We'll use the PIL imaging library (which is referred to as *Pillow* for newer versions).

```
from PIL import Image
```

```
img = Image.open('../data/iss.jpg')
img
```



S116E05968

We can convert the image to a 3-D array using `numpy`. After all, an image can be seen as a table of pixels. For each pixel, the values of red, green and blue are specified. So, our image is really a 3 dimensional table, where rows and columns correspond to the pixel index and the depth correspond to the color channel.

```
imgarray = np.asarray(img)
```

```
imgarray.shape
```

```
(435, 640, 3)
```

The shape of the above array indicating (width, height, channels). While it's quite easy to think of features when dealing with tabular data, it's trickier when we deal with images. We could imagine unrolling this image onto a long list of numbers, walking along each of the 3 dimensions, and we did so, our dataset of images would again be a tabular dataset, with each row corresponding to a particular image and each column corresponding to a specific pixel and color channel.

```
imgarray.ravel().shape
```

```
(835200, )
```

However, not only this procedure created 835200 features for our image, but also by doing so we lost most of the useful information in the image. In other words, a single pixel in an image carries very little information, while most of the information is contained in **changes and correlations between nearby pixels**. Neural Networks can learn features from that through a technique called *convolution*, which we will learn about later in this course.

Sound

Now take sound. Digitally recorded sound is a long series of ordered numbers representing the sound wave. Let's load an example file.

```
from scipy.io import wavfile
```

```
rate, snd = wavfile.read(filename='./data/sms.wav')
```

We can play the audio file in the notebook:

```
from IPython.display import Audio
```

```
Audio(data=snd, rate=rate)
```

```
<audio controls="controls" >
    <source src="data:audio/wav;base64,UklGRnhdAwBXQVFZm10IBAAAAABAAEARKwAAI"
    Your browser does not support the audio element.
</audio>
```

This file is sampled at 44.1 kHz, which means 44100 times per second. So, our 3 second file contains over 100k samples:

```
len(snd)
```

```
110250
```

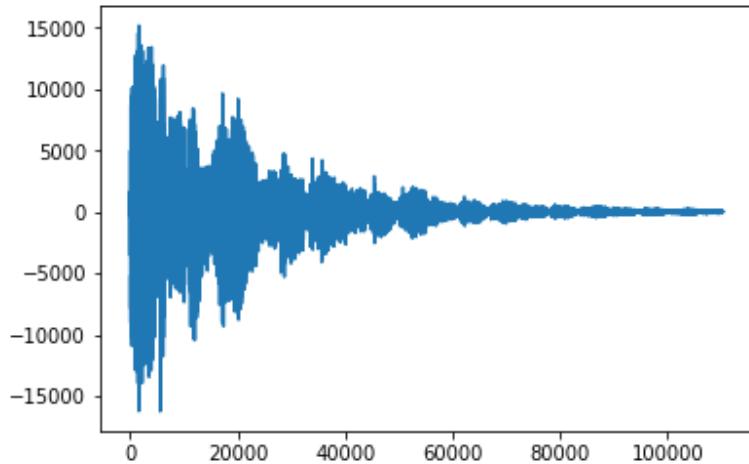
```
snd
```

```
array([70, 14, 27, ..., 58, 68, 59], dtype=int16)
```

We can use matplotlib to plot the sound like this:

```
plt.plot(snd)
```

```
[<matplotlib.lines.Line2D at 0x7f0d91e08860>]
```



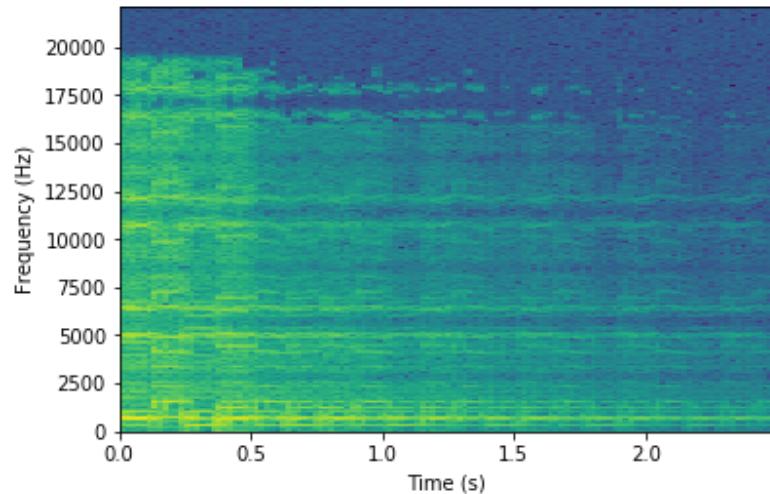
If each point in our dataset is a recorded sound, it is likely that each will have a different length. We could still represent our data in tabular form by taking each consecutive sample as a feature and padding with zeros the records that are shorter, but these extra zeros would carry no information (unless we had taken great care to synchronize each file so that the sound started at the same sample number).

Besides, sound information is carried in modulations of frequency, suggesting that the raw form may not be the best to use. As we shall see, there are better ways to represent sound and to feed it to a Neural Network

for tasks like music recognition or speech-to-text.

```
_ = plt.specgram(snd, NFFT=1024, Fs=44100)
plt.ylabel('Frequency (Hz)')
plt.xlabel('Time (s)')
```

```
Text(0.5,0,'Time (s)')
```



Text data

Text documents pose similar challenges. If each datapoint is a document, we need to find a good representation for it if we want to build a model that identifies it. We could use a dictionary of words and count the relative frequencies of words, but with Neural Networks we can do better than this.

In general this is called the *problem of representation*, and Deep Learning is a great technique to tackle it!

FEATURE ENGINEERING

As we have seen, unstructured data does not look like tabular data. The traditional solution to connect the two is *feature engineering*.

In feature engineering, an expert uses her domain knowledge to create features that correctly encapsulate the relevant information from the unstructured data. Feature engineering is fundamental to the application of Machine Learning, and it is both difficult and expensive.

For example, if we are training a Machine Learning model on a face recognition task from images, we could use well tested existing methods to detect a face and measure the distance between key points like eyes, mouth and nose. These distances would be the engineered features we would pass to the model being trained.

Similarly, in the domain of speech recognition, features based on [wavelets](#) and [Short Time Fourier Transforms](#) were the standard until not long ago.

Deep Learning disrupts feature engineering by **learning the best features directly from the raw unstructured data**. This approach is not only very powerful but also much much faster. This is a paradigm shift: more versatile technique taking the role of the domain expert.

EXERCISES

Now it's time to test what you've learned with a few exercises.

Exercise 1

- load the dataset: `../data/international-airline-passengers.csv`
- inspect it using the `.info()` and `.head()` commands
- use the function `pd.to_datetime()` to change the column type of 'Month' to a datetime type (you can find the doc [here](#))
- set the index of df to be a datetime index using the column 'Month' and the `df.set_index()` method
- choose the appropriate plot and display the data
- choose appropriate scale
- label the axes

Exercise 2

- load the dataset: `../data/weight-height.csv`
- inspect it
- plot it using a scatter plot with Weight as a function of Height
- plot the male and female populations with two different colors on a new scatter plot
- remember to label the axes

Exercise 3

- plot the histogram of the heights for males and for females on the same plot
- use `alpha` to control transparency in the plot command
- plot a vertical line at the mean of each population using `plt.axvline()`
- bonus: plot the cumulative distributions

Exercise 4

- plot the weights of the males and females using a box plot
- which one is easier to read?

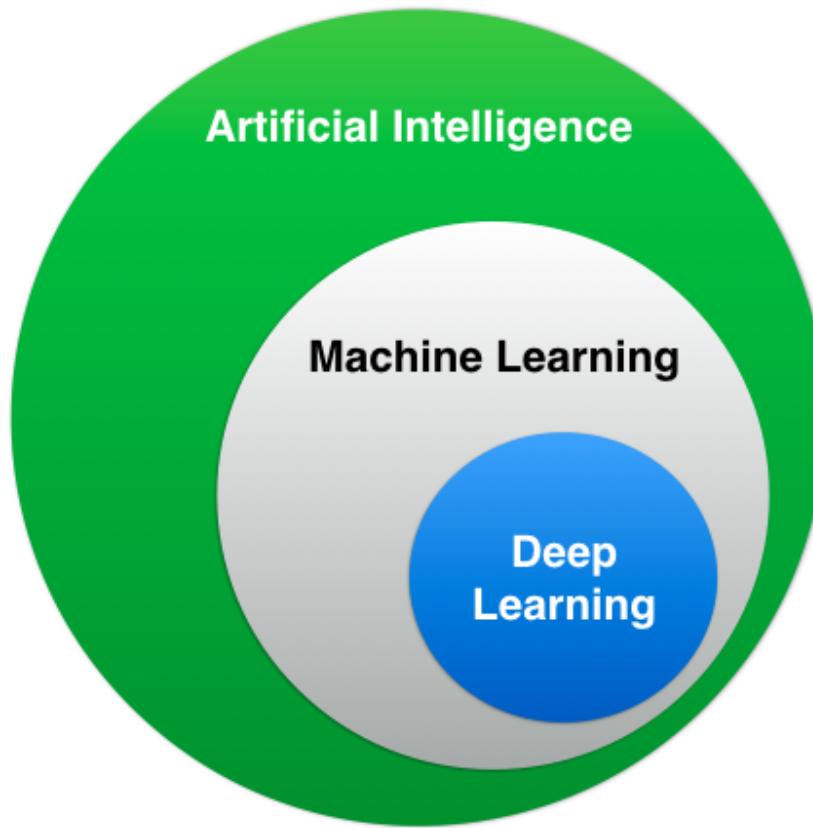
- (remember to put in titles, axes and legends)

Exercise 5

- load the dataset: `../data/titanic-train.csv`
- learn about `scattermatrix` [here](#)
- display the data using a `scattermatrix`

Chapter 3: Machine Learning

This chapter will introduce some common Machine Learning terms and techniques. In fact when we talk about Deep Learning we indicate a set of tools and techniques in Machine Learning that involve artificial Neural Networks.



Since for the rest of the book we will be using terms like `train_test_split` or `cross_validation` it makes sense to introduce these first and then move on to explain Deep Learning.

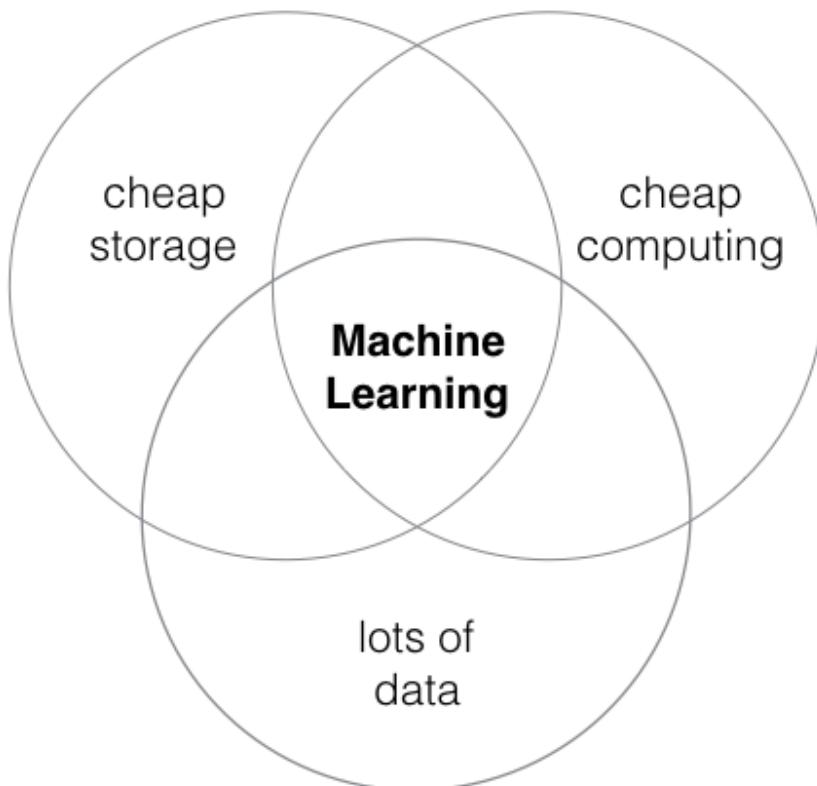
THE PURPOSE OF MACHINE LEARNING

Machine Learning is a branch of Artificial Intelligence that develops computer programs that are able to learn patterns and rules from data. Although its origins can be traced back to the early days of modern computer science, only in the last decade has Machine Learning become a **fundamental tool for companies in all industries**.

Product recommendation, advertisement optimization, automated translation, image recognition, self-driving cars, spam and fraud detection, automated medical diagnoses: these are just a few examples of how **Machine Learning is omnipresent in business and life**.

This revolution has largely been possible thanks to the combination of **3 factors**:

- cheap and powerful memory storage
- cheap and powerful computing power
- explosion of data collected by mobile phones, web apps and sensors



These same **3 factors are enabling the current Deep Learning and AI revolution**. Deep Neural Networks have been around for quite a while, but it wasn't until relatively recently that we've powerful enough computers (and large enough datasets) to make good use of them. This has changed in the last few years, and a lot of companies that used other Machine Learning techniques are now switching to Deep Learning.

Before we start studying Neural Networks, we need to make sure to have a **shared understanding of Machine Learning**, so this chapter is a quick summary of its main concepts.

If you are already familiar with terms like Regression, Classification, Cross-Validation and Confusion matrix, you may want to skim through this section quickly. However, make sure you understand **cost functions** and **parameter optimization** as they are fundamental for everything that will follow!

DIFFERENT TYPES OF LEARNING

There are **several types of Machine Learning**, including:

- supervised learning
- unsupervised learning
- reinforcement learning

	SUPERVISED	UNSUPERVISED	REINFORCEMENT
GOAL	Generalize from examples	Find structure in data	Learn behavior in environment
EXAMPLE	Catch spam messages	Group users by purchase habits	Win videogame

While this course will primarily focus on **supervised learning**, it is important to understand the difference in each of the types.

In **supervised learning** an algorithm learns from labeled data. For example, let's say we are training an image recognition algorithm to distinguish cats from dogs: each **training** datapoint will be the pair of an image (training data) and a label, which specifies if the image is a cat or a dog. Similarly, if we are training a translation engine, we will provide both input and output sentences, asking the algorithm to learn the function that connects them.

Conversely, in **unsupervised learning**, data comes without labels, and the task is to find similar datapoints in the dataset, in order to identify any underlying higher order structure. For example, in a dataset containing the purchase preferences of ecommerce users, these users will likely form clusters with similar purchase behavior in terms of amount spent, objects bought etc. We can think of these as different "tribes" with different preferences. Once these tribes are identified, we can describe each data point (that is, each user) in terms of the "tribe" it belongs to, gaining a deeper understanding of the data.

Finally, **reinforcement learning** is similar to supervised learning, but in this case **the algorithm is training an agent to act in an environment**. The actions of the agent lead to outcomes that are attached to a score and the algorithm tries to maximize such score. Typical examples here are algorithms that learn to play games, like Chess or Go. The main difference with supervised learning is that the score is that the algorithm does not receive a label (score) for each action it takes. Instead, it needs to perform a sequence of actions before it knows if that lead to a higher score.

In 2016 a software trained with reinforcement learning beat the world Go champion, marking a new milestone in the race towards artificial intelligence.

SUPERVISED LEARNING

Let's dive into supervised learning by first reviewing some of its successful applications. Have you ever noticed that email spam is practically non-existent any longer? This thought is thanks to supervised learning.

In the early 2000s, mailboxes were plagued by tons of emails advertising pills, money making schemes and other crappy information. The first step to get rid of these was to allow users to move spam emails into a *spam* folder. This provided the training labels. With million of users manually cataloguing spam, large email providers like Google and Yahoo could quickly gather enough examples of what a spam mail looked like to train a model that would predict the probability for a message to be spam.

This technique is called a **binary classifier**, and it is a **Machine Learning algorithm that learns to distinguish between 2 classes**, like true or false, spam or not spam, positive or negative sentiment, dead or alive.

Binary classifiers trained with supervised learning are ubiquitous. Telecom companies use them to predict if a user is about to churn and go to a competitor, so they know when and to whom to make an offer in order to retain them.

Social media analytics companies use binary classifiers to judge the prevalent sentiment on their clients' pages. If you are a celebrity, you receive millions of comments each time you post something on Facebook or Twitter. How can you know if your followers were prevalently happy or angry at what you tweeted? A sentiment analysis classifier can distinguish that for each single comment, and therefore give us the overall reaction by aggregating over all comments.

The movie is great!



The movie is about AI.



The movie is terrible!



Supervised learning is also used to predict continuous quantities, for example to forecast retail sales of next month or to predict how many cars there will be at a certain intersection in order to offer better route for car navigation. In this case the labels are not discrete like "true/false" "black/blue/green" but they have a continuous values, like 68, 73, 71 if we're trying to predict temperature.

What other examples of supervised learning can you think of?

LINEAR REGRESSION

Let's take a second look at the plot we drew in Exercise 2 of Section 2. As we know by now, it represents a population of individuals. Each dot is a person, and the position of the dot on the chart is defined by two coordinates: weight and height. Let's plot it again:

```
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

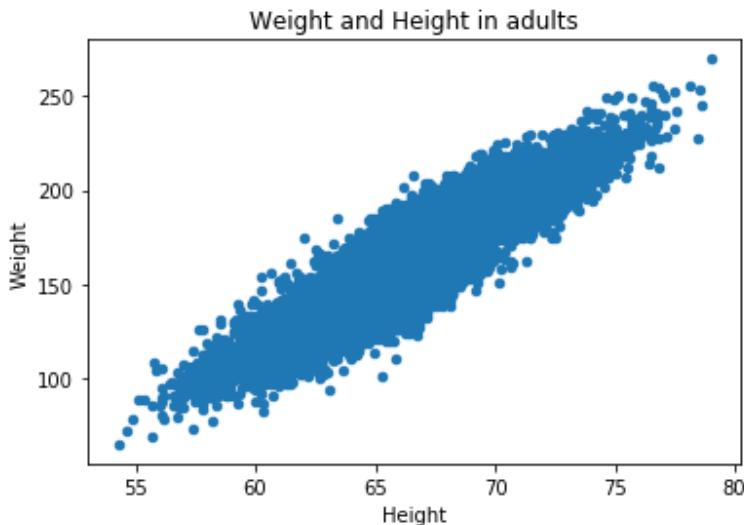
```
df = pd.read_csv('~/data/weight-height.csv')
```

```
df.head()
```

	Gender	Height	Weight
0	Male	73.847017	241.893563
1	Male	68.781904	162.310473
2	Male	74.110105	212.740856
3	Male	71.730978	220.042470
4	Male	69.881796	206.349801

```
def plot_humans():
    df.plot(kind='scatter',
            x='Height',
            y='Weight',
            title='Weight and Height in adults')

plot_humans()
```



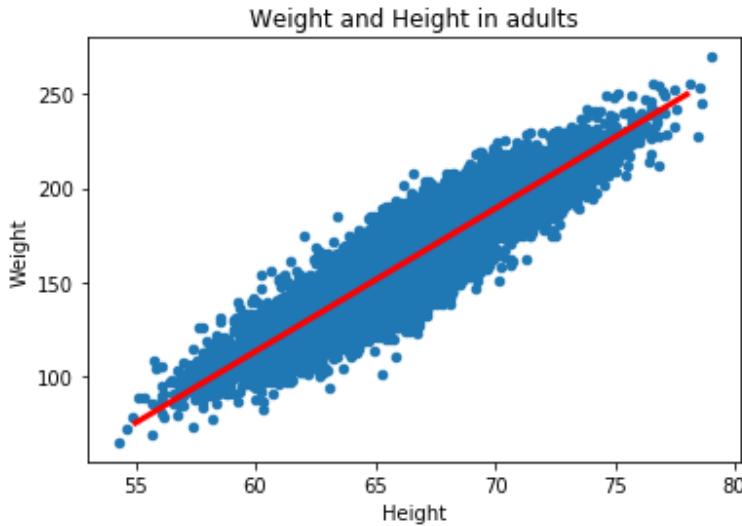
Can we tell if there is a pattern in how the dots are laid out on the graph or do they seem completely randomly spread? Our visual cortex, a great pattern recognizer, tells us that there is a pattern: dots are roughly spread around a straight diagonal line. This line seems to indicate the obvious: taller people are also heavier on average.

Let's sketch a line to represent this relation. We can plot this line "by hand", without any learning, by choosing the values of the two extremities. For example, let's draw a segment that starts at the point [55, 78] and ends at the point [75, 250].

```
plot_humans()

# Here we're plotting the red line 'by hand' with fixed values
# We'll try to learn this line with an algorithm below
plt.plot([55, 78], [75, 250], color='red', linewidth=3)
```

[<matplotlib.lines.Line2D at 0x7f16406d5208>]



Can we specify this relationship more precisely? Instead of guessing the position of the line, can we ask an algorithm to find the best possible line to describe our data? The answer is yes! Let's see how.

We are saying that **weight (our target or label) is a linear function of height (our only feature)**.

Let's assign variable names to our quantities. As we saw in the introduction, it is common to assign the letter y to the labels (people's weight in this case) and the letter X to the input features (only height in this case).

You may remember from high school math that a line in a 2D-space can be described by an equation between X and y that involves only two parameters. One parameter controls the point where the line crosses the vertical axis, the other controls the slope of the line. We can write the equation of a line in a 2D plane as:

$$\hat{y} = b + Xw$$

where \hat{y} is pronounced y-hat. Let's first convince ourselves that this indicates any possible line in the 2D plane (with the exception of a perfectly vertical line).

If we choose $b = 0$ and $w = 0$, we obtain the equation $\hat{y} = 0$ for any value of X . This is the set of points that form the horizontal line passing through zero.

If we start changing b , we will obtain $\hat{y} = b$, which is still a horizontal line, passing through the constant point b . Finally, if we also change w the line will start to be inclined in some way.

So yes, any line in the 2D-plane, except a vertical line, will have its unique values for w and b .

To find a linear relation between X and y means to describe our labels y as a linear function of X plus some small correction ϵ :

$$y = b + Xw + \epsilon = \hat{y} + \epsilon$$

It's good to get used to distinguish between the values of the output (y , our labels) and the values of the predictions (\hat{y}).

Let's draw some examples.

In this chapter we are going to explain how an algorithm can find the perfect line to fit a dataset. Before writing an algorithm it's helpful to understand the dynamics of this line formula. So what we're going to do is draw a few plots where we change the values of b and w and see how they affect the position of the line in the 2D plane. This will give us better insight when we try to automate this process.

Let's start by defining a simple line function:

```
def line(x, w=0, b=0):  
    return x * w + b
```

Then let's create an array of equally spaced x values between 55 and 80 (these are going to be the values of height):

```
x = np.linspace(55, 80, 100)  
x
```

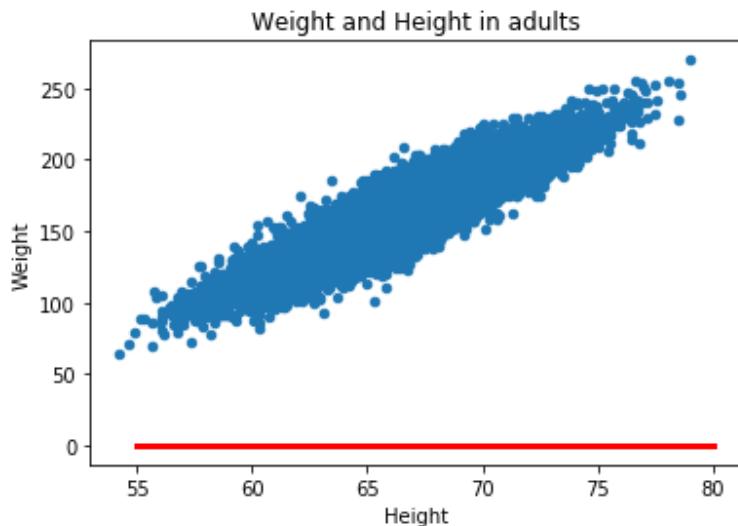
```
array([55.          , 55.25252525, 55.50505051, 55.75757576, 56.01010101,  
      56.26262626, 56.51515152, 56.76767677, 57.02020202, 57.27272727,  
      57.52525253, 57.77777778, 58.03030303, 58.28282828, 58.53535354,  
      58.78787879, 59.04040404, 59.29292929, 59.54545455, 59.7979798 ,  
      60.05050505, 60.3030303 , 60.55555556, 60.80808081, 61.06060606,  
      61.31313131, 61.56565657, 61.81818182, 62.07070707, 62.32323232,  
      62.57575758, 62.82828283, 63.08080808, 63.33333333, 63.58585859,  
      63.83838384, 64.09090909, 64.34343434, 64.5959596 , 64.84848485,  
      65.1010101 , 65.35353535, 65.60606061, 65.85858586, 66.11111111,  
      ...  
      78.98989899, 79.24242424, 79.49494949, 79.74747475, 80.        ])
```

And let's pass these values to the line function and calculate \hat{y} . Since both w and b are zero, we expect \hat{y} to also be zero:

```
yhat = line(x, w=0, b=0)  
yhat
```

```
plot_humans()
```

[<matplotlib.lines.Line2D at 0x7f164063f710>]



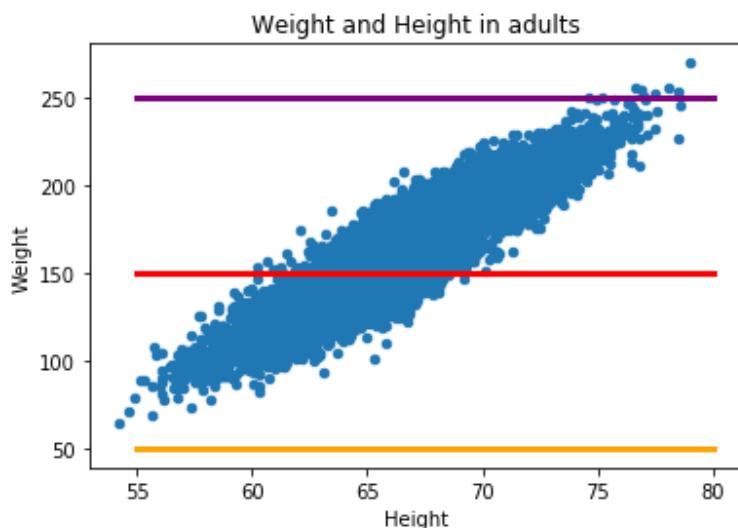
So we've drawn a horizontal line as our model. This is not really a good model for our data! It would be a good model if everyone in our population was floating in space and therefore measured 0 weight regardless of their height. Fun, but not accurate for our chart! See how far the line is from our data.

If we let b vary, the horizontal line starts to move up or down, indicating constant weight b , regardless of the value of x (the height).

```
plot_humans()

# three settings for b "offset" the
plt.plot(x, line(x, b=50), color='orange', linewidth=3)
plt.plot(x, line(x, b=150), color='red', linewidth=3)
plt.plot(x, line(x, b=250), color='purple', linewidth=3)
```

[<matplotlib.lines.Line2D at 0x7f16405e6fd0>]



This would be a good model only if we had a broken scale that always returned a fixed value, regardless of who steps on it. Also not accurate.

Finally, if we vary w , the line starts to tilt, with w indicating the *increment in weight* corresponding to the increment in 1 unit of height. For example, if $w=1$, that would imply that **1 pound is gained for each inch of height**.

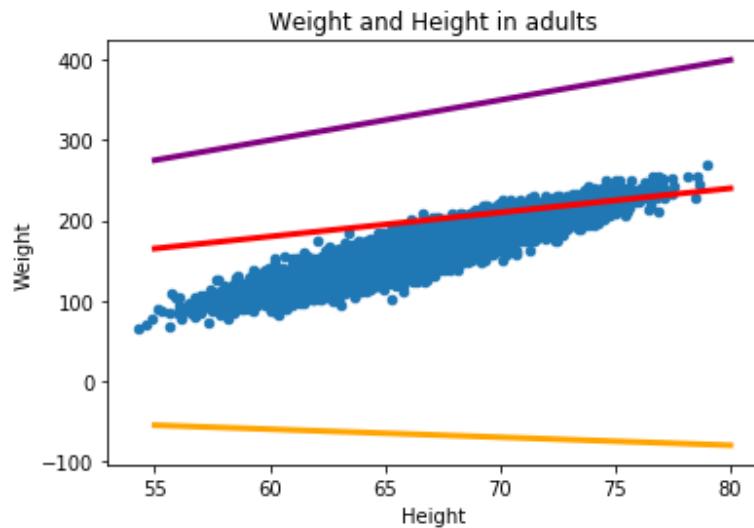
```
plot_humans()
```

```

plt.plot(x, line(x, w=5), color='purple', linewidth=3)
plt.plot(x, line(x, w=3), color='red', linewidth=3)
plt.plot(x, line(x, w=-1), color='orange', linewidth=3)

```

[<matplotlib.lines.Line2D at 0x7f164055e3c8>]



So, to recap, we started from the intuitive observation that taller people are heavier and we decided to look for a line function to predict the weight of a person as a function of the height.

Then we observed that any line in the 2D plane can be drawn by defining just two parameters, b and w , we plotted a few such lines and compared them with our data. Now we need to find the values of such parameters that correspond to the best line for our data.

Cost Function

In order to find the best possible linear model to describe our data, we need to define a criterion to evaluate the "goodness" of a particular model.

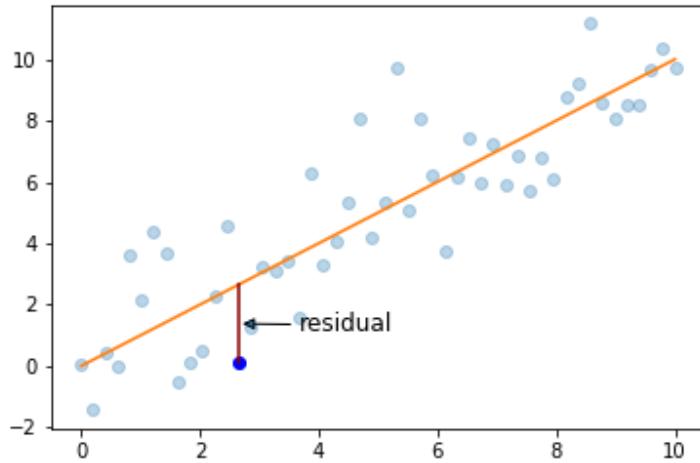
In supervised learning **we know the values of the labels**. So we can **compare the value predicted by the hypothesis with the actual value of the label** and calculate the error for each datapoint:

$$\epsilon_i = y_i - \hat{y}_i$$

Remember that y_i is the actual value of the output while \hat{y}_i is our prediction. Also notice that we used a subscript index i to indicate the i -th datapoint. Each datapoint difference is a residual and the group of them

together are the *residuals*.

Note that in this definition, a residual carries a sign, it will be positive if our hypothesis underestimates the true weight and negative if it overestimates it.



Since we don't really care about the direction in which our hypothesis is wrong (we only care about the total amount of being wrong), we can define the *total error* as the sum of the absolute values of the residuals:

$$\text{Total Error} = \sum_i |\epsilon_i| = \sum_i |y_i - \hat{y}_i|$$

The **total error** is one possible example of what's called a **cost function**. We have associated a well defined cost that can be calculated from our features and labels through the use of our hypothesis $\hat{y} = h(x)$.

For reasons that will be apparent later, it is often preferable to use another cost function called **Mean Squared Error**. This is defined as:

$$\text{MSE} = \frac{1}{N} \sum_i (y_i - \hat{y}_i)^2$$

where N is the number of datapoints used to train our model.

Notice that since the square is a positive function, this will be big when the total error is big and small when the total error is small. In that sense they are equivalent. The *Mean Squared Error* (or 'mse', for short) is preferred because it's smooth and guaranteed to have a global minimum, which is exactly what we are going to look for.

Finding the best model

Now that we have both a hypothesis (linear model) and a cost function (mean squared error), we need to find the combination of parameters b and w that **minimizes such cost**.

Remember, that **cost** is another way to say the 'error amount' of our prediction - we're assigning a number to how wrong our prediction is. We want to *minimize* this error (cost) because if our error was zero, that means we predicted perfectly.

Let's first define a helper function to calculate the MSE and then evaluate the cost for a few lines:

```
def mean_squared_error(y_true, y_pred):
    s = (y_true - y_pred)**2
    return s.mean()
```

Let's also define inputs and outputs for our data. Our input is the height column. We will assign it to the variable X :

```
X = df[['Height']].values
X
```

```
array([[73.84701702],
       [68.78190405],
       [74.11010539],
       ...,
       [63.86799221],
       [69.03424313],
       [61.94424588]])
...
[61.94424588]])
...
[61.94424588]])
```

Notice that X is matrix with 10000 rows and a single column:

```
X.shape
```

```
(10000, 1)
```

This format will allow us to extend the linear regression to cases where we want to use more than one column as input.

Then let's define the outputs:

```
y_true = df['Weight'].values  
y_true
```

```
array([241.89356318, 162.31047252, 212.74085556, ..., 128.47531878,  
      163.85246135, 113.64910268])  
...  
      163.85246135, 113.64910268])  
...  
      163.85246135, 113.64910268])
```

The outputs are a single array of values. What is the cost going to be for the horizontal line passing through zero? We can calculate it as follows.

First we generate predictions for each value of X :

```
y_pred = line(X)  
y_pred
```

```
array([[0.],  
      [0.],  
      [0.],  
      ...,  
      [0.],  
      [0.],  
      [0.]])  
...  
      [0.]])  
...  
      [0.]])
```

And then we calculate the cost, i.e. the mean squared error between these predictions and our true values:

```
mean_squared_error(y_true, y_pred.ravel())
```

27093.83757456157

Notice that we flattened out the predictions so that it has the same shape as the output vector.

The cost is above 27,000. What does it mean? Is it bad? Is it good? It's really hard to say because we don't have anything to compare it to. Different datasets will have very different numbers here depending on the units of measure of the quantity we are predicting. So the value of the cost has very little meaning by itself. What we need to do is compare this cost with that of another choice of b and w . Let's increase w a little bit:

```
y_pred = line(X, w=2)
mean_squared_error(y_true, y_pred.ravel())
```

1457.1224504786412

The total mse decreased from over 27000 to below 2000. This is good! It means our new hypothesis with $w=2$ is less-wrong than using $w=0$.

Let's see what happens if we also change b :

```
y_pred = line(X, w=2, b=20)
mean_squared_error(y_true, y_pred.ravel())
```

708.9129575511095

Even better! As you can see we can keep changing b and w by small amounts and the value of the cost will keep changing.

Of course, it's going to take forever for us to find the best combination if we sit here and tweak numbers until we find the best ones. A better way would be if we could write a program that would test all possible values for us and then simply report to us the result.

Before we do that, let's check a couple of other combinations of w and b . Let's try to keep w fixed and vary only b .

```

plt.figure(figsize=(10, 5))

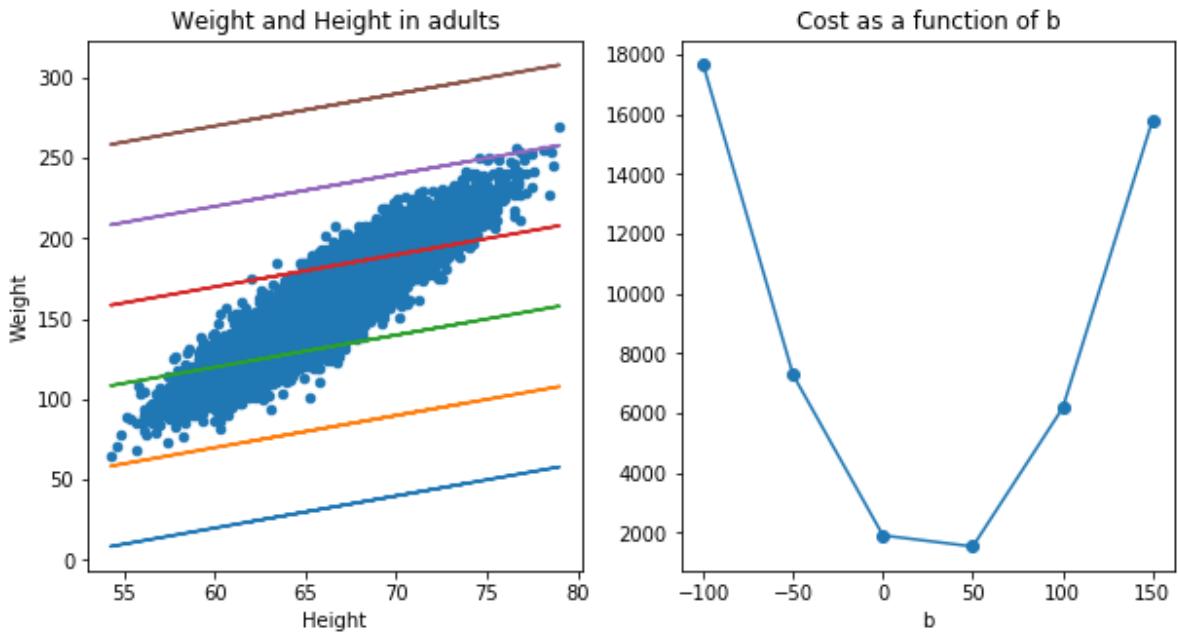
# we are going to draw 2 plots in the same figure
# first plot, data and a few lines
ax1 = plt.subplot(121)
df.plot(kind='scatter',
        x='Height',
        y='Weight',
        title='Weight and Height in adults', ax=ax1)

# let's explore the cost function for a few values of b between -100 and +150
bbs = np.array([-100, -50, 0, 50, 100, 150])
mses = [] # we will append the values of the cost here, for each line
for b in bbs:
    y_pred = line(X, w=2, b=b)
    mse = mean_squared_error(y_true, y_pred)
    mses.append(mse)
    plt.plot(X, y_pred)

# second plot: Cost function
ax2 = plt.subplot(122)
plt.plot(bbs, mses, 'o-')
plt.title('Cost as a function of b')
plt.xlabel('b')

```

Text(0.5, 0, 'b')



When $w = 2$, the cost as a function of b has a minimum value somewhere near 50.

The same would be true if we let w vary, there will be a value of w for which the cost is minimum. Since we choose a cost function that is *quadratic* in b and w , there is a *global minimum*, corresponding to the combination of parameters b and w that minimize the mean squared error cost.

TIP: A **quadratic function** is a polynomial function in one or more variables in which the highest-degree term is of the second degree. This is a very nice feature, that guarantees us that there is only one minimum, and therefore it is the global one.

Once our parameters w and b are set to the combination that minimizes the cost, we can say that the model is trained over the training set.

Notice what just happened:

- We started with a hypothesis: height and weight are connected by a linear model that depends on parameters.
- We defined a cost function: the mean squared error is calculated for each combination of b and w using the training set features and labels.
- Finally, we minimized the cost: the model is trained when we have found the values of b and w that minimize the cost over the training set.

Another way to say this is that we have turned the problem of training a Machine Learning model into a **minimization problem**, where our cost defines a "landscape" made of valleys and peaks, and we are looking

for the global minimum.

This is great news, because there are plenty of techniques to look for the minimum value of a function.

TIP: We solved a Linear Regression problem using Gradient Descent. This was not really necessary, since Linear Regression has an exact solution. We used this simple case to introduce the Gradient Descent technique that we will use throughout the book to train our Neural Networks.

Linear Regression with Keras

Let's see if we can use Keras to perform linear regression. We will start by importing a few elements to build a model, as we did in chapter 1.

```
from keras.models import Sequential  
from keras.layers import Dense  
from keras.optimizers import Adam, SGD
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: Fu  
      from ._conv import register_converters as _register_converters  
Using TensorFlow backend.
```

The model we need to build is super simple: it has one input and one output, connected by one parameter w and one parameter b . Let's do that! First, we initialize the model as a `Sequential` model. This is the simplest way to define models in Keras, because we add layers one by one, starting from the input and working our way towards the output.

```
model = Sequential()
```

Then we add a single element to our model: a linear operation with 1 input and 1 output, connected by the two parameters w and b . In `keras` this is done with the `Dense` class. In fact, from the documentation of `Dense` we read:

Just your regular densely-connected NN layer.

`Dense` implements the operation:
`output=activation(dot(input, kernel) + bias)`

We can recover our notation with the following substitutions:

```
output -> y  
activation -> None  
input -> X  
kernel -> w  
bias -> b
```

and noticing that the dot product with a single input is just the multiplication. So `Dense(1, input_shape=(1,))` implements a linear function with 1 input and 1 output. Let's add it to the model:

```
model.add(Dense(1, input_shape=(1,)))
```

The `.summary()` method will tell us the number of parameters in our model:

```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 1)	2

Total params: 2
Trainable params: 2
Non-trainable params: 0

...

As expected our model has 2 parameters: the bias or b and the kernel or weight or w . Let's go ahead and *compile* the model.

Compilation tells Keras what the cost function is (Mean Squared Error in this case) and what method we choose to find the minimum value (the `Adam` method in this case).

TIP If you have never seen an optimizer don't worry about it, we will explain it in detail later in the book.

```
model.compile(Adam(lr=0.8), 'mean_squared_error')
```

Now that we have compiled the model, let's go ahead and train it on our data. To train a model we use the method `model.fit(X, y)`. This method requires the input data and the input labels and it trains a model by minimizing the value of the cost function over the training data. As an additional parameter to the fit model, we will pass the number of epochs. This is the number of time we want the training to loop over the whole training dataset.

TIP: an **Epoch** in deep learning indicates one cycle over the training dataset. At the end of an epoch the model has seen each pair of `(input, output)` once.

In this example we train the model for 40 epochs, which means we will cycle through the whole `(X, y_true)` dataset 40 times.

```
model.fit(X, y_true, epochs=40)
```

```
Epoch 1/40
10000/10000 [=====] - 2s 178us/step - loss: 1433.9738
Epoch 2/40
10000/10000 [=====] - 1s 58us/step - loss: 557.6043
Epoch 3/40
10000/10000 [=====] - 1s 59us/step - loss: 537.5011
Epoch 4/40
10000/10000 [=====] - 1s 60us/step - loss: 500.2912
Epoch 5/40
...
Epoch 5/40
```

```
<keras.callbacks.History at 0x7f15a50a2c88>
```

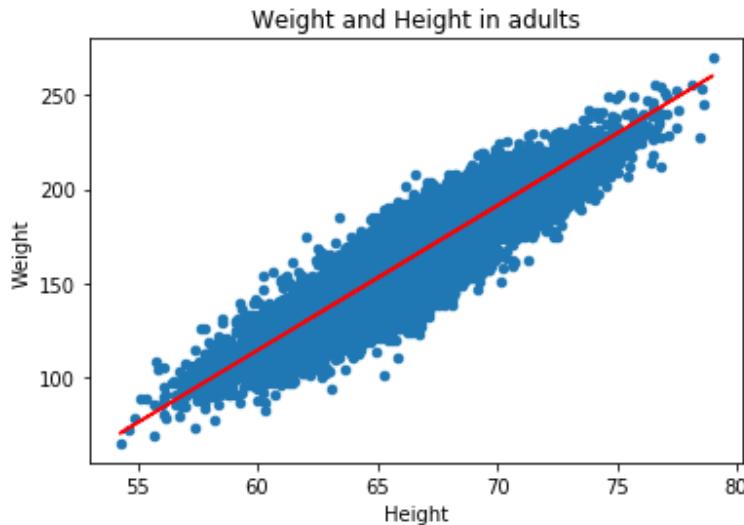
Notice that the value of our loss stopped decreasing at some point. Let's see how well the model fits our data. We will store our predictions in a variable called `y_pred` and plot them over the data:

```
y_pred = model.predict(X)
```

```
df.plot(kind='scatter',
        x='Height',
        y='Weight',
```

```
    title='Weight and Height in adults')
plt.plot(X, y_pred, color='red')
```

```
[<matplotlib.lines.Line2D at 0x7f15a4034160>]
```



The line is not perfectly where we would have liked it to be, but it seems to have captured the relationship between Height and weight quite well. We can inspect the parameters of the model to see what values our training decided were optimal for b and w .

```
w, b = model.get_weights()
```

```
w
```

```
array([[7.7028933]], dtype=float32)
```

```
b
```

```
array([-348.00958], dtype=float32)
```

Notice here that W is returned as a matrix, because in the general case of a Neural Network we could have many more parameters. In this simple case of a linear regression our matrix has 1 row and 1 column and a single number for the slope of our line, so let's extract it.

$$w = W[0, 0]$$

B is also a vector with just one entry, so we can extract that too:

$$b = B[0]$$

The slope parameter w has a value near 7.7 . This means that, for 1 inch increase, people are on average 7.7 pounds heavier. The b parameter is roughly -350 . This is called *offset* or *bias* and corresponds to the weight of an adult of zero height.

Since negative weight doesn't actually make sense, we have to be careful about how we interpret this value. Let's see if we can see what's the minimum height that makes sense in this model. This will be the height that produces a weight of zero, since negative weights are nonsense. Zero weight means $y = 0$, so now that we have a model we can look for the value of X that corresponds to $y = 0$.

Setting $y = 0$ in the line equation gives:

$$0 = Xw + b$$

and we can shuffle things around to obtain:

$$X = \frac{-b}{w}$$

Let's calculate it:

$$-b/w$$

$$45.179073$$

So this model only makes sense for people who are at least about 45 inches tall. If you are shorter than 45 inches, this model predicts you'd have a negative weight, which is obviously wrong.

Evaluating Model Performance

Great! We have trained our **first supervised learning model** and have found the best combination of parameters b and w . But is this really a good model? Can we trust it to predict the height of new people that were not part of the training data? In other words, will our model "generalize well" when offered new, unknown data? Let's see how we can answer that question.

R^2 coefficient of determination

First of all we need to define a sort of standard score, a number that will allow us to compare the *goodness* of a model regardless of how many data points we used. We could compare losses, but the value of the loss is ultimately arbitrary and dependent on the scale of the features, so we don't want to use that. Instead, let's use the *coefficient of determination* R^2 .

This coefficient can be defined for any model predicting continuous values (like regression) and it will give some information about the goodness of fit. In the case of regression, the R^2 coefficient is a measure of how well the regression model approximates the real data points. An R^2 of 1 indicates a regression line that perfectly fits the data.

If the line does not perfectly fit the data, the value of R^2 , will decrease. A value of 0 or lower indicates that the model is not a good one.

We recall here **Scikit-Learn**, a Python package introduced in chapter 1 that contains many Machine Learning algorithms and supporting functions, including the R^2 score. Let's calculate it for the current model.

First of all, let's import it:

```
from sklearn.metrics import r2_score
```

and then let's calculate it on the current predictions:

```
print("The R2 score is {:.3f}".format(r2_score(y_true, y_pred)))
```

```
The R2 score is 0.852
```

TIP: In the last command we introduced a way to define **Python format**, to make numbers more readable. In particular, we specified the format `{:.3f}`. The brackets and characters within them (called format

fields) are replaced with the objects passed into the `str.format()` method. The integer after the `:` will cause that field to be a minimum number of characters wide, `0` in this case. `3` indicates the number of decimal digits and `f` stands for a floating point decimal format.

It's not too far from 1, which means our regression is not too bad. It doesn't answer the question about generalization though, how can we know if our model is going to generalize well?

Train / Test split

Let's go back to our dataset. What if, instead of using all of it to train our model, we held out a small fraction of it, say 20% randomly sampled points. We could train the model on the remaining 80% and use the 20% to test how good the model actually is. This would be a good way to test if our model is **overfitting**.

Overfitting means that our model is just memorizing the answers instead of learning general rules about the training examples. By withholding a test set, we can test our model on data never seen before. If it performs just as well we can assume it will perform well on new data when deployed.

On the other hand, if our model has a good score on the training set but has a bad score on the test set, this would mean it is not able to generalize to unseen examples, and therefore it's not ready for deployment.

This is called a train/test split, it's standard practice for supervised learning and there's a convenient Scikit-Learn function for it.

```
from sklearn.model_selection import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split(X, y_true,  
                                                test_size=0.2)
```

```
len(x_train)
```

8000

```
len(x_test)
```

2000

Using `train_test_split` we split the data into two sets, the `training set` and the `test set`. Now we can use each according to its name: we let the parameters of our models vary to minimize the cost over the training set and then check the cost and the R^2 score over the test set. If things went well, these two should be comparable, i.e. the model should perform well on new data. Let's do that!

First, let's train our model on the **training data** (notice the test data is not involved here):

```
model.fit(X_train, y_train, epochs=50, verbose=0)
```

```
<keras.callbacks.History at 0x7f14f06bd8d0>
```

Then let's calculate predictions for both the train and test sets.

TIP: Note that unlike training, making predictions is a "read-only" operation and does **not change** our model. We're just making predictions.

```
y_train_pred = model.predict(X_train).ravel()
y_test_pred = model.predict(X_test).ravel()
```

Let's calculate the mean squared error and the R^2 score for both. We will also import the `mean_squared_error` function from Scikit-Learn, which does the same calculation as the function we defined above, but it's probably better defined.

```
from sklearn.metrics import mean_squared_error as mse
```

```
err = mse(y_train, y_train_pred)
print("Mean Squared Error (Train set):\t{:0.1f}".format(err))

err = mse(y_test, y_test_pred)
print("Mean Squared Error (Test set):\t{:0.1f}".format(err))
```

```
Mean Squared Error (Train set): 185.3  
Mean Squared Error (Test set): 183.1
```

```
r2 = r2_score(y_train, y_train_pred)  
print("R2 score (Train set):\t{:0.3f}".format(r2))  
  
r2 = r2_score(y_test, y_test_pred)  
print("R2 score (Test set):\t{:0.3f}".format(r2))
```

```
R2 score (Train set): 0.820  
R2 score (Test set): 0.823
```

It appears that both the loss and the R^2 score are comparable for the Train and Test set, which is great! If we had obtained values that were significantly different, we would have had a problem. Generally speaking our test set could perform a little worse because the test data hasn't been seen before. If the performance on the test set is significantly lower than on the training set, we are *overfitting*.

TIP: The test fraction does not need to be 20%. We could use 5%, 10%, 30%, 50% or anything we like. Keep in mind that if we do not use enough data for testing, we may not have a credible test of how well the model generalizes, while if we use too much testing data, we make it harder for the model to learn because it is only exposed to few examples.

Note that this is another reason to prefer an average cost (i.e divided by the total number of sample points) rather than a total cost. In this way, the cost will not depend on the size of the set used to calculate it and we will be therefore able to compare costs obtained over sets of different sizes.

Congratulations! We have just encountered the three basic ingredients of a Neural Network: **hypothesis with parameters, cost function and optimization**.

CLASSIFICATION

So far we have just learned about linear regression and how we can use it to predict a continuous target variable. We have learned about formulating a *hypothesis* that depends on *parameters* and about *optimizing a cost* to find the optimal values for such parameters.

We can apply the same framework to cases where the target variable is discrete and not continuous. All we need to do is to **adapt the hypothesis and the cost function**.

Let's see how that is done. Let's imagine we are predicting whether a visitor on our website is going to buy a product, based on how many seconds he/she spent on the product page. In this case, the outcome variable is binary: the user either buys or doesn't buy the product. How can we build a model with a binary outcome? Let's load some data and find out:

```
df = pd.read_csv('..../data/user_visit_duration.csv')
```

```
df.head()
```

	Time (min)	Buy
0	2.000000	0
1	0.683333	0
2	3.216667	1
3	0.900000	0
4	1.533333	1

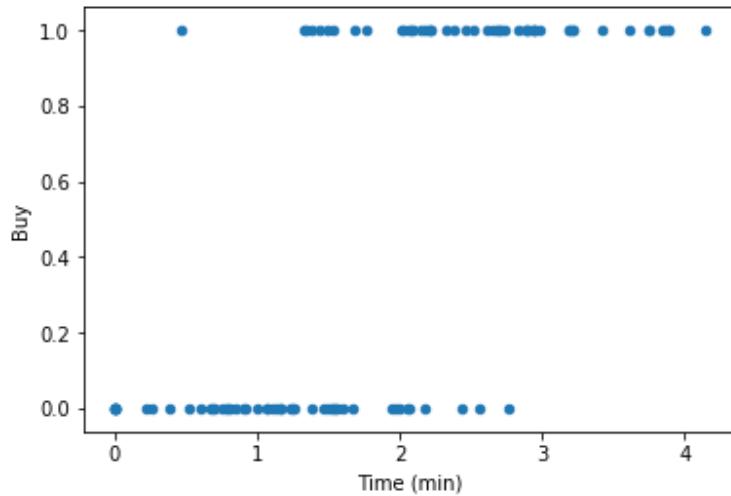
The dataset we loaded has 2 columns:

- Time (min)
- Buy

and we can plot it like this:

```
df.plot(kind='scatter', x='Time (min)', y='Buy')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f14f06f36d8>
```



Since the outcome variable can only assume a finite set of distinct values (only 0 and 1 in this case), this is a *classification* problem, i.e. we are looking for a model that is capable of predicting to which class a data point belongs.

TIP: There are many algorithms to solve a classification problem, including K-nearest neighbors, decision trees, support vector machines and Naive Bayes classifiers. The interested reader is referred to our other book on Machine Learning for an in-depth explanation of each of them.

Linear regression fail

What happens if we use the same model we have just used to fit this data? Will the model refuse to work? Will it converge? Will it give helpful predictions?

Let's try it and see what happens. First we need to define our features and target variables.

```
X = df[['Time (min)']].values  
y = df['Buy'].values
```

Then we can use the exact same model we used before. We will simply re-initialize it by resetting the parameter w to 1 and b to 0:

```
model.set_weights([[[ 1.0]], [0.]])
```

Then we fit the model on X and y for 200 epochs, suppressing the output with `verbose=0` :

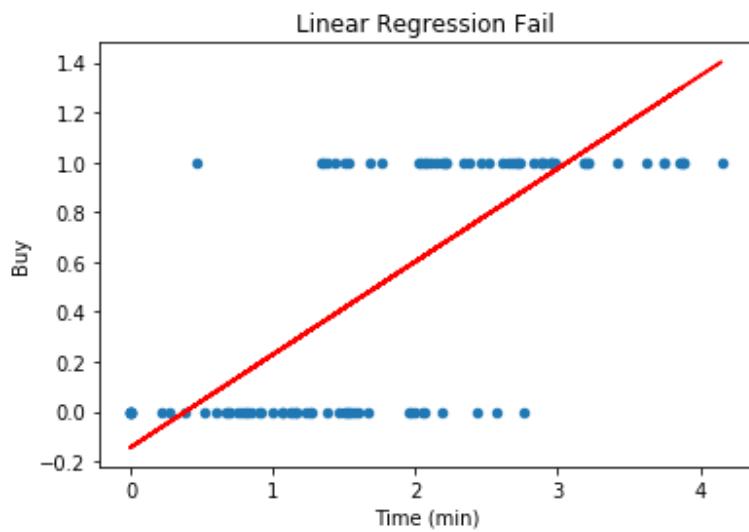
```
model.fit(X, y, epochs=200, verbose=0)
```

```
<keras.callbacks.History at 0x7f14f06bd940>
```

Let's see what the predictions look like:

```
y_pred = model.predict(X)

df.plot(kind='scatter', x='Time (min)', y='Buy', title='Linear Regression Fail')
plt.plot(X, y_pred, color='red')
plt.show()
```



As you can see the linear regression it doesn't make much sense to use a straight line to predict an outcome that can only either 0 or 1. That said, the modification we need to apply to our model in order to make it work is actually quite simple.

Logistic Regression

We will approach this problem with a method called *Logistic Regression*. Despite the name being "regression", this technique is actually useful to solve classification problems, i.e. problems where the outcome is discrete.

The linear regression technique we have just learned predicts values in the real axis for each input data point. Can we modify the form of the hypothesis so that we can **predict the probability** of an outcome? If we can do that, for each value in input, our model would give us a value between 0 and 1. At that point we could use $p = 0.5$ as our dividing criterion and assign every point predicted with probability less than 0.5 to class 0, and every point predicted with probability more than 0.5 to class 1.

In other words, if we modify the regression hypothesis to allow for a nonlinear function between the domain of our data and the interval $[0, 1]$, we can use the same machinery to solve a classification problem.

There's actually one additional point we will need to address, which is how to adapt the cost function. In fact, since our labels are only the values 0 and 1 the **Mean Squared Error is not the correct cost function to use**. We will see below how to define a cost that works in this case.

Let us first start by defining a nonlinear hypothesis. We need a nonlinear function that will map all of the real axis into the interval $[0, 1]$. There are many such functions and we will see a few in the next chapters. A simple, smooth and well-behaved function is the **Sigmoid** function:

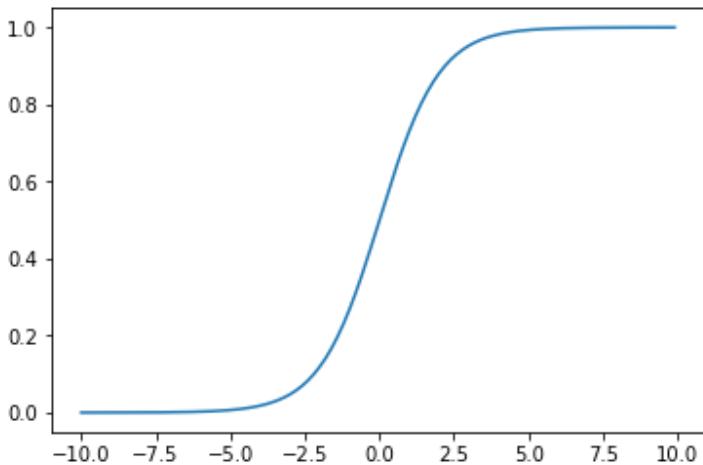
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

which looks like this:

```
def sigmoid(z):
    return 1.0/(1.0 + np.exp(-z))

z = np.arange(-10, 10, 0.1)
plt.plot(z, sigmoid(z))
```

[<matplotlib.lines.Line2D at 0x7f14f002df28>]



The Sigmoid starts at values really close to 0 for negative values of x . Then it gradually increases and near $x = 0$ it smoothly transitions to values close to 1. Mathematically speaking, the sigmoid function is like a smooth step function.

Hypothesis

Using the sigmoid we can formulate the *hypothesis* for our classification problem as:

$$\text{Buy} = \frac{1}{1+e^{-(Time \ w+b)}}$$

or

$$\hat{y} = \sigma(Xw + b)$$

We will encounter this function many times in this book. In fact it has been a very important function in the early days of Neural Networks and it is still very important.

Notice that we have introduced two parameters, w and b , in our definition. One of them controls the speed of the transition between 0 and 1, while the other controls the position of the transition. Let's plot a few examples:

```
x = np.linspace(-10, 10, 100)

plt.figure(figsize=(15, 5))

plt.subplot(121)

ws = [0.1, 0.3, 1, 3]
```

```

for w in ws:
    plt.plot(x, sigmoid(line(x, w=w)))

plt.legend(ws)
plt.title('Changing w')

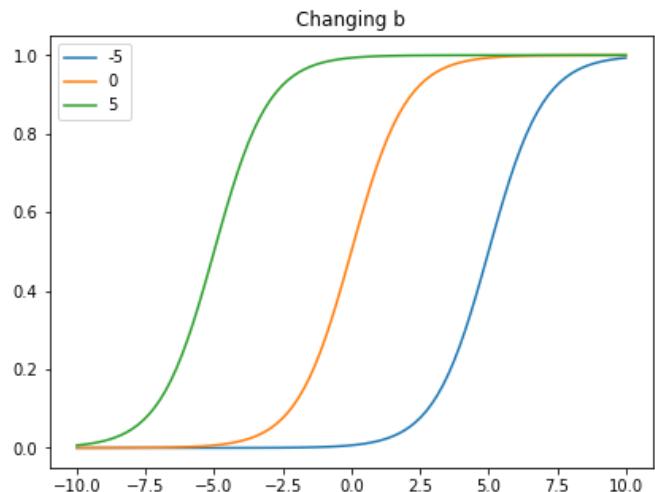
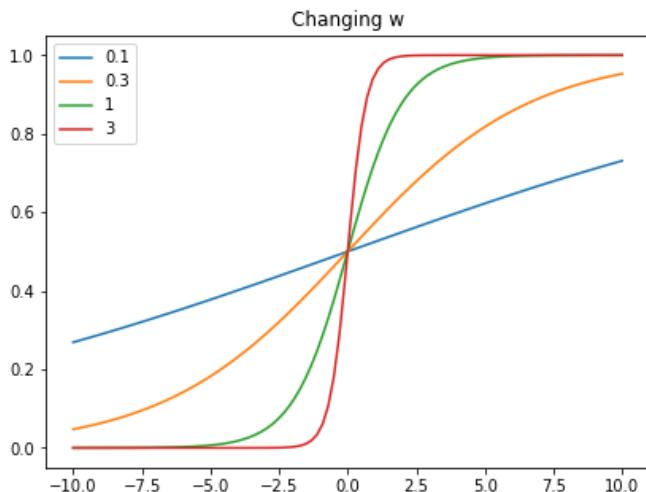
plt.subplot(122)

bs = [-5, 0, 5]
for b in bs:
    plt.plot(x, sigmoid(line(x, w=1, b=b)))

plt.legend(bs)
plt.title('Changing b')

```

Text(0.5,1,'Changing b')



Cost function

Now that we have defined the hypothesis, we need to adapt the definition of the cost function so that it makes sense for a binary classification problem. There are various options for this, similarly to the regression case, including **square loss**, **hinge loss** and **logistic loss**.

As we shall see in chapter 5, Deep Learning models are trained by performing gradient descent minimization of the cost function, which requires the cost function to be "minimizable" in the first place. In mathematics we say that the cost function needs to be **convex** and **differentiable**.

One of the most commonly used cost function in Deep Learning is the **cross-entropy loss**.

Let's explore how it is calculated. We can define the cost for a single point as:

$$c_i = -y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$$

Notice that due to the binary nature of the outcome variable y , only one of the two terms is present at each time. If the label y_i is 0, then $c_i = -\log(1 - \hat{y}_i)$, if the label y_i is 1, then $c_i = -\log(\hat{y}_i)$.

Another way of thinking about this in programmable terms might be

$$c_i = \begin{cases} -\log(\hat{y}_i) & \text{for } y_i = 1 \\ -\log(1 - \hat{y}_i) & \text{for } y_i = 0 \end{cases}$$

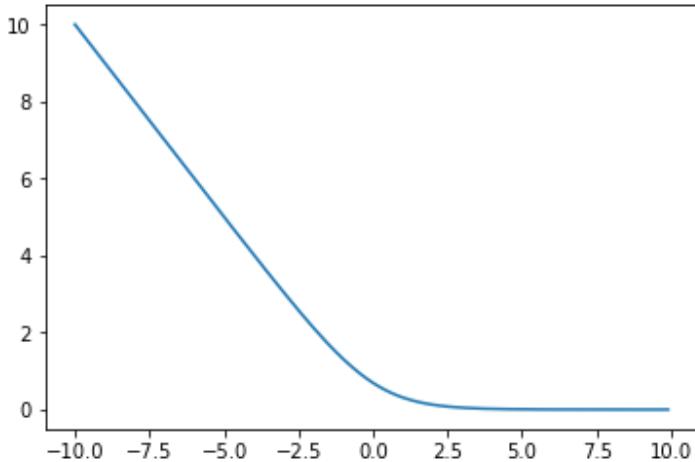
Let's look at the first term first, which only contributes to the cost when $y_i = 1$. Remember that \hat{y} contains the sigmoid function, so its negative logarithm is:

$$\log(\sigma(z)) = -\log(1) + \log(1 + e^{-z}) = \log(1 + e^{-z})$$

What this means is if z is really big, this quantity goes to zero, if z is negative, this quantity goes to infinity:

```
plt.plot(z, -np.log(sigmoid(z)))
```

```
[<matplotlib.lines.Line2D at 0x7f14906eaba8>]
```



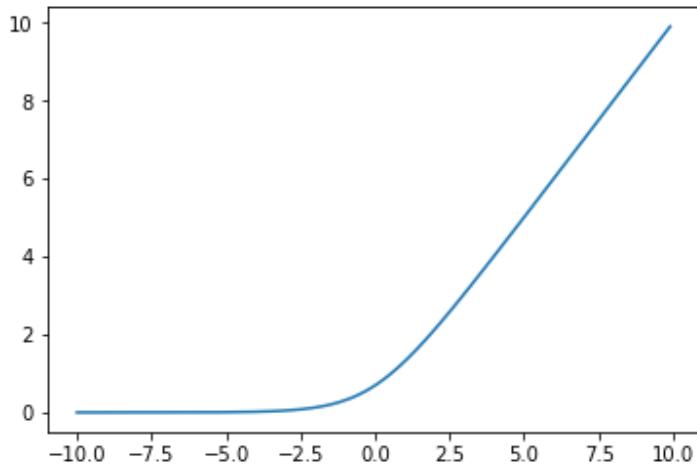
In other words, when the label is 1 ($y = 1$), our predictions should also approach 1. Since our predictions are obtained with the sigmoid, we want $\hat{y} = \sigma(z)$ to approach 1 as well. This happens for very large values of z . Therefore, the cost should be very small when z is large. On the other hand, if z is small, the sigmoid

goes to zero and our prediction is wrong. That's why the cost becomes increasingly large for negative values of z .

The same logic applies to the second term for when $y = 0$: it should push z to have negative values so that the sigmoid goes to zero and our prediction is correct in this case.

```
plt.plot(z, -np.log(1 - sigmoid(z)))
```

```
[<matplotlib.lines.Line2D at 0x7f149064f390>]
```



Now that we have defined the cost for a single point, we can define the average cost as:

$$c = \frac{1}{N} \sum_i c_i$$

This is the *average cross-entropy* or *log loss*.

Now that we have defined hypothesis and cost for the logistic regression case, we can go ahead and look for the best parameters that minimize the cost, very much in the same way as we did for the linear regression case.

Logistic regression in Keras

First let's define a model in Keras. As we have seen above, `Dense(1, input_shape=(1,))` implements a linear function with one input and one output. The only change we need to perform is to add a *sigmoid*

function that takes the linear variable and maps it to the interval [0, 1]. In a way, it's as if we were "wrapping" the Dense layer with the sigmoid function.

Let's first create a model like we did for the linear regression:

```
model = Sequential()  
model.add(Dense(1, input_dim=1))
```

We can add the activation as a layer:

```
from keras.layers import Activation
```

```
model.add(Activation('sigmoid'))
```

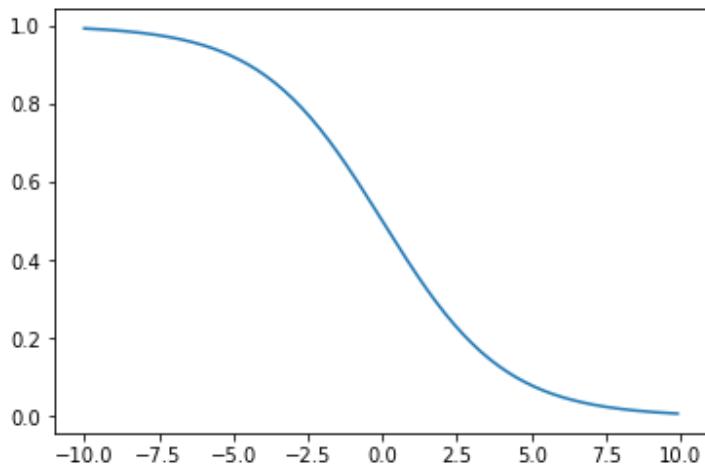
```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 1)	2
activation_1 (Activation)	(None, 1)	0
Total params:	2	
Trainable params:	2	
...		
Trainable params:	2	

As you can see the model has two parameters, a weight and a bias, and it has a sigmoid activation function as a second layer. We can convince ourselves that it's a sigmoid by using the model to predict values for a few z values:

```
plt.plot(z, model.predict(z))
```

```
[<matplotlib.lines.Line2D at 0x7f14901a8438>]
```



Also notice that the weights in the model are initialized randomly, so your sigmoid may look different from the one in the figure above.

TIP: Keras allows a more compact model specification by including the activation function in the Dense layer definition. We can define the same model above by:

```
model.add(Dense(1, input_dim=1, activation='sigmoid'))
```

The next step is to compile the model like we did before to specify the cost function and the optimizer. Keras offers [several cost functions for classification](#). The cross-entropy for the binary classification case is called `binary_crossentropy` so we will use this one now:

```
model.compile(SGD(lr=0.5), 'binary_crossentropy', metrics=['accuracy'])
```

Accuracy

Notice that this time we also included an additional metric at compile time: `accuracy`. [Accuracy](#) is one of the possible scores we can use to judge the quality of a classification model. It tells us what fraction of samples are predicted in the correct class, so for example an accuracy of `80%` or `0.8` means that 80 samples out of 100 are predicted correctly.

Let's train this new model on our data.

```
model.fit(X, y, epochs=25)
```

```
Epoch 1/25
100/100 [=====] - 0s 2ms/step - loss: 0.8045 - acc: 0.4600
Epoch 2/25
100/100 [=====] - 0s 92us/step - loss: 0.6307 - acc: 0.5300
Epoch 3/25
100/100 [=====] - 0s 96us/step - loss: 0.5963 - acc: 0.6400
Epoch 4/25
100/100 [=====] - 0s 94us/step - loss: 0.5493 - acc: 0.7900
Epoch 5/25
...
Epoch 5/25
```

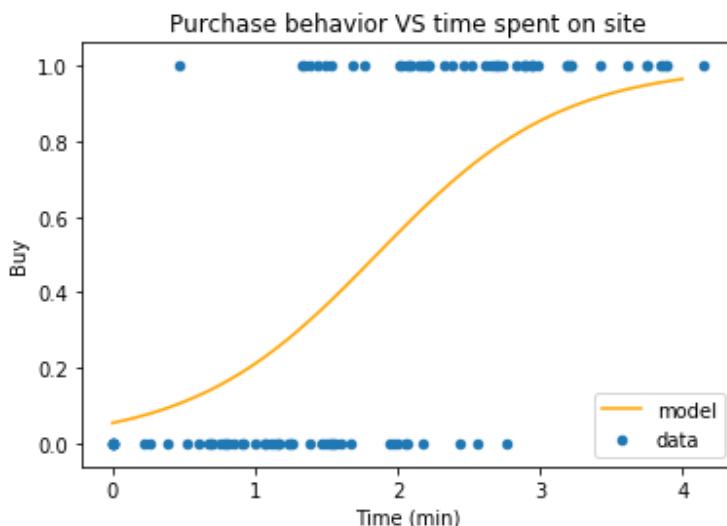
```
<keras.callbacks.History at 0x7f1490676400>
```

The model seems to have converged because the loss does not seem to improve in the last epochs. Let's see what the predictions look like:

```
ax = df.plot(kind='scatter', x='Time (min)', y='Buy',
              title='Purchase behavior VS time spent on site')

temp = np.linspace(0, 4)
ax.plot(temp, model.predict(temp), color='orange')
plt.legend(['model', 'data'])
```

```
<matplotlib.legend.Legend at 0x7f14780eb0b8>
```



Great! The two parameters in our logistic regression have been tuned to best reproduce our data.

Notice that the logistic regression model predicts a probability. If we want to convert this to a binary prediction we need to set a threshold. For example we could say that all points predicted to be 1 with $p > 0.5$ are set to 1 and the others are set to 0.

```
y_pred = model.predict(X)
```

```
y_class_pred = y_pred > 0.5
```

With this definition we can calculate the accuracy of our model as the number of correct predictions over the total number of points. Scikit-learn offers a ready to use function for this behavior called `accuracy_score` :

```
from sklearn.metrics import accuracy_score
```

```
print("Accuracy score: {:.3f}".format(accuracy_score(y, y_class_pred)))
```

```
Accuracy score: 0.820
```

Train/Test split

We can repeat the above steps using train/test split. Remember, we're aiming for similar accuracies in the train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

We need to reset the model, or it will retain the previous training. How do we do that? Our model only has 2 parameters, w and b , so we can just reset these two parameters to zero.

```
params = model.get_weights()
```

```
params
```

```
[array([[1.543166]]), array([-2.8641326]), dtype=float32)]
```

```
params = [np.zeros(w.shape) for w in params]
```

```
params
```

```
[array([[0.]]), array([0.])]
```

```
model.set_weights(params)
```

Let's check that the model is now predicting garbage:

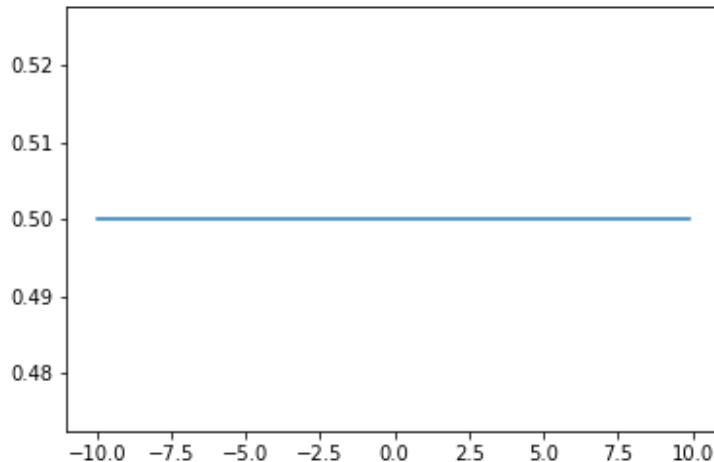
```
acc = accuracy_score(y, model.predict(X) > 0.5)
print("The accuracy score is {:.3f}".format(acc))
```

```
The accuracy score is 0.500
```

And in fact the model is now a straight line at 0.5:

```
plt.plot(z, model.predict(z))
```

```
[<matplotlib.lines.Line2D at 0x7f145cac2208>]
```



Let's re-train it on the training data only

```
model.fit(X_train, y_train, epochs=25)
```

```
Epoch 1/25
80/80 [=====] - 0s 100us/step - loss: 0.6547 - acc: 0.5250
Epoch 2/25
80/80 [=====] - 0s 94us/step - loss: 0.6059 - acc: 0.6250
Epoch 3/25
80/80 [=====] - 0s 93us/step - loss: 0.5953 - acc: 0.6500
Epoch 4/25
80/80 [=====] - 0s 94us/step - loss: 0.5699 - acc: 0.6250
Epoch 5/25
...
Epoch 5/25
```

```
<keras.callbacks.History at 0x7f14780eb9e8>
```

And let's check the accuracy score on training and test sets:

```
acc = accuracy_score(y_train, model.predict(X_train) > 0.5)
print("Train accuracy score {:.3f}".format(acc))

acc = accuracy_score(y_test, model.predict(X_test) > 0.5)
print("Test accuracy score {:.3f}".format(acc))
```

```
Train accuracy score 0.762
Test accuracy score 0.950
```

So, in this case the model is performing as well on the test set as on the training set. Good!

OVERFITTING

We are advancing quickly! This table recaps what we have learned so far:

Target Variable	Method	Hypothesis	Cost Function
Continuous	Linear Regression	$\hat{y} = X \cdot w + b$	Mean squared Error
Discrete	Logistic Regression	$\hat{y} = \text{sigmoid}(X \cdot w + b)$	Cross Entropy Error

Notice we have extended the models to datasets with multiple features using the vector notation:

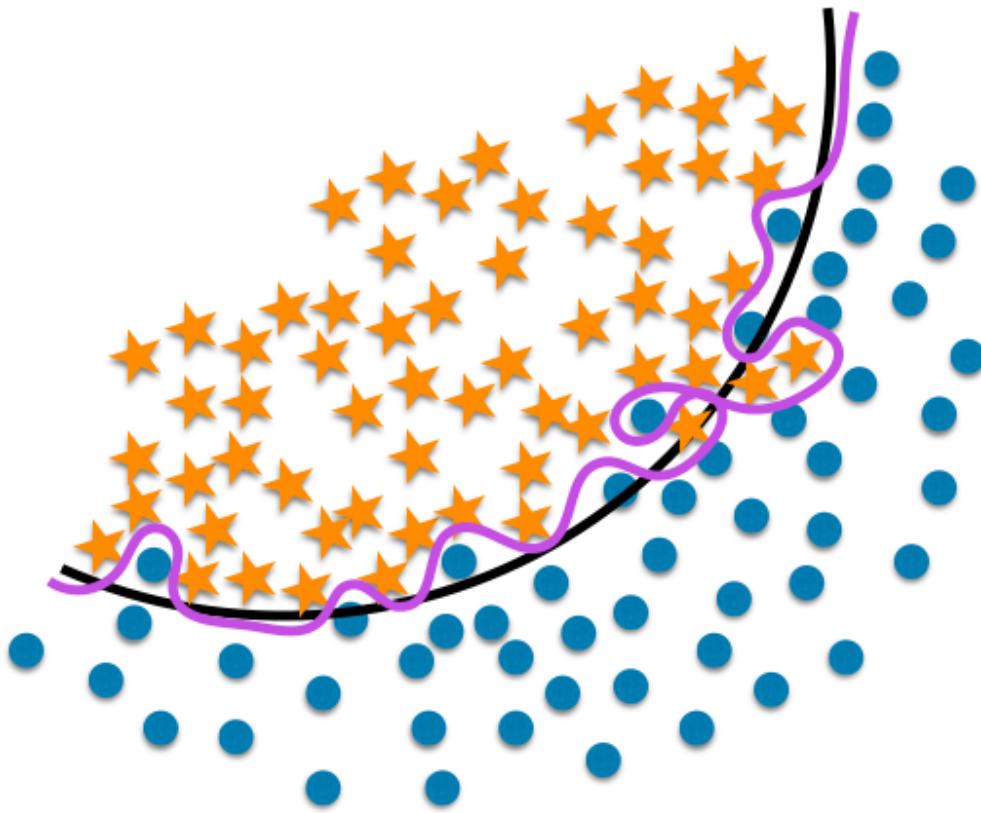
$$X \cdot w = x_{j0}w_0 + x_{j1}w_1 + x_{j2}w_2 + \dots = \sum_i x_{ji}w_i \text{ for each data point } j$$

In this case w is a weight vector of size M , where M is the number of features, while X is a matrix of size $N \times M$, where N is the number of records in our dataset.

We have also learned to split our data in two parts: a training set and a test set.

Now let's talk about one thing to watch out for: **overfitting!**

Overfitting happens when our model learns the probability distribution of the training set too well and is not able to generalize to the test set with the same performance. Think of this as learning things by heart without really understanding them, in a new situation you will be lost and probably under-perform.



A very simple way to check for overfitting is to compare the cost and the performance metrics of the training and test set. For example, let's say we are performing a classification and we measure the number of correct prediction, aka the accuracy, to be 99% for the training set and only 85% for the test set. This means our model is not performing as well on the test set and we are therefore overfitting.

It is going to be very hard to overfit with a simple model with only one parameter, but as the number of parameters increases, the likelihood of overfitting increases as well. We'll need to watch out for our model *overfitting* the dataset.

How to avoid overfitting

There are several actions we can take to minimize the risk of overfitting.

The first simple check is to make sure that our train/test split is performed correctly and both the train and test sets are representative of the whole population of features and labels. Common errors include:

- Not preserving the ratio of labels.
- Not randomly sampling the dataset.

- Using a too small test set.
- Using a too small train set.

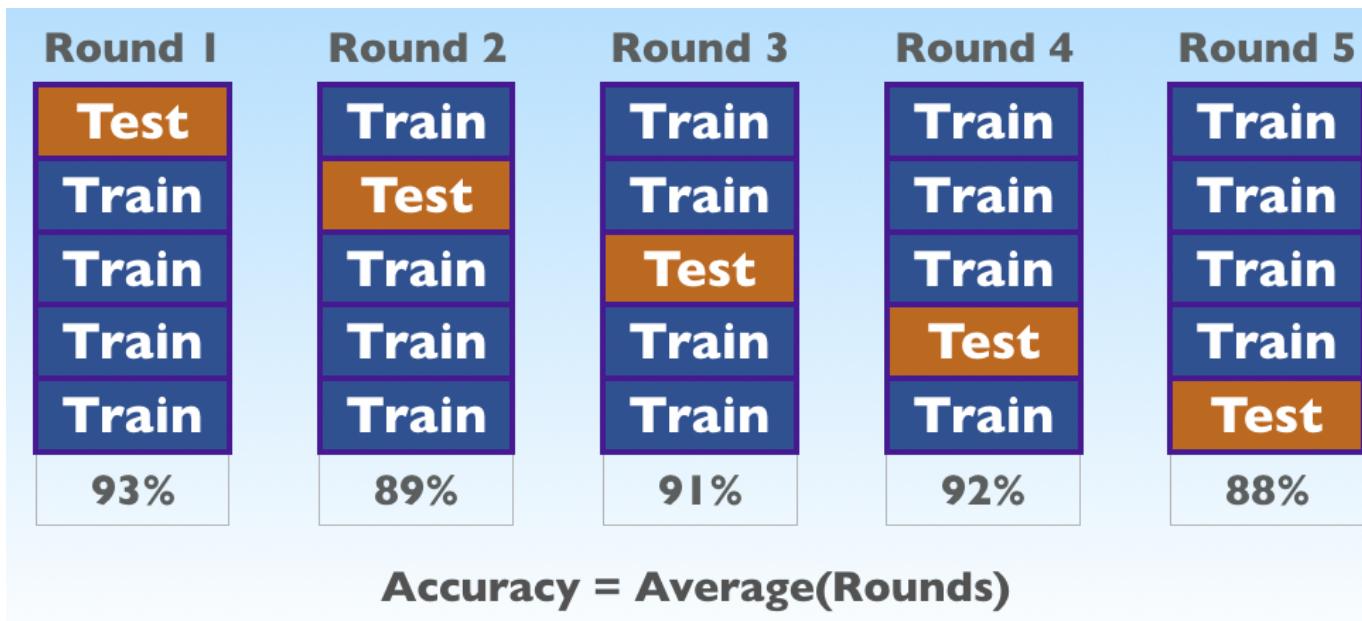
If the train/test split seems correct, it could be the case that our model has too much "freedom" and therefore learns by heart the training set. This is usually the case if the number of parameters in the model is comparable or greater than the number of data points in the training set. In order to mitigate this we can either reduce the complexity of the model, or use regularization, as we shall see later on in the book.

CROSS VALIDATION

Is train/test split the most efficient way to use our dataset? Even if we took great care in randomly splitting our data, that's only one of many possible ways to perform a split. What if we performed several different train/test splits, checked the test score in each of them and finally averaged the scores?

Not only we would have a more precise estimation of the true accuracy, but also we could calculate the standard deviation of the scores and therefore know the error on the accuracy itself.

This procedure is called cross-validation. There are many ways to perform cross-validation. The most common is called **K-fold cross-validation**.



In K -fold cross validation the whole dataset is split into K equally sized random subsets. Then, each of the K subsets gets to play the role of test set, while the others are aggregated back to form a training set. In this way, we obtain K estimations of the model score, each calculated from a test set that does not overlap with any of the other test sets.

Not only do we get a better estimate of the validation score, including its standard deviation, but we also used each datapoint more efficiently, since each data point gets to play the role of both train and test.

These advantages do not come for free, in fact we had to train the model K times, which takes longer and consumes more resources than training it just one time. On the other hand, we can parallelize the training over each fold, either by distributing them across processes or across different machines.

Scikit-Learn offers cross-validation out of the box, but we'll have to wrap our model in a way that can be understood by Scikit-Learn. This is easy to do using a wrapper class called `KerasClassifier`.

```
from keras.wrappers.scikit_learn import KerasClassifier
```

```
def build_logistic_regression():
    model = Sequential()
    model.add(Dense(1, input_dim=1, activation='sigmoid'))
    model.compile(SGD(lr=0.5), 'binary_crossentropy', metrics=['accuracy'])
    return model
```

```
model = KerasClassifier(build_fn=build_logistic_regression,
                       epochs=25, verbose=0)
```

We've just redefined the same model, but in a format that is compatible with Scikit-Learn. Let's calculate the cross validation score on a 3-fold (it means $K = 3$) cross validation:

```
from sklearn.model_selection import cross_val_score, KFold
```

```
cv = KFold(3, shuffle=True)
scores = cross_val_score(model, X, y, cv=cv)
```

```
scores
```

```
array([0.76470588, 0.75757576, 0.75757576])
```

The cross validation produced 3 scores, 1 for each fold. We can average them and take their standard deviation as a better estimation of our accuracy:

```
m = scores.mean()
s = scores.std()
print("Cross validation accuracy: {:.4f} ± {:.4f}".format(m, s))
```

```
Cross validation accuracy: 0.7600 ± 0.0034
```

There are also other ways to perform a cross validation. Here we mention a few.

Stratified K-fold is similar to K-fold but it makes sure that the proportions of labels is preserved in the folds. For example, if we are performing a binary classification and 40% of the data is labeled True and 60% is labeled False, each of the folds will also contain 40% True labels and 60% False labels.

We can also perform cross validation by randomly selecting a test set of fixed size multiple times. In this case, we do not need make sure that the test sets are disjoint and they will overlap in some points.

Finally it is worth mentioning **Leave-One-Label-Out** cross validation, or **LOLO**. LOLO is useful when our data is organized in subgroups. For example, imagine we are building a model to recognize gestures from phone accelerometer data. Our training dataset probably contains multiple recordings of different gestures from different users. The labels we are trying to predict are the gestures.

By performing a simple cross validation both our training and test sets would leave us with sets which contain recordings from all users. If we train the model in this way, we could very well end up with a good test score, but we would have no idea about how the model would perform if a **new user** performed the same gestures. In other words, the model could be overfitting over each user, and we would have no way of knowing it.

In this case, it is better to split the data on the users, using the data from some of them as training, while testing on the data from some other users. If the test score is good in this case, we can be fairly sure that the model will perform well with a new user.

CONFUSION MATRIX

Is **accuracy** the best way to check the performance of our model? It surely tells us how well we are doing overall, but it doesn't give us any insight on the kind of errors the model is doing. Let's see how we can do better.

In the problem we just introduced, we are estimating the purchase probability from the time spent on a page. This is a binary classification and we can be either right or wrong in the four ways represented here:

	<i>Test Negative</i>	<i>Test Positive</i>
<i>Condition Negative</i>	TRUE NEGATIVE	FALSE POSITIVE (Type I error)
<i>Condition Positive</i>	FALSE NEGATIVE (Type II error)	TRUE POSITIVE

This table is called **confusion matrix** and it gives a better view of what's being predicted correctly and what's not.

Let's look at the four cases one at a time. We could be right in predicting the purchase or right in predicting the absence of a purchase. These are the **True Positives** and **True Negatives**. Summed together they amount to the number of correct predictions we formulated. If we divide this number by the total number of data points, we obtain the **Accuracy** of the model. In other words, the accuracy is the overall ratio of correct predictions:

$$\text{Acc} = \frac{(TP+TN)}{\text{All}}$$

On the other hand our model could be wrong in two ways.

1. It could predict buy, when the person is actually not buying: this is a **False Positive**.
2. It could predict not buy when the person is actually buying: this is a **False Negative**.

Let's use Scikit-Learn to calculate the confusion matrix of our data:

```
from sklearn.metrics import confusion_matrix
```

We define a short helper function to add column and row labels for nice display:

```
def pretty_confusion_matrix(y_true, y_pred, labels=["False", "True"]):
    cm = confusion_matrix(y_true, y_pred)
    pred_labels = ['Predicted ' + l for l in labels]
    df = pd.DataFrame(cm, index=labels, columns=pred_labels)
    return df
```

```
pretty_confusion_matrix(y, y_class_pred, ['Not Buy', 'Buy'])
```

	Predicted Not Buy	Predicted Buy
Not Buy	41	9
Buy	9	41

Let's stop here for a second. Let's say that, if the model was predicting True, the user is offered to buy an additional product at a discount. On which side would you rather the model be wrong? Would you like the model to offer a discount to users with no intention of buying (False Positive) or would you rather it not offer a discounted additional item to users who intend to buy (False Negative)?

What if, instead of predicting the purchase behavior from time spent on a page we were predicting the likelihood to have cancer based on the value of a blood screening exam? Would you rather have a False Positive or a False Negative in that case?

Most people would prefer a False Positive, and do an additional screening to make sure of the result, rather than go home feeling safe and healthy while they are actually not. Would that be your choice too?

What if you were an (evil) health insurance company instead? Would you still choose to optimize the model in the same way? A False Positive would be an additional cost to you, because the patient would go on to see

a specialist. Would you rather minimize False Positives in this case?

As you can see, there is no one correct answer. Different stakeholders will make opposite choices. This is to say that the data scientist is not a neutral observer of a Machine Learning process. The choices he/she makes, fundamentally determine the outcome of the training!

False Positives and False Negatives are usually expressed in terms of two sister quantities: Precision and Recall. Here they are:

Precision

We define precision as the ratio of True Positives to the total number of positive tests:

$$\text{Precision} = \frac{(TP)}{TP+FP}$$

Precision P will tend towards 1 when the number of False Positives goes to zero, i.e. when we do not create any false alerts and are thus, "precise". Here on every positive case we are correct.

Recall

On the other hand, recall is defined as the ratio of True Positives to the total number of actually positive cases:

$$\text{Recall} = \frac{(TP)}{TP+FN}$$

Recall R will tend towards 1 when the number of False Negatives goes to zero, i.e. when we do not miss many of the positive cases or we "recall" all of them.

F1 Score

Finally, we can combine the two in what's called F1-score:

$$F1 = 2 \cdot \frac{PR}{P+R}$$

$F1$ will be close to 1 if both precision and recall are close to 1, while it will go to zero if either of them is low. In this sense the F1 score is a good way to make sure that both precision and recall are high.

The $F1$ score is a [harmonic mean](#) of precision and recall. The harmonic mean is an average for ratios. There are also other F-scores that weigh one of precision or recall more or less heavily, called F-beta scores. You can read about them on [Wikipedia](#) and on [Scikit-Learn doc](#).

Let's evaluate these scores for our data:

```
from sklearn.metrics import precision_score, recall_score, f1_score
```

```
print("Precision:\t{:0.3f}".format(precision_score(y, y_class_pred)))
print("Recall: \t{:0.3f}".format(recall_score(y, y_class_pred)))
print("F1 Score:\t{:0.3f}".format(f1_score(y, y_class_pred)))
```

```
Precision:      0.820
Recall:         0.820
F1 Score:       0.820
```

Scikit-Learn offers a handy `classification_report` function that combines all these:

```
from sklearn.metrics import classification_report
```

```
print(classification_report(y, y_class_pred))
```

	precision	recall	f1-score	support
0	0.82	0.82	0.82	50
1	0.82	0.82	0.82	50
avg / total	0.82	0.82	0.82	100

support here means how many point were present in each class.

While these definitions hold true only for the binary classification case, we can still extend the confusion matrix to the case where there are more than 2 classes.

	<i>Prediction A</i>	<i>Prediction B</i>	<i>Prediction C</i>	<i>Prediction D</i>
<i>Class A</i>	84			
<i>Class B</i>		69	3	
<i>Class C</i>	2		78	
<i>Class D</i>				91

In this case the element i,j of the matrix will tell us how many datapoints in class i have been predicted to be in class j . This is very powerful to see if any of the classes are being confused. If so we can isolate the data being misclassified and try to understand why.

FEATURE PREPROCESSING

Categorical Features

Sometimes input data will be categorical, i.e. the feature values will be discrete classes instead of continuous numbers. For example, in the weight/height dataset above, there's a 3rd column called `Gender` which can either be `Male` or `Female`. How can we convert this categorical data to numbers that can be consumed by our model?

There are several ways to do it, the most common being **One-Hot** or **Dummy** encoding. In Dummy encoding, we substitute the categorical column with a set of boolean columns, one for each category present in the column. In the Male/Female example above, we would replace the `Gender` column with 2 columns called `Gender_Male` and `Gender_Female` that would have binary values. Pandas offers a quick way to do that:

```
df = pd.read_csv('..../data/weight-height.csv')
df.head()
```

	Gender	Height	Weight
0	Male	73.847017	241.893563
1	Male	68.781904	162.310473
2	Male	74.110105	212.740856
3	Male	71.730978	220.042470
4	Male	69.881796	206.349801

Here's how to create the dummy columns:

```
pd.get_dummies(df['Gender'], prefix='Gender').head()
```

	Gender_Female	Gender_Male
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

In this particular case, we only need one of the two columns, since we only have 2 classes, but if we had 3 or more categories, then we would need to pass all the dummy columns to our model.

There are other ways to encode categorical information, including **index encoding**, **hashing trick** and **embeddings**. We will learn more of these later in the book.

Feature Transformations

As we will see in the exercises, Neural Network models are quite sensitive to the absolute size of the input features. This means that passing in features with very large or very small values will not help our model converge to a solution. An easy way to overcome this problem is to normalize the features to a number near 1.

Here are a few methods we can use to transform our features.

1) Rescale with fixed factor

We could change the unit of measurement. For example, in the Humans example we could rescale the height by 12 (go from inches to feet) and the weight by 100 (go from pounds to 100 pounds):

```
df.head()
```

	Gender	Height	Weight
0	Male	73.847017	241.893563

	Gender	Height	Weight
1	Male	68.781904	162.310473
2	Male	74.110105	212.740856
3	Male	71.730978	220.042470
4	Male	69.881796	206.349801

```
df['Height (feet)'] = df['Height']/12.0
df['Weight (100 lbs)'] = df['Weight']/100.0
```

```
df.describe().round(2)
```

	Height	Weight	Height (feet)	Weight (100 lbs)
count	10000.00	10000.00	10000.00	10000.00
mean	66.37	161.44	5.53	1.61
std	3.85	32.11	0.32	0.32
min	54.26	64.70	4.52	0.65
25%	63.51	135.82	5.29	1.36
50%	66.32	161.21	5.53	1.61
75%	69.17	187.17	5.76	1.87
max	79.00	269.99	6.58	2.70

As you can see our new features have values that are close to 1 in order of magnitude, which is good enough.

2) MinMax normalization

A second way to normalize features is to take the minimum value and the maximum value and rescale all values to the interval (0,1). This can be done using the `MinMaxScaler` provided by `sklearn` like so:

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
df['Weight_mms'] = mms.fit_transform(df[['Weight']])
df['Height_mms'] = mms.fit_transform(df[['Height']])
df.describe().round(2)
```

	Height	Weight	Height (feet)	Weight (100 lbs)	Weight_mms	Height_mms
count	100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
mean	66.37	161.44	5.53	1.61	0.47	0.49
std	3.85	32.11	0.32	0.32	0.16	0.16
min	54.26	64.70	4.52	0.65	0.00	0.00
25%	63.51	135.82	5.29	1.36	0.35	0.37
50%	66.32	161.21	5.53	1.61	0.47	0.49
75%	69.17	187.17	5.76	1.87	0.60	0.60
max	79.00	269.99	6.58	2.70	1.00	1.00

Our new features have a maximum value of 1 and a minimum value of 0, exactly as we wanted them.

3) Standard normalization

A third way to normalize large or small features is to subtract the mean and divide by the standard deviation.

```
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
df['Weight_ss'] = ss.fit_transform(df[['Weight']])
```

```
df['Height_ss'] = ss.fit_transform(df[['Height']])
df.describe().round(2)
```

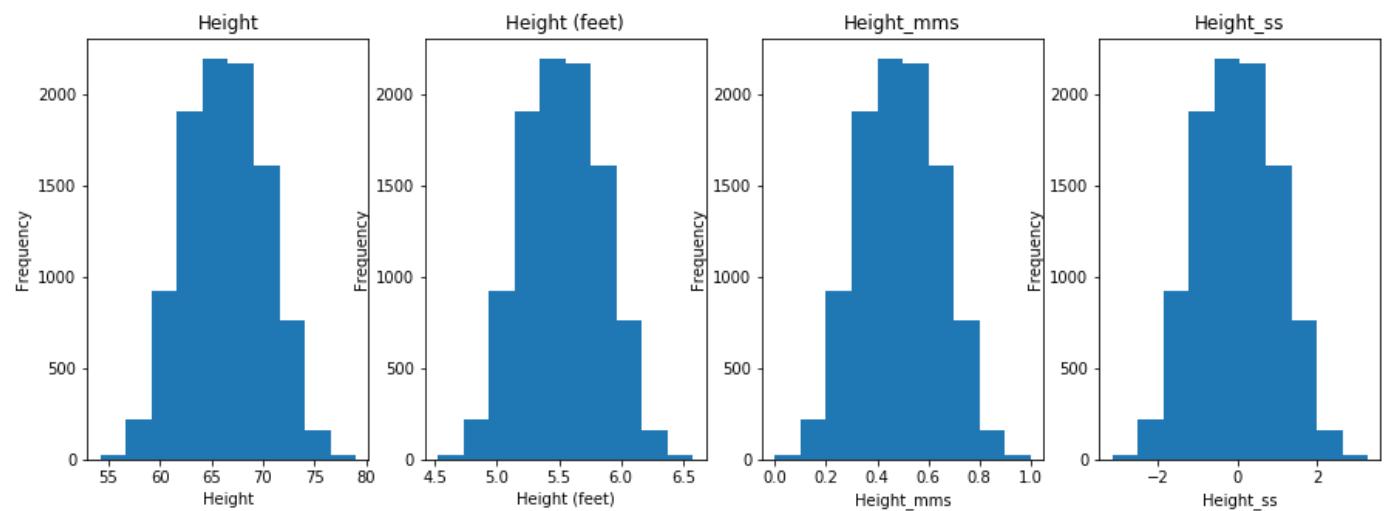
	Height	Weight	Height (feet)	Weight (100 lbs)	Weight_mms	Height_mms	Weigh
count	100000.00	100000.00	100000.00	100000.00	100000.00	100000.00	10000.
mean	66.37	161.44	5.53	1.61	0.47	0.49	0.00
std	3.85	32.11	0.32	0.32	0.16	0.16	1.00
min	54.26	64.70	4.52	0.65	0.00	0.00	-3.01
25%	63.51	135.82	5.29	1.36	0.35	0.37	-0.80
50%	66.32	161.21	5.53	1.61	0.47	0.49	-0.01
75%	69.17	187.17	5.76	1.87	0.60	0.60	0.80
max	79.00	269.99	6.58	2.70	1.00	1.00	3.38

After standard normalization, our new features have approximately zero mean and standard deviation of 1. This is good in a linear model, because each feature is multiplied by a weight that the model has to find. Since the weights are initialized to have values near 1, if the feature had a very large or very small scale, the model could have to adjust the value of the weight enormously, just to account for the different scale. It is therefore good practice to normalize our features before giving them to a Neural Network.

Note that we have just rescaled the units of our features, but their distribution is the same:

```
plt.figure(figsize=(15, 5))

for i, feature in enumerate(['Height', 'Height (feet)',
                            'Height_mms', 'Height_ss']):
    plt.subplot(1, 4, i+1)
    df[feature].plot(kind='hist', title=feature)
    plt.xlabel(feature)
```



Alright! Time has come to apply what you've learned with some exercises.

EXERCISES

Exercise 1

You've just been hired at a real estate investment firm and they would like you to build a model for pricing houses. You are given a dataset that contains data for house prices and a few features like number of bedrooms, size in square feet and age of the house. Let's see if you can build a model that is able to predict the price. In this exercise we extend what we have learned about linear regression to a dataset with more than one feature. Here are the steps to complete it:

1. load the dataset `../data/housing-data.csv`
- plot the histograms for each feature
- create 2 variables called `x` and `y` : `x` shall be a matrix with 3 columns (`sqft,bdrms,age`) and `y` shall be a vector with one column (`price`)
- create a linear regression model in Keras with the appropriate number of inputs and output
- split the data into train and test with a 20% test size
- train the model on the training set and check its accuracy on training and test set
- how's your model doing? Is the loss growing smaller?
- try to improve your model with these experiments:
 - normalize the input features with one of the rescaling techniques mentioned above
 - use a different value for the learning rate of your model
 - use a different optimizer
- once you're satisfied with training, check the R^2 on the test set

Exercise 2

Your boss was extremely happy with your work on the housing price prediction model and decided to entrust you with a more challenging task. They've seen a lot of people leave the company recently and they would like to understand why that's happening. They have collected historical data on employees and they would like you to build a model that is able to predict which employee will leave next. They would like a model that is better than random guessing. They also prefer false negatives than false positives, in this first phase. Fields in the dataset include:

- Employee satisfaction level

- Last evaluation
- Number of projects
- Average monthly hours
- Time spent at the company
- Whether they have had a work accident
- Whether they have had a promotion in the last 5 years
- Department
- Salary
- Whether the employee has left

Your goal is to predict the binary outcome variable `left` using the rest of the data. Since the outcome is binary, this is a classification problem. Here are some things you may want to try out:

1. load the dataset at `../data/HR_comma_sep.csv`, inspect it with `.head()` , `.info()` and `.describe()` .
 - Establish a benchmark: what would be your accuracy score if you predicted everyone stay?
 - Check if any feature needs rescaling. You may plot a histogram of the feature to decide which rescaling method is more appropriate.
 - convert the categorical features into binary dummy columns. You will then have to combine them with the numerical features using `pd.concat` .
 - do the usual train/test split with a 20% test size
 - play around with learning rate and optimizer
 - check the confusion matrix, precision and recall
 - check if you still get the same results if you use a 5-Fold cross validation on all the data
 - Is the model good enough for your boss?

As you will see in this exercise, this logistic regression model is not good enough to help your boss. In the next chapter we will learn how to go beyond linear models.

This dataset comes from <https://www.kaggle.com/ludobenistant/hr-analytics/> and is released under CC BY-SA 4.0 License.

Chapter 4: Deep Learning

This chapter is about Deep Learning and it will walk you through a few simple examples that generalize how we approach regression and classification problems.

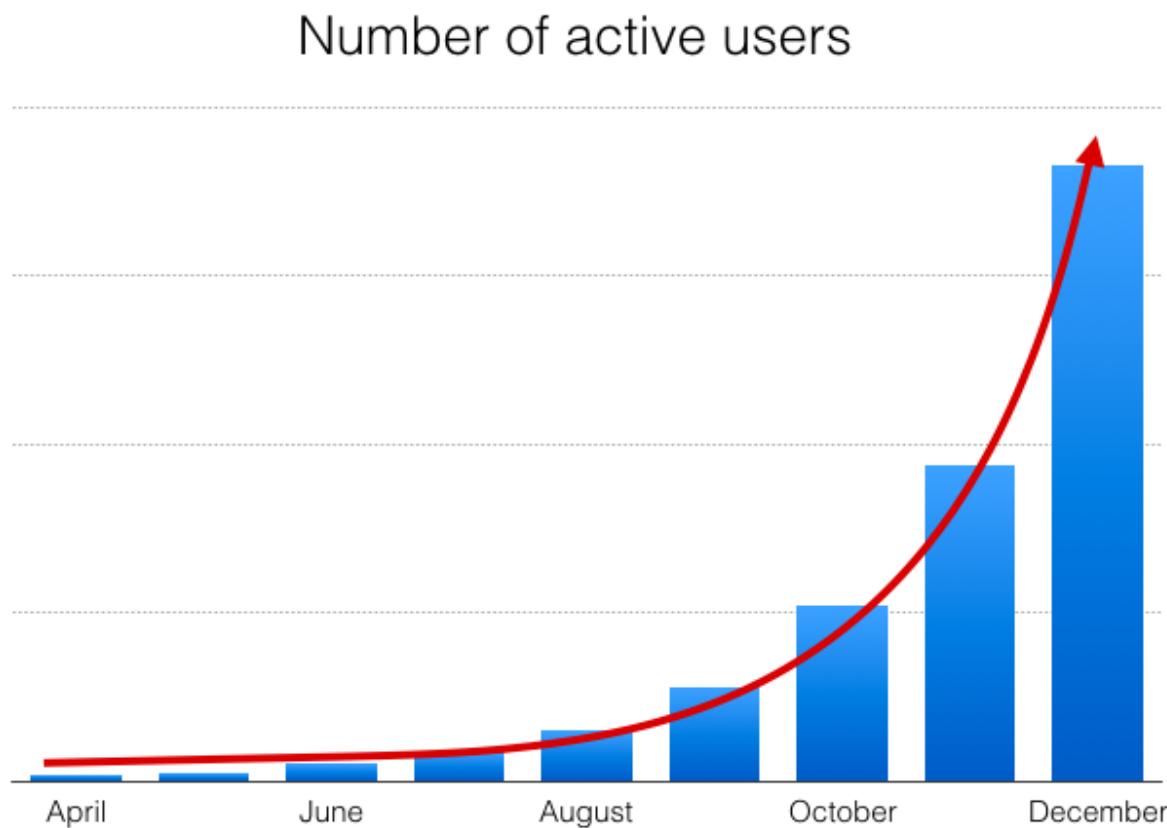
BEYOND LINEAR MODELS

In the previous chapter we encountered two techniques to solve supervised learning problems: **linear regression** and **logistic regression**.

These two techniques share many characteristics. For instance, both formulate a hypothesis about the link between features and target, both require a cost function, both depend on parameters, both learn by **finding the combination of parameters that minimizes a given cost over the training set**.

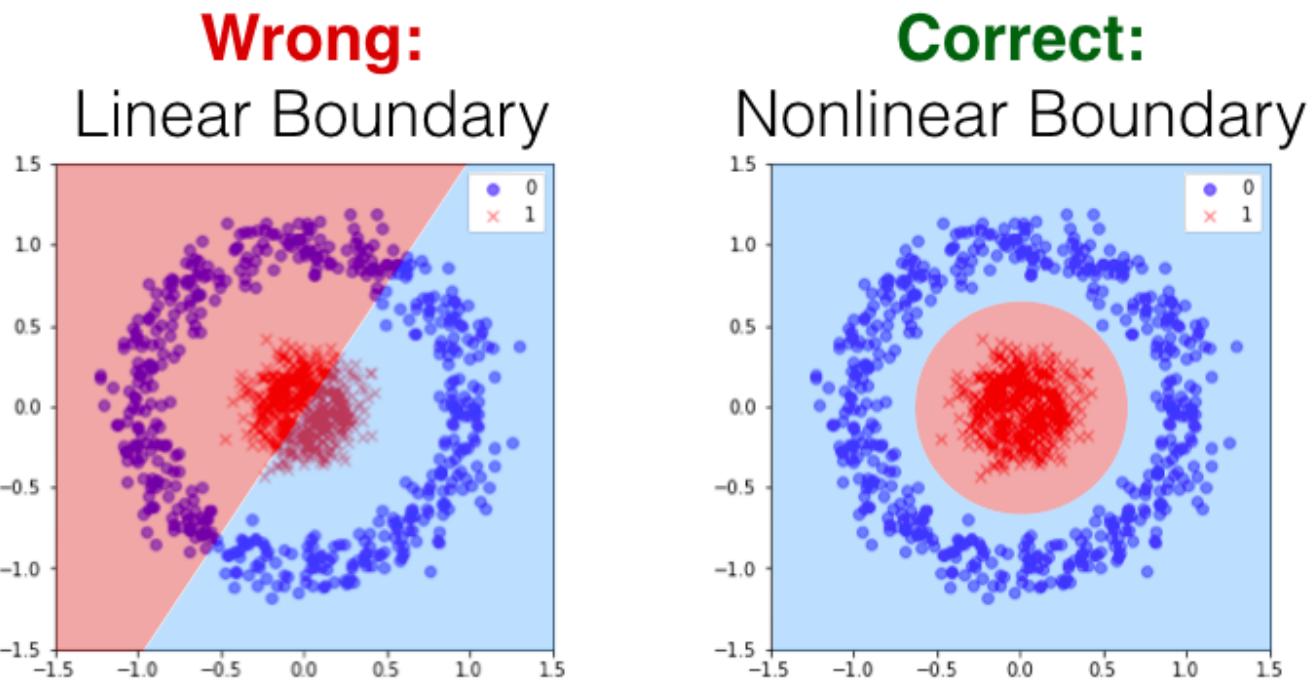
While these techniques are very powerful, they also have some limitations.

For example, linear regression doesn't work well when the relationship between features and output is not linear, i.e. when it is not described by a straight line or a flat plane. For example, think of the number of active users of a web product or a social media. If the product is successful, the number of new users added each month will grow, resulting in a nonlinear relationship between the number of users and time.



Similarly, logistic regression is incapable of separating classes that cannot be pulled apart by a flat boundary (a line in 2D, a plane in 3D, a hyperplane if we have more than 3 features). This happens all the

time, and you may hear the term "not linearly separable" to describe two classes that cannot be separated by a flat boundary. We saw an example of this in the first chapter when we tried to separate the blue dots from the red crosses.



In general, the boundary between two classes is rarely linear, especially when dealing with interesting classification problems with thousands of features. In order to extend regression and classification beyond the linear cases we need to use more complex models. Historically, computer scientists have invented many techniques to extend beyond linear models including models such as Decision Trees, Support Vector Machines, and Naive Bayes.

Deep Neural Networks bring together a unified framework to tackle all these cases: we can do linear and nonlinear regression, classification, use them to generate new data, and much more!

In this chapter, we will introduce a notation for discussing Neural Networks and rewrite linear and logistic regression using this notation. Finally, we work through stacking multiple nodes and create a deep network.

NEURAL NETWORK DIAGRAMS

Let's look at a few high-level diagrams looking at a more mathematical definition of what we're doing. If the math looks latin to you (it is), don't worry. These are just the more formal definitions of what we're doing. After this part of the chapter, we'll dive right back into code.

For the visual learners out there, this section is really helpful to "chalk-board" the algorithms we're building.

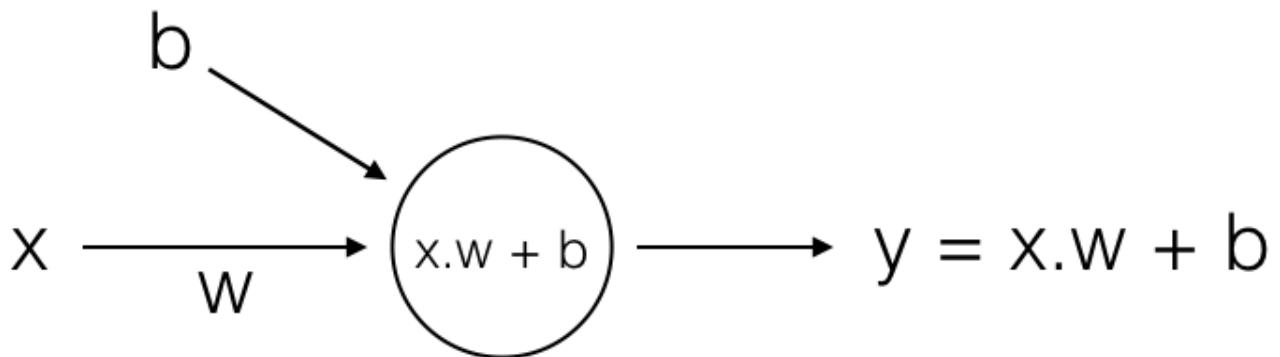
Linear regression

Let's look at linear regression. We have introduced [linear regression](#) in Chapter 3. As you may remember, it refers to problems where we try to predict a number from a set of input features. Examples are: predicting the price of a house, predicting the number of clicks a page will get or predicting the revenue a business will generate in the future.

As usual, we will refer to the inputs in the problem using the variable x and to the outputs using the variable y . So, for example, if we are trying to predict the price of a house from its size, x will be the size of the house and y will be the price. The equation of linear regression is:

$$y = x \cdot w + b$$

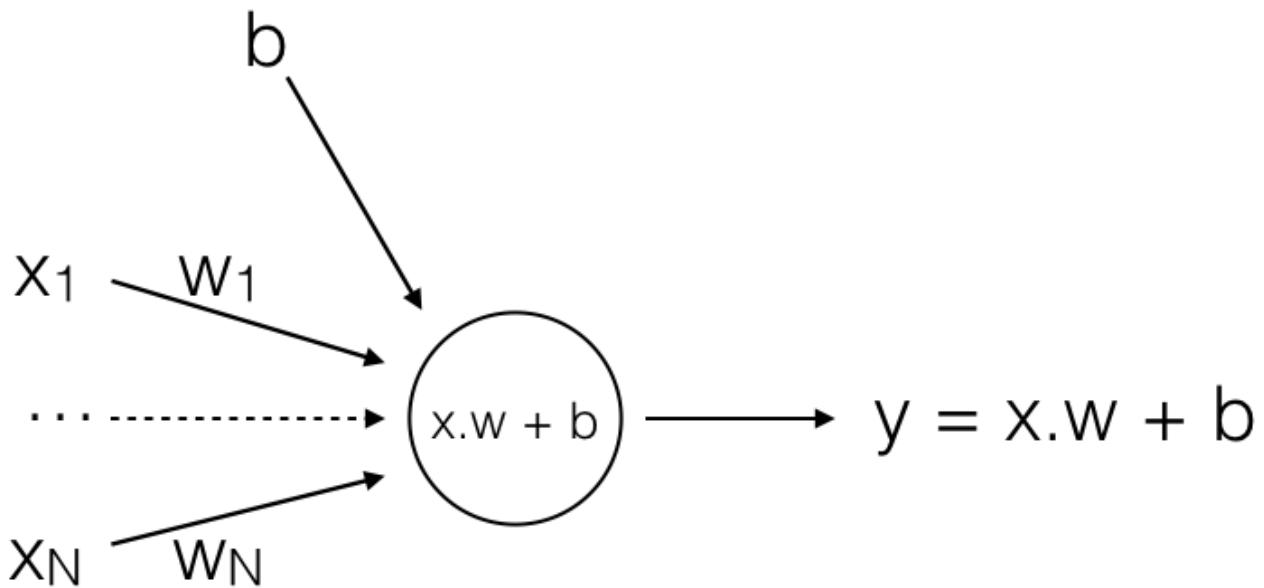
and we can represent its operation as an artificial Neural Network like this:



This network has only 1 node, the output node, represented by the circle in the diagram. This node is connected to the input feature x by a weight w . A second edge enters the node carrying the value of the parameter b , which we will call **bias**.

Fantastic! We have a simple way to represent linear operations in a graph. Let's extend the graph to multiple input features. We encountered an example of multivariate regression problem in [Exercise 1 of Chapter 3](#), where we built a model to predict the price of a house as a function of 3 inputs: the size in square feet (x_1), the number of bedrooms (x_2) and the age (x_3) of the house.

In that case we had 3 input features and the model had 3 weights (w_1 , w_2 and w_3) and 1 bias (b). We can extend our graph notation very simply to accommodate for this case:



The output node here is connected to the N inputs through N weights and it is also connected to a bias parameter. The equation is the same as before:

$$y = X \cdot w + b$$

but now X and w are arrays that contain more than one entry, multiplied using a dot product. So, what the above equation really means is:

$$y = x_1w_1 + \dots + x_Nw_N + b = X \cdot w + b$$

This is great! We can now visually represent linear regression with as many inputs as we like.

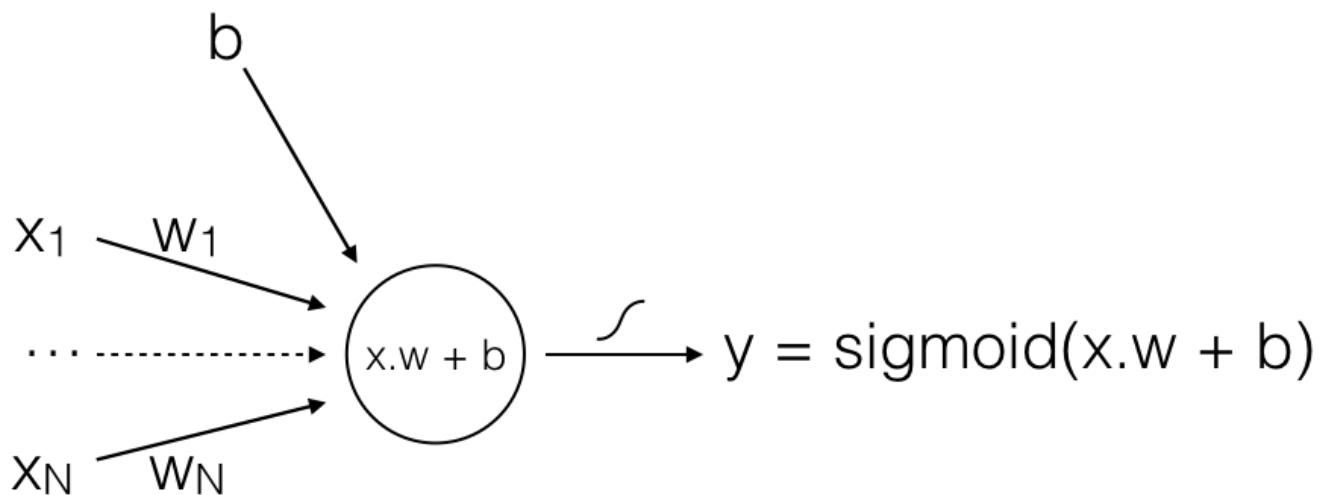
Logistic regression

Linear regression gives us a linear relationship between the inputs and outputs, but what if we want a non-binary answer instead of a linear one. For instance, what if we want a binary answer, yes/no answer? For

example, given a list of passengers on the titanic, can we predict if a specific person would survive or not?

Can you think of a way to change our equation so that we can allow for binary output?

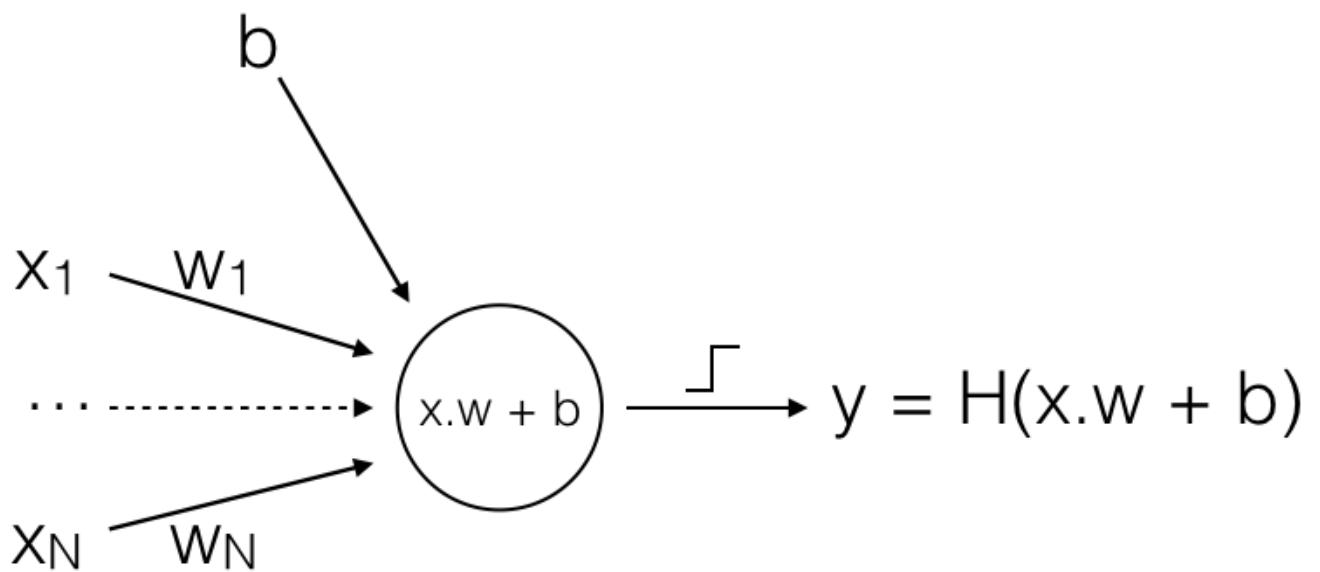
This is where we use logistic regression. Just before we output the value, we'll use the **sigmoid** function to output a binary value instead of sliding one. As you may remember from [Chapter 3](#) the sigmoid function **maps the all real values to the interval [0, 1]**. We can use the sigmoid to map the output of the node (so far linear) into the interval [0, 1]. We will interpret the result as the probability of a binary outcome.



TIP: if you need a refresher about the sigmoid you can check [Chapter 3](#) as well as this nice article on [Wikipedia](#).

Perceptron

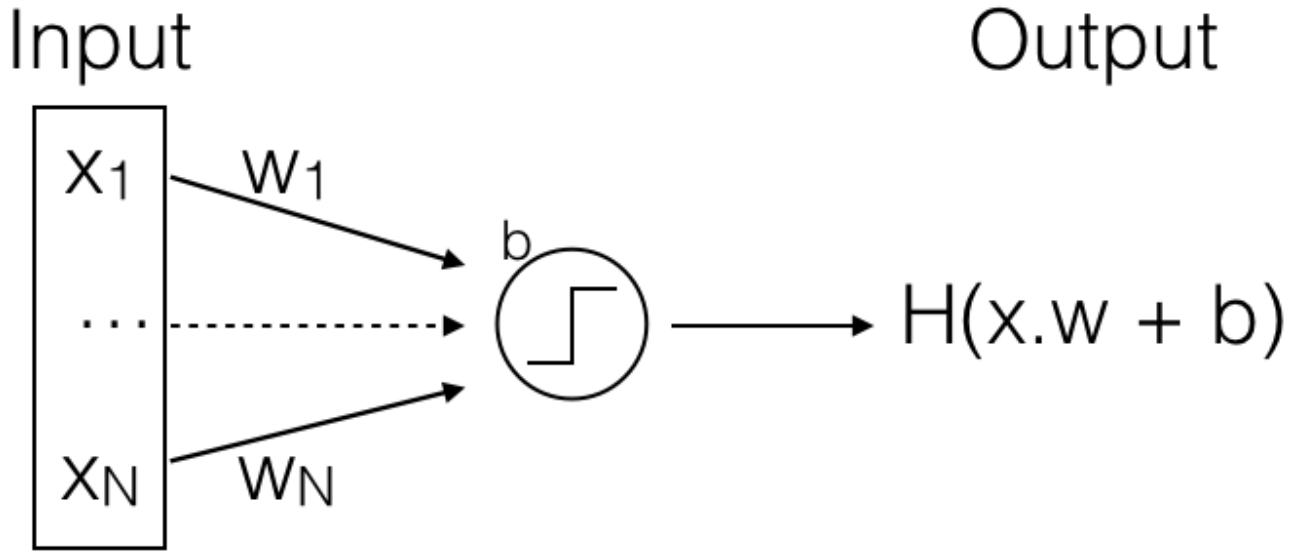
Adding a sigmoid function is just a special case of what is called an **activation function**. **Activation Function** is just a fancy name we give to the function that sits at the output of a node in a neural network. There are many different types of activation functions, and we will encounter them later in this chapter. For now, just know that they are important. For example, the first Neural Network invented, had can be described by a diagram similar to that of the Logistic Regression with just a different activation function. This network is called **Perceptron**.



The Perceptron is also a binary classifier, but instead of using a smooth *sigmoid* activation function, it uses the *step function*:

$$y = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

We could even simplify our diagram notation without losing information by including the bias and the activation symbols in the node itself, like this:



Before we move on, let's review each element in the diagram with an example. Let's say our goal is to build a model that predicts if a banknote is fake or real based on some of its properties (we'll actually do this later in the book).

First, let's define our **inputs** and **outputs**.

Our inputs are the properties of the banknote we plan to use. These could be for example: length, height, thickness, transparency, and so on. These input properties of the banknotes are also called *features*.

Our output is the prediction value, True or False, one or zero, that we hope our model to give us to tell us if the note is real or not.

The architecture of our network is represented by the graph connecting input to output. In the simple graph above our network consists of a single node performing a weighted sum of the input features.

Weights and biases are the parameters of the model. These parameters are the things we have control over (in the beginning). These are *what* the machine learns in our machine learning algorithm. They are the knobs that can be turned to change the model predictions.

During training, the network will attempt to find the best values for weights and biases, but the inputs x_1, \dots, x_n , the outputs and the network architecture, are given and cannot be changed by the model (or us, for that matter).

Now that we have established a symbolic notation that allows us to describe both linear regression and logistic regression in a very compact and visual way, let's see how we can expand the networks.

Deeper Networks

The above simple networks take multiple inputs and calculate each of their outputs as a weighted sum of the inputs plus a few other things to define a classification model (to make sure numbers make sense -- yes, we can do that). The other *things* we add to each of the inputs of our model is a fixed *bias* (usually just some small number that makes sense the input isn't zero) and an optional nonlinear activation function for the classification models.

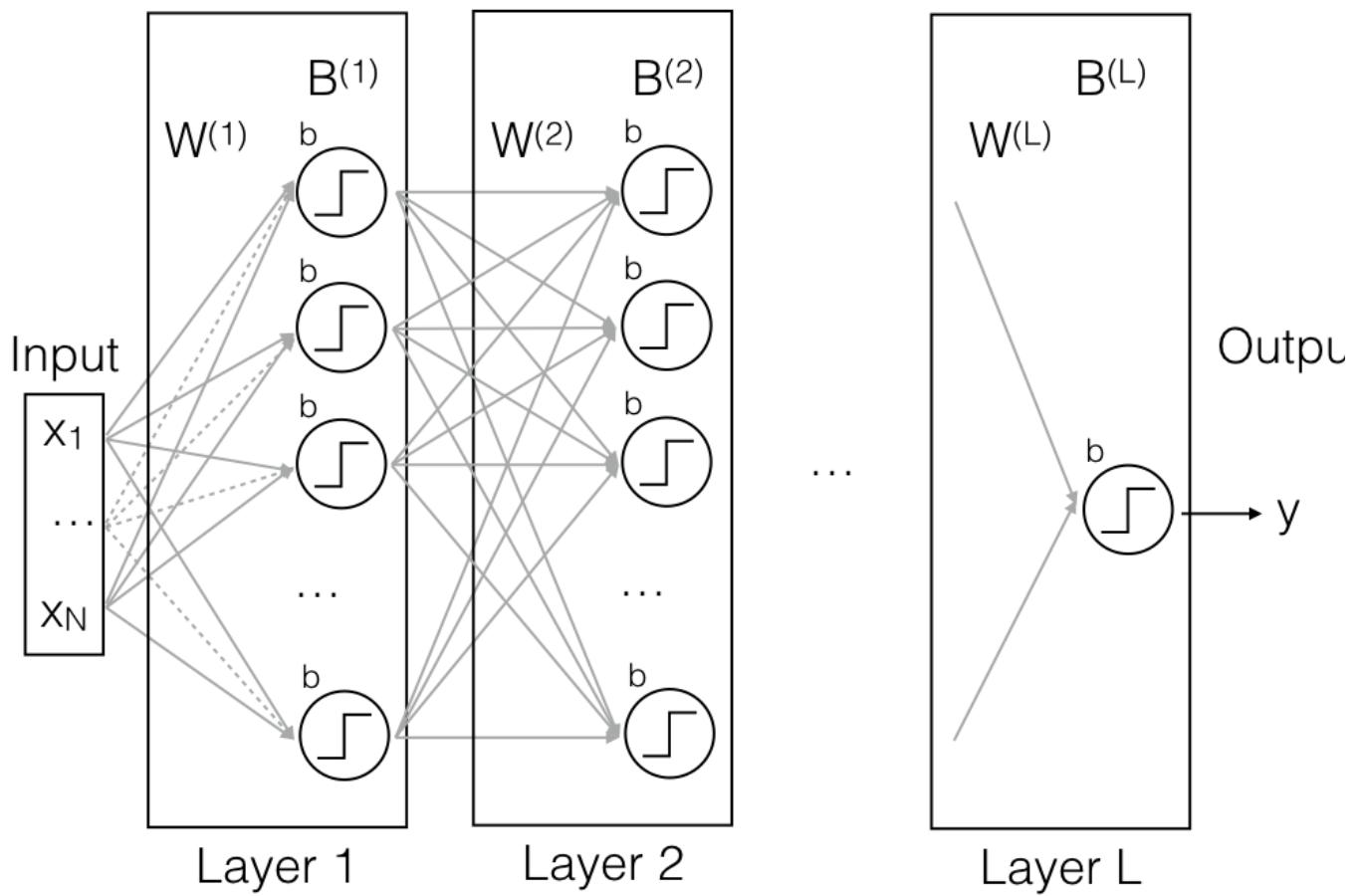
The weighted sum of the input plus the bias is sometimes also called a *linear combination* of the input features because it only involves sums and multiplications by parameters (no strange functions like exponentials, cosines etc.).

Let's see what happens when several Perceptrons are used in the same model.

We start by taking many Perceptrons, each connected by different weights to the same input nodes. Let's then calculate the output for each of the nodes, obtaining a bunch of different predictions, one for each of the Perceptrons. This is called a **fully connected layer**, sometimes called a **dense layer**.

We can think of a dense layer as is nothing more than many identical nodes connected to the same inputs through independent weights, all operating in parallel.

Nothing prevents us from using the output values of the dense layer as features (or inputs) for even more Perceptrons. In other words, we can create a deeper **fully connected Neural Network** by stacking fully connected layers on top of each other.



These **fully connected layers** are the root of deep learning and are used all the time.

Perceptrons with the same inputs are organized in layers, where a layer is just a group of Perceptrons that receive the same inputs. As we will see later, creating a fully connected network in `keras` is very easy, it's just a matter of adding more layers.

Maths of the Forward Pass

We can think of a Neural Network as a function (F), that takes an input value from the feature space and outputs a value in the target space. This calculation, called **Forward Pass** is a composition of linear and nonlinear steps.

For the math inclined reader, let's look at how we can write the operations performed by a node in the first layer. Each node in the first layer performs a linear transformation of the input features. Mathematically

speaking, it performs a weighted average of the features and then add a bias.

If we use the index k to enumerate the nodes in the first layer, we can write the weighted sum $z^{(1)}$ calculated by that node as:

$$z_k^{(1)} = x_1 w_{1k}^{(1)} + x_2 w_{2k}^{(1)} + \dots + b_k^{(1)} \text{ for every node } k \text{ in the first layer.}$$

where we have used the superscript (1) to indicate that the weights belong to the first layer, and the subscript jk to indicate the weight multiplying the input feature at position j for the node at position k .

In the previous example of the price prediction of a house, the index j runs over the features, i.e. so for example $j = 1$ locates the first feature, i.e. the size of the house in square feet (x_1).

If we consider all the input features as a vector $X = [x_1, x_2, x_3, \dots]$ and all the output sums of the first layer as a vector $Z^{(1)} = [z_1^{(1)}, z_2^{(1)}, z_3^{(1)}, \dots]$, the above weighted sum can be written as a matrix multiplication of the weight matrix $W^{(1)}$ with the input features:

TIP: if you are not familiar with vectors, matrices, and linear algebra you can keep going and ignore this mathematical part. There is a more in-depth discussion of these concepts in the next chapter. That said, linear algebra is a fundamental component of how machine learning and deep learning work. So if you are completely foreign to these notions, you may find it valuable to take a class or two on YouTube about vectors, matrices and their operations.

$$Z^{(1)} = X \cdot W^{(1)} + B^{(1)} = \sum_j x_j w_{jk}^{(1)} + b_k^{(1)}$$

where the weights are arranged in a matrix $W^{(1)}$ whose rows run along the input features and whose columns run along the nodes in the layer.

The nonlinear activation function will be applied to the weighted sum to yield the activation at the output. For example, in the case of the Perceptron, we will apply the step function like this:

$$A^{(1)} = H(Z^{(1)})$$

The activation vector $A^{(1)}$, is a vector of length k , and it becomes the input vector to the second layer in the network. The second layer will take the output of the first and perform the exact same calculation:

$$A^{(2)} = H(A^{(1)} \cdot W^{(2)} + B^{(2)})$$

yielding a new activation vector $A^{(2)}$ with as many elements as the number of nodes in the second layer.

This is true for any of the layers: a layer takes the output of the previous layer and performs a linear combination, followed by a non linear function. The nonlinear activation function is the most important part of the transformation. If that were not present, a deep network would produce the same result as a shallow network, and it wouldn't be powerful at all.

ACTIVATION FUNCTIONS

We've looked at two nonlinear activation functions already:

- the step function
- sigmoid

These functions are applied to the output weighted sum calculated by a layer before we pass the values onto the next layer or to output. They are the key element of Neural Networks. **Activation functions are what make Neural Networks so versatile and powerful!** Besides sigmoid and step functions there are other powerful options. Let's look at a few more.

```
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
```

Sigmoid and Step functions are easy to define using `numpy` (using their mathematical formulas):

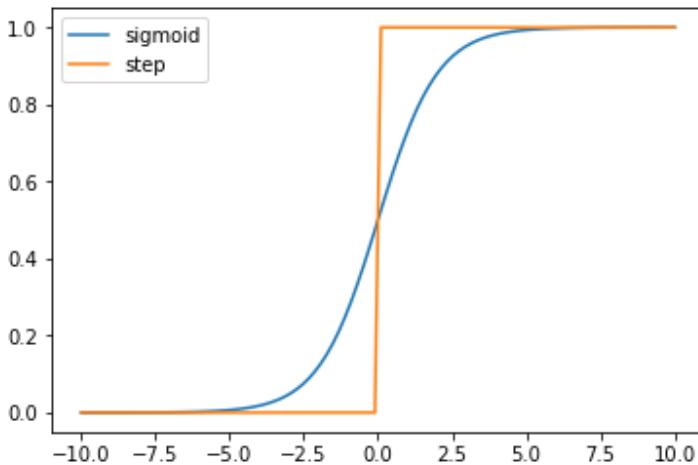
```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def step(x):
    return x > 0
```

They both map the real axis onto the interval between 0 and 1 ($[0, 1]$), i.e. they are bounded:

```
x = np.linspace(-10, 10, 100)
plt.plot(x, sigmoid(x))
plt.plot(x, step(x))
plt.legend(['sigmoid', 'step'])
```

```
<matplotlib.legend.Legend at 0x7f0625f3ba20>
```



They are designed to squeeze a large output sum to 1 while taking a really negative output that sums to 0.

It's as if each node was performing an independent classification of the input features and feeding the output binary outcome onto the next layer.

Besides the `sigmoid` and `step`, other nonlinear activation functions are possible and will be used in this book. Let's look at a few of them:

Tanh

The [hyperbolic tangent](#) has a very similar shape to the sigmoid, but it is bounded and smoothly varying between $[-1, +1]$ instead of $[0, 1]$, and is defined as:

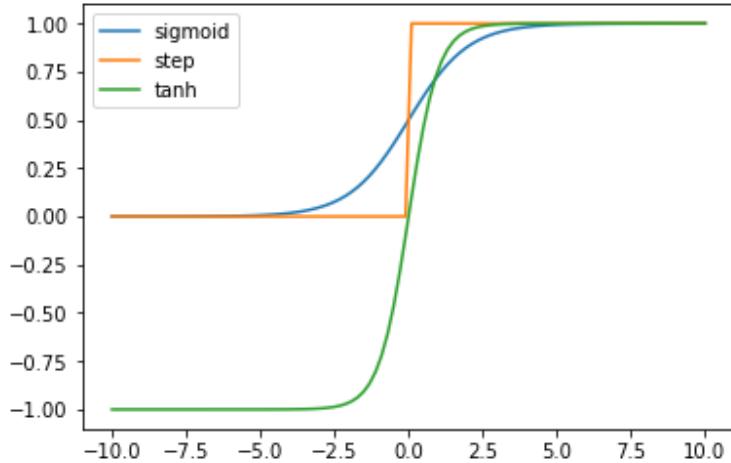
$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The advantage of this is that negative values of the weighted sum are not forgotten by setting them to zero, but are given a negative weight. In practice `tanh` makes the network learn much faster than `sigmoid` or `step`.

We can write the `tanh` function simply in python as well, but we don't have to. An efficient version of the `tanh` function is available through numpy:

```
x = np.linspace(-10, 10, 100)
plt.plot(x, sigmoid(x))
plt.plot(x, step(x))
plt.plot(x, np.tanh(x))
plt.legend(['sigmoid', 'step', 'tanh'])
```

```
<matplotlib.legend.Legend at 0x7f0625e75978>
```



ReLU

The rectified linear unit function or simply *rectifier* is defined as:

$$y = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

or simply:

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

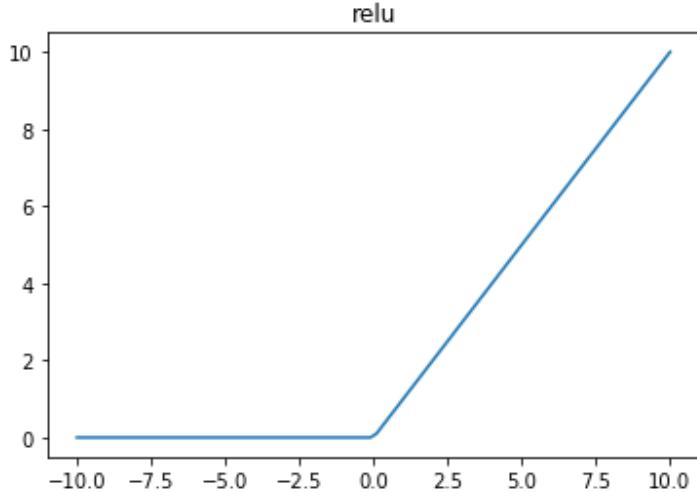
Originally motivated from biology, it has been shown to be very effective and it is probably the most popular activation function for deep Neural Networks. It offers two advantages.

1. If it's implemented as an `if` statement (the former of the two formulations above), its calculation is very fast, much faster than smooth functions like `sigmoid` and `tanh`.
2. Not being bounded on the positive axis, it can distinguish between two large values of input sum, which helps back-propagation converge faster.

```
def relu(x):
    cond = x > 0
    return cond * x
```

```
x = np.linspace(-10, 10, 100)
plt.plot(x, relu(x))
plt.title('relu')
```

```
Text(0.5,1,'relu')
```



Softplus

The **Softplus** function is a smooth approximation of the ReLU :

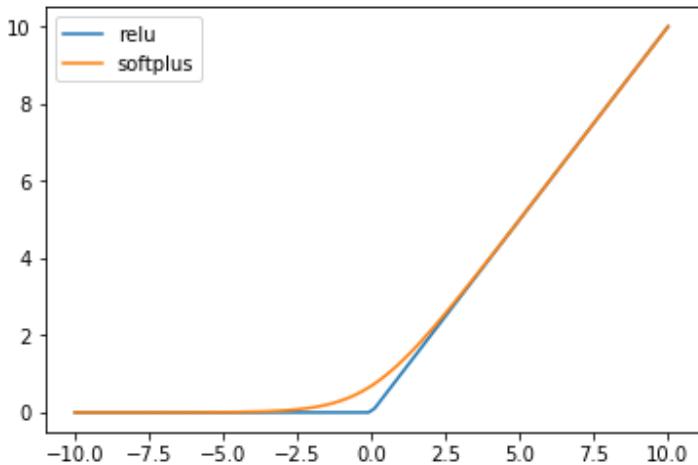
$$y = \log(1 + e^x)$$

We mention it for completeness, though it's rarely used in practice.

```
def softplus(x):
    return np.log1p(np.exp(x))
```

```
x = np.linspace(-10, 10, 100)
plt.plot(x, relu(x))
plt.plot(x, softplus(x))
plt.legend(['relu', 'softplus'])
```

```
<matplotlib.legend.Legend at 0x7f0625dd77f0>
```



SeLU

Finally, the SeLU activation function is a very recent development (see paper published in June 2017). The name stands for **scaled exponential linear unit** and it's implemented as:

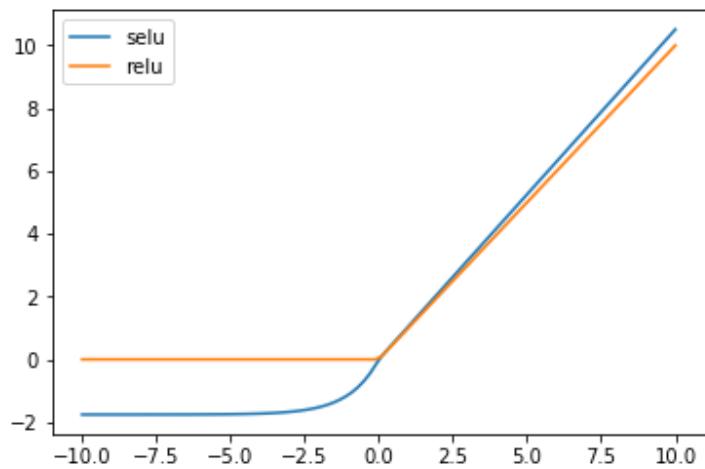
$$y = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

On the positive axis it behaves like the rectified linear unit (ReLU), scaled by a factor λ . On the negative axis it smoothly goes down to a negative value. This activation function, combined with a new regularization technique called **Alpha Dropout**, offers better convergence properties than ReLU !

```
def selu(x):
    alpha = 1.6732632423543772848170429916717
    scale = 1.0507009873554804934193349852946
    return scale * np.where(x>0.0, x, alpha * (np.exp(x) - 1))
```

```
x = np.linspace(-10, 10, 100)
plt.plot(x, selu(x))
plt.plot(x, relu(x))
plt.legend(['selu', 'relu'])
```

<matplotlib.legend.Legend at 0x7f0625d44dd8>



When creating a deep network, we will use one of these activation functions *between* one layer and the next, in order to make the Neural Network nonlinear. These functions are the secret power of neural networks: with nonlinearities at each layer they are able to approximate very complex functions.

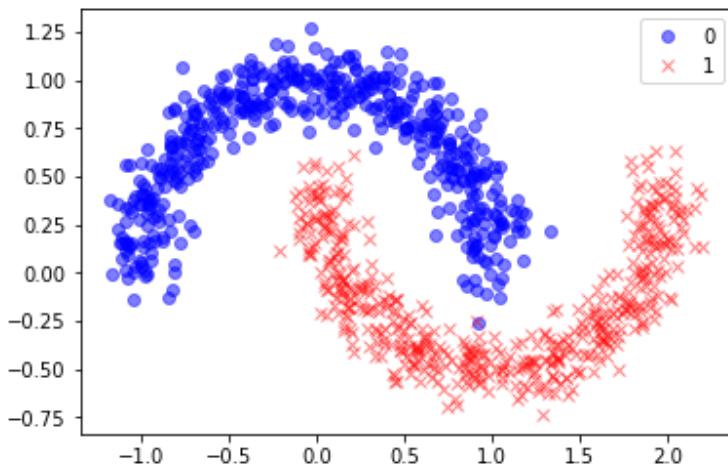
BINARY CLASSIFICATION

Let's work through classifying a binary dataset using a neural network. We'll need a dataset to work with to train our neural network. Let's create an example dataset with two classes that are not separable with a straight boundary, and let's separate them with a fully connected Neural Network:

```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.1, random_state=0)
plt.plot(X[y==0, 0], X[y==0, 1], 'ob', alpha=0.5)
plt.plot(X[y==1, 0], X[y==1, 1], 'xr', alpha=0.5)
plt.legend(['0', '1'])
```

<matplotlib.legend.Legend at 0x7f061b8acf98>



```
X.shape
```

(1000, 2)

We split the data into training and test sets:

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

To build our neural network, let's import a few libraries from the Keras package:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD, Adam
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: Fu
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Logistic Regression

Let's first verify that a shallow model cannot separate the two classes. This is more for educational purposes than anything else. We are going to build a model that we know is wrong, since it can only draw straight boundaries. This model will not be able to separate our data correctly but we will then be able to extend it and see the power of neural networks.

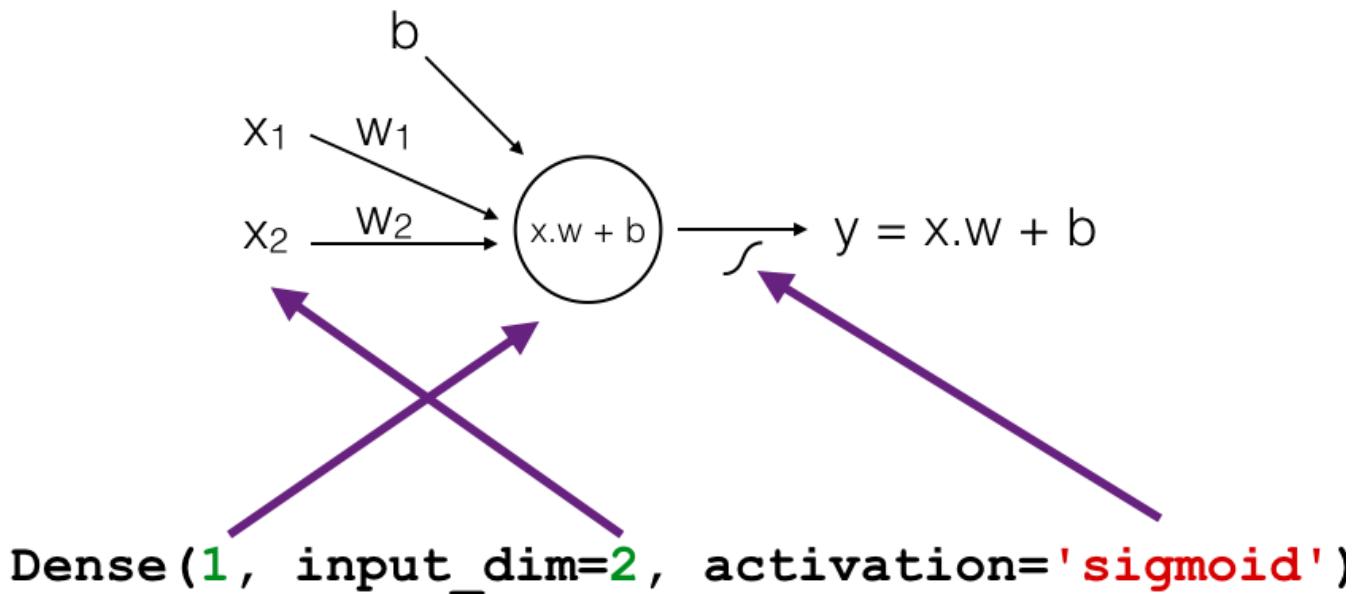
Let's start by building a **Logistic Regression** model like we did in the previous chapter. We will create it using the `Sequential` API, which is the simpler way to build models in Keras. We add a single `Dense` layer with 2 inputs, a single node and a `sigmoid` activation function:

```
model = Sequential()
```

Now we'll add a single `Dense` layer with 2 inputs and we'll use the `sigmoid` activation function here.

```
model.add(Dense(1, input_dim=2, activation='sigmoid'))
```

The arguments of the `Dense` layer definition map really well to our graph notation above



Then we compile the model assigning the optimizer, the loss and any additional metric we would like to include (like the accuracy in this case):

```
model.compile(Adam(lr=0.05), 'binary_crossentropy', metrics=['accuracy'])
```

Let's look at the three arguments to make sure we understand them.

- Adam(`lr=0.05`) is the `optimizer`, this is the algorithm that performs the actual learning. There are many different optimizers, and we will explore them in detail in the next chapter. For now know that Adam is a very good one.
- `binary_crossentropy` is the **loss or cost function**. We have described it in detail in [Chapter 3](#). For binary classification problems where we have a single output with a `sigmoid` activation we need to use `binary_crossentropy` function. For multi-class classifications where we have multiple classes with a `softmax` activation we need to use `categorical_crossentropy`, as we'll see below.
- `metrics` is just a list of additional metrics we'd like to calculate, in this case we add the `accuracy` of our classification, i.e. the fraction of correct predictions as seen in [Chapter 3](#).

As we have seen in the [previous chapter](#), we can now train the compiled model using our training data. The `model.fit(X, y)` method does just that: it uses the training inputs `x_train` to generate predictions. It then compares the predictions with the actual labels `y_train` through the use of the cost function and it finally adapts the parameters to minimize such cost.

We will train our model for 200 epochs, which means our model will get to see our training data completely for 200 times. We also set `verbose=0` to suppress printing during the training. Feel free to change it to `verbose=1` or `verbose=2` if you want to monitor training as it progresses.

```
model.fit(X_train, y_train, epochs=200, verbose=0)
```

```
<keras.callbacks.History at 0x7f05c863b470>
```

Now that we have trained our model, we can evaluate its performance on the test data using the function `.evaluate`. This takes the input features of the test data `X_test` and the input labels of the test data `y_test` and calculates the average loss and any other metric added during `model.compile`. In the present case `.evaluate` will return 2 numbers, the loss (cost) and the accuracy:

```
results = model.evaluate(X_test, y_test)
```

```
300/300 [=====] - 0s 119us/step
```

We can print out the accuracy by retrieving the second element in the `results` tuple:

```
print("The Accuracy score on the Test set is:\t{:0.3f}".format(results[1]))
```

```
The Accuracy score on the Test set is: 0.847
```

The accuracy is better than random guessing, but it's not 100%. Let's see the boundary identified by the logistic regression by plotting the boundary as a line:

```
def plot_decision_boundary(model, X, y):
    amin, bmin = X.min(axis=0) - 0.1
    amax, bmax = X.max(axis=0) + 0.1
    hticks = np.linspace(amin, amax, 101)
    vticks = np.linspace(bmin, bmax, 101)

    aa, bb = np.meshgrid(hticks, vticks)
    ab = np.c_[aa.ravel(), bb.ravel()]
```

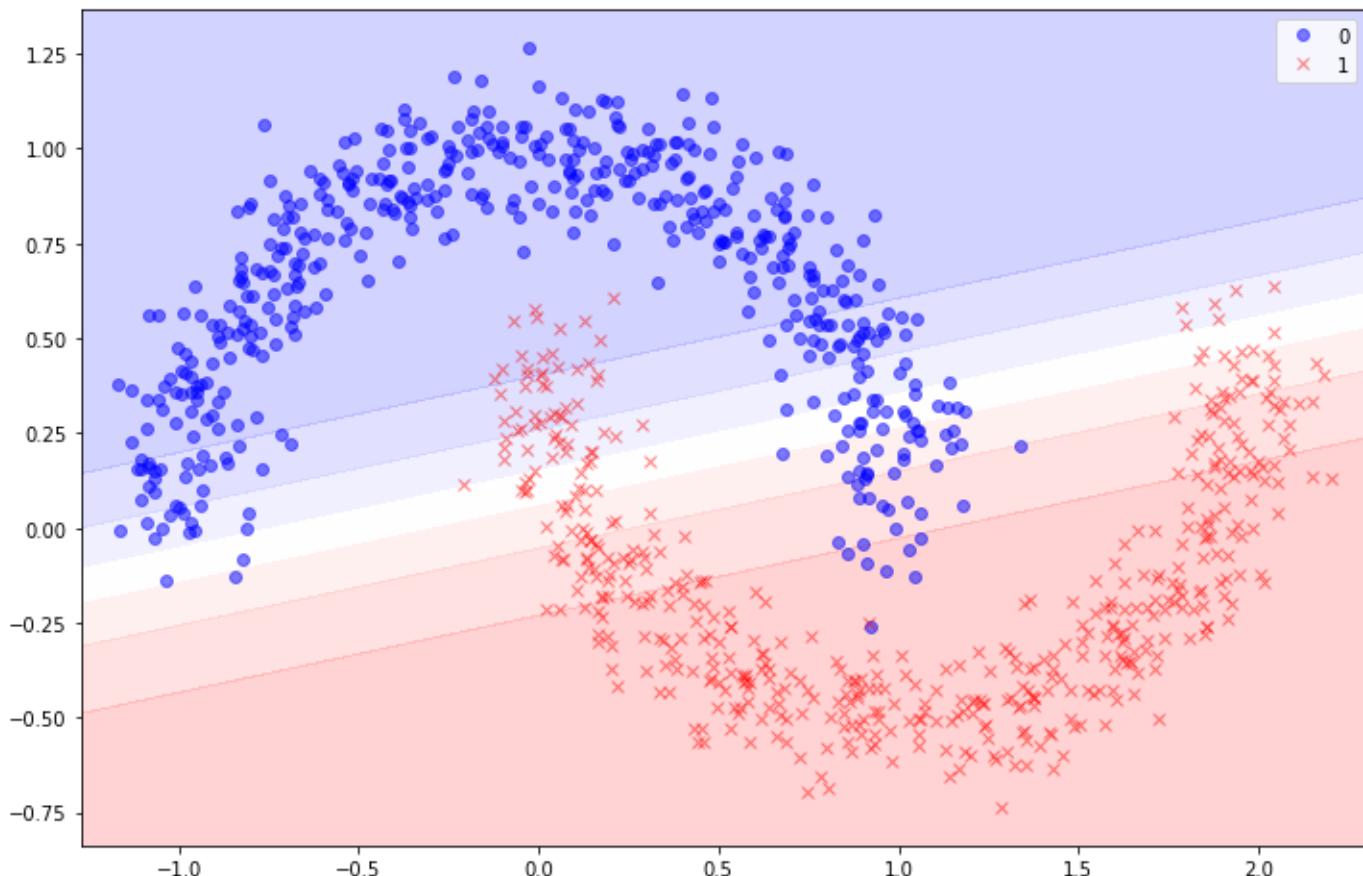
```

c = model.predict(ab)
cc = c.reshape(aa.shape)

plt.figure(figsize=(12, 8))
plt.contourf(aa, bb, cc, cmap='bwr', alpha=0.2)
plt.plot(X[y==0, 0], X[y==0, 1], 'ob', alpha=0.5)
plt.plot(X[y==1, 0], X[y==1, 1], 'xr', alpha=0.5)
plt.legend(['0', '1'])

plot_decision_boundary(model, X, y)

```

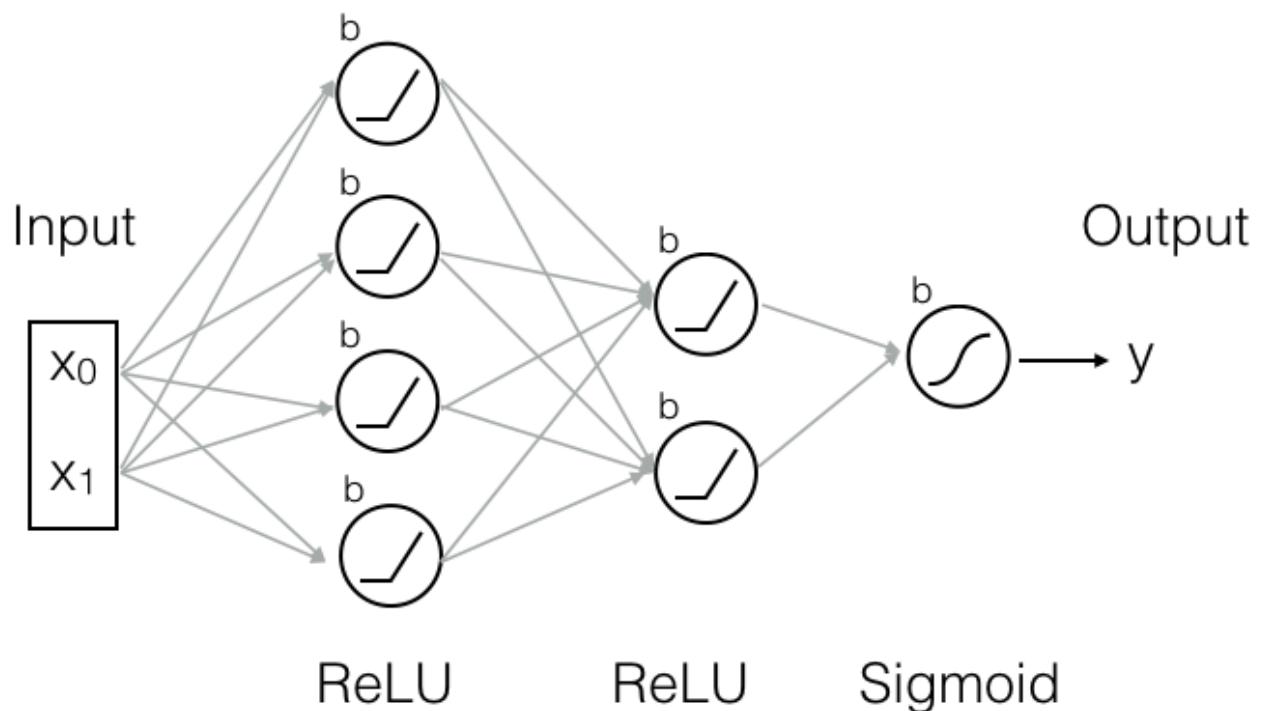


As you can see in the figure, since a shallow model like logistic regression is not able to draw curved boundaries, the best it can do is align the boundary so that most of the blue dots fall in the blue region and most of the red crosses fall in the red region.

Deep model

The word **deep** in Deep Learning has changed meaning over time. Initially it was used to refer to networks that had more than a single layer. As the field progressed and more and more complex models were invented, the word shifted to meaning networks with hundreds of layers and billions of parameters. In this book we will use the original meaning and call "deep" any model with more than one layer, so let's add a few layers and create our first "deep" model.

Let's build a model with the following structure:



This model has 3 layers. The first layer has 4 nodes, with 2 inputs and a `relu` activation function. The 4 values at the output of the first layer will be fed into a second layer with 2 nodes and a `relu` activation function and finally the 2 outputs of this layer will be fed into the third layer, which is also our output layer. This only has 1 node and a `sigmoid` activation function, so that the output values are constrained between 0 and 1.

We can build this network in `keras` very easily. All we have to do is add more layers to the `Sequential` model, specifying the number of nodes and the activation for each of them using the `.add()` function. Let's start with the first layer:

```
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))
```

This is very similar to what we did above, except that now this `Dense` layer has 4 nodes instead of 1. How many parameters are there in this layer? There are 12 parameters, 2 weights for each of the nodes (2×4) plus 1 bias for each of the nodes (4).

Let's now add a second layer after the first one, with 2 nodes:

```
model.add(Dense(2, activation='relu'))
```

Notice that we didn't have to specify the `input_dim` parameter, because keras is smart and automatically matches it with the output size of the previous layer.

Finally, let's add the output layer:

```
model.add(Dense(1, activation='sigmoid'))
```

and let's compile the model:

```
model.compile(Adam(lr=0.05), 'binary_crossentropy', metrics=['accuracy'])
```

The `input_dim` parameter is the number of dimensions in our input data points. In this case, each point is described by two numbers, so the input dimension is equal to 2 (for the first `Dense()` layer). `Dense(1)` is the output layer. Here we are classifying 2 classes, blue dots and red crosses, and therefore it's a binary classification and we are predicting a single number: the probability of being in the class of the red crosses.

Let's train it and see how it performs, using the `.fit()` method again:

```
model.fit(X_train, y_train, epochs=100, verbose=0)
```

```
<keras.callbacks.History at 0x7f05840b5d68>
```

We'll use a couple handy functions from the `sklearn.metrics` package, the `accuracy_score()` and `confusion_matrix()` functions. First of all let's see what classes our model predicts using the `.predict_classes()` method:

```
y_train_pred = model.predict_classes(X_train)
y_test_pred = model.predict_classes(X_test)
```

This is different from the `.predict()` method because it returns the actual predicted class instead of the predicted probability of each class.

```
y_train_prob = model.predict(X_train)
y_test_prob = model.predict(X_test)
```

Let's look at the first few values for comparison:

```
y_train_pred[:3]
```

```
array([[1,
       [1,
        [0]], dtype=int32)
...
[0]], dtype=int32)
...
[0]], dtype=int32)
```

```
y_train_prob[:3]
```

```
array([[1.0,
       [0.9999995 ],
       [0.00102984]], dtype=float32)
...
[0.00102984]], dtype=float32)
...
[0.00102984]], dtype=float32)
```

Let's compare the predicted classes with the actual classes on both the training and the test set. First, let's import the `accuracy_score` and the `confusion_matrix` methods from `sklearn`:

```
from sklearn.metrics import accuracy_score, confusion_matrix
```

Let's check out the score accuracy here for both the training set and the test set:

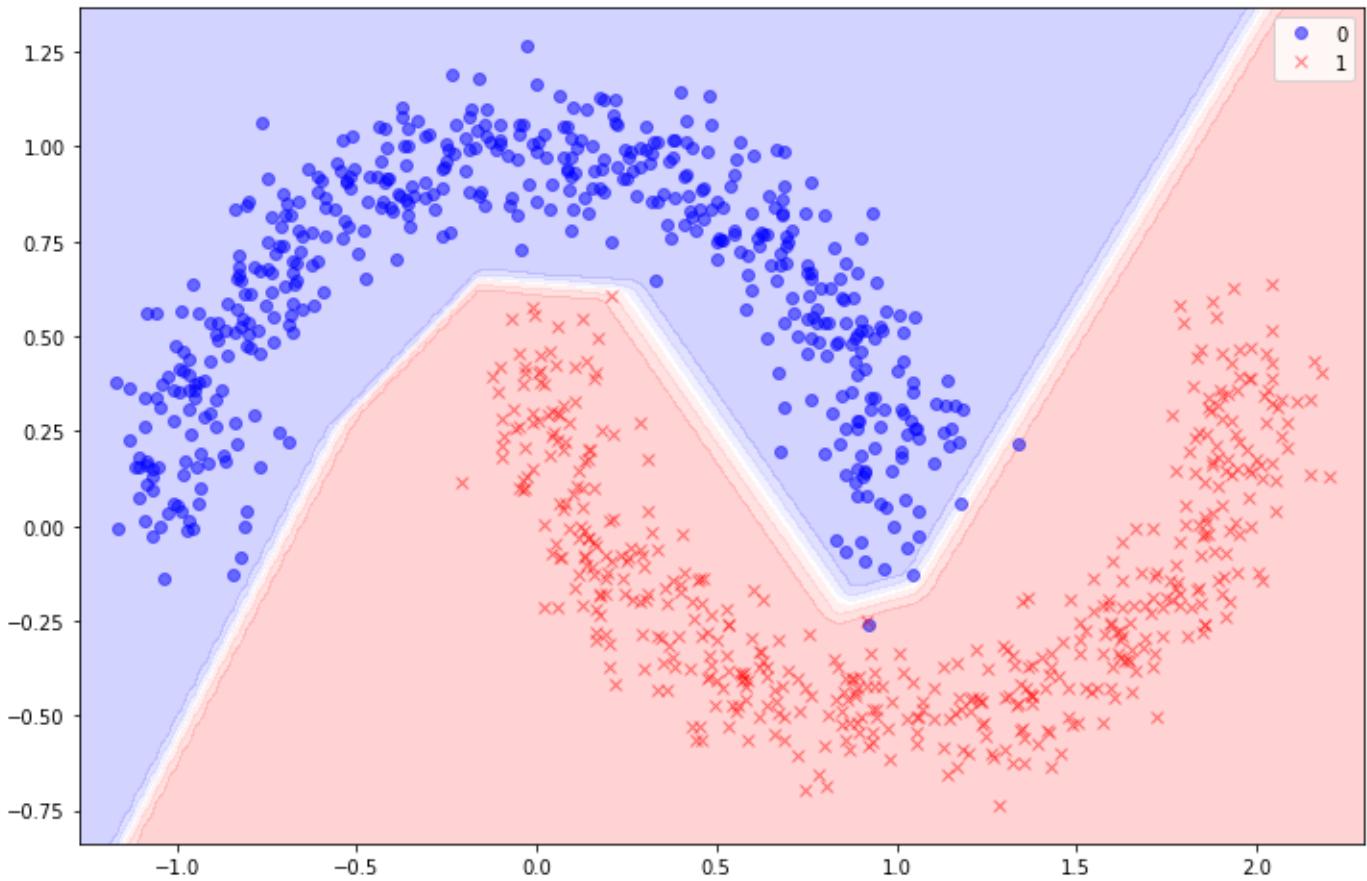
```
acc = accuracy_score(y_train, y_train_pred)
print("Accuracy score (Train set):\t{:0.3f}".format(acc))

acc = accuracy_score(y_test, y_test_pred)
print("Accuracy score (Test set):\t{:0.3f}".format(acc))
```

```
Accuracy score (Train set):      0.999
Accuracy score (Test set):      0.997
```

Let's plot the decision boundary for the model:

```
plot_decision_boundary(model, X, y)
```



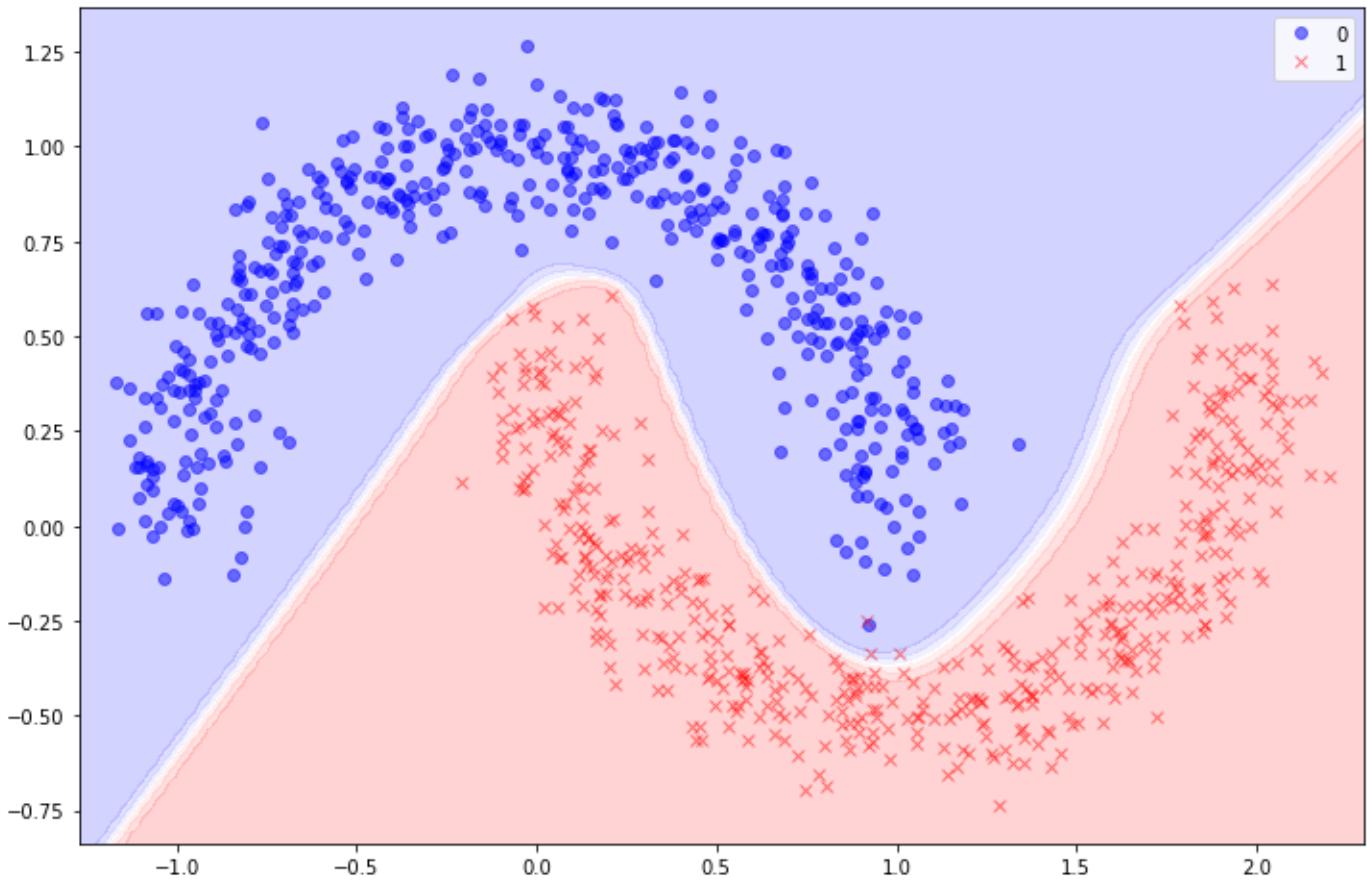
As you can see, our network learned to separate the two classes with a zig-zag boundary, which is typical of the `ReLU` activation.

Let's try building our model again, but use a different activation this time. If we used the `tanh` function instead, we'd have obtained a smoother boundary:

```
model = Sequential()
model.add(Dense(4, input_dim=2, activation='tanh'))
model.add(Dense(2, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
model.compile(Adam(lr=0.05),
             loss='binary_crossentropy',
             metrics=['accuracy'])

model.fit(X_train, y_train, epochs=100, verbose=0)

plot_decision_boundary(model, X, y)
```



```

y_train_pred = model.predict_classes(X_train)
y_test_pred = model.predict_classes(X_test)

acc = accuracy_score(y_train, y_train_pred)
print("Accuracy score (Train set):\t{:0.3f}".format(acc))

acc = accuracy_score(y_test, y_test_pred)
print("Accuracy score (Test set):\t{:0.3f}".format(acc))

```

```

Accuracy score (Train set):      0.999
Accuracy score (Test set):      0.993

```

Adding depth to our model allows us to separate two classes with a boundary of arbitrary shape. The complexity of the boundary profile is given by the number of nodes and layers we add to the network. The

more we add, the more parameters our network will learn. This is really powerful because we can always add more layers if we want to be able to capture more complex boundaries.

Deep Learning models can have as little as few hundred parameters to as much as a few billions. As models get more parameters, they also need more data, so to train a model with millions of parameters we will likely need tens of millions of data points. This will also imply much computational resources as we shall see later on.

MULTICLASS CLASSIFICATION

Neural Networks can be easily extended to cases where the output is not a single value.

In the case of regression, this means that the output is a vector, while in the case of classification, it means we have more than one class we'd like to separate.

For example, if we are doing image recognition, we may have several classes for all the objects we'd like to distinguish (e.g. cat, dog, mouse, bird, etc.). Instead of having a single output Yes/No, we allow the network to predict multiple values.

Similarly, for a self driving car, we may want our network to predict the direction of the trajectory the car should take, which means both the speed and the steering angle. This would be a regression with multiple outputs at the same time. The extension is trivial in the case of regression: we add as many output nodes as needed and minimize the mean squared error on the whole vector output.

The case of classification requires a little more discussion, because we need to carefully choose the activation function. In fact, when we are predicting discrete output we could be in one of two cases:

1. we could be predicting mutually exclusive classes
2. each class could be independent

Let's consider the example of email classification. We would like to use our Machine Learning model to organize a large pool of emails sitting in our inbox. We could choose two way to organize them.

Tags

One way to arrange our emails would be to add tags to each email to specify the content. We could have a tag for `work`, a tag for `Personal`, but also a tag for `Has_Picture` or `Has_Attachment`. These tags are not mutually exclusive. Each one is independent from the others and a single email could carry multiple tags.

The extension of the Neural Network to this case is also pretty straightforward, because we will perform an independent logistic regression on each tag. Just like in the case of the regression, all we have to do is add multiple sigmoid output nodes and we are done.

Mutually exclusive classes

A different case is if we decided to arrange our emails in folders, for example: `work` , `Personal` , `Spam` etc., and move each email to the corresponding folder. In this case, each email can only be in one folder. If it's in folder `work` , it is automatically not in folder `Personal` . In this case, we cannot use independent sigmoids, we need to use an activation function that will normalize the output so that if a node predicts a high probability, all the others will predict a low probability and the sum of all the probabilities will add up to one.

Mathematically, the `softmax` function is a generalization of the logistic function that does just that:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \text{ for } j = 1, \dots, K.$$

When we deal with mutually exclusive classes, we always have to apply the `softmax` function to the last layer.

The Iris dataset

The Iris dataset is a classic dataset used in Machine Learning. It describes 3 species of flowers, with 4 features each, so it's a great example for a Multiclass classification. Let's see how Multiclass classification's done using `keras` and the Iris dataset. First of all let's load the data.

```
df = pd.read_csv('~/data/iris.csv')
```

```
df.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

We need to do a bit of massaging of the data, separate the input features from the target column containing the species.

First of all let's create a feature matrix X where we store the first 4 columns:

```
X = df.drop('species', axis=1)
X.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Let's also create a target column, where we encode the labels in alphabetical order. We need to do this because machine learning models do not understand string values like `setosa` or `versicolor`. We will first look at the unique values contained in the `species` column:

```
target_names = df['species'].unique()
target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

And then build a dictionary where we assign an index to each target name in alphabetical order:

```
target_dict = {n:i for i, n in enumerate(target_names)}
target_dict
```

```
{'setosa': 0, 'versicolor': 1, 'virginica': 2}
```

Now we can use the `.map` method to create a new Series from the `species` column, where each of the entries is replaced using `target_dict`:

```
y= df['species'].map(target_dict)  
y.head()
```

```
0    0  
1    0  
2    0  
3    0  
4    0  
Name: species, dtype: int64  
...  
Name: species, dtype: int64  
...  
Name: species, dtype: int64
```

Now `y` is a number indicating the class (0, 1, 2). In order to use this with Neural Networks, we need to perform one last step: we will expand it to 3 binary dummy columns. We could use the `pandas.get_dummies` function to do this, but Keras also offers an equivalent function, so let's use that instead:

```
from keras.utils import to_categorical
```

```
y_cat = to_categorical(y)
```

Let's check out what the data looks like by looking at the first 5 values:

```
y_cat[:5]
```

```
array([[1.,  0.,  0.],  
       [1.,  0.,  0.],  
       [1.,  0.,  0.],  
       [1.,  0.,  0.],  
       [1.,  0.,  0.]], dtype=float32)  
...  
[1.,  0.,  0.]], dtype=float32)
```

```
...  
[1., 0., 0.]], dtype=float32)
```

Now we create a train and test split, with 20% test size. We'll pass the values of the `x` dataframe because keras doesn't like pandas dataframes. Also notice that we introduce 2 more parameters:

- `stratify = True` to make sure that we preserve the ratio of labels in each set, i.e. we want each set to be composed of one third of each flower type.
- `random_state = 0` sets the seed of the random number generator in a way that we all get the same results.

```
x_train, x_test, y_train, y_test = train_test_split(X.values,  
                                                 y_cat,  
                                                 test_size=0.2,  
                                                 random_state=0,  
                                                 stratify=y)
```

and then create a model with:

- 4 features in input (the `sepal_length`, `sepal_width`, `petal_length`, `petal_width`)
- 3 in output (each one being the probability of the flower being one of `setosa`, `versicolor`, `virginica`)
- A `softmax` activation

This is a shallow model, equivalent of a Logistic Regression with 3 classes instead of two.

```
model = Sequential()  
model.add(Dense(3, input_dim=4, activation='softmax'))  
model.compile(Adam(lr=0.1),  
             loss='categorical_crossentropy',  
             metrics=['accuracy'])
```

```
model.fit(x_train, y_train, validation_split=0.1, epochs=30)
```

```
Train on 108 samples, validate on 12 samples  
Epoch 1/30  
108/108 [=====] - 0s 2ms/step - loss: 4.2067 - acc: 0.3611 - val  
Epoch 2/30  
108/108 [=====] - 0s 103us/step - loss: 1.6192 - acc: 0.4444 - v
```

```
Epoch 3/30
108/108 [=====] - 0s 104us/step - loss: 1.8737 - acc: 0.3519 - v
Epoch 4/30
108/108 [=====] - 0s 102us/step - loss: 1.0101 - acc: 0.4722 - v
...
108/108 [=====] - 0s 102us/step - loss: 1.0101 - acc: 0.4722 - v
```

```
<keras.callbacks.History at 0x7f05c411feb8>
```

The output of the model is a matrix with 3 columns, corresponding to the predicted probabilities for each class where each of the 3 output predictions are listed in the columns, ordered by their order in the `y_train` array:

```
y_pred = model.predict(X_test)
y_pred
```

```
array([[ 9.7439539e-01,  2.5596119e-02,  8.4583598e-06],
       [1.7425800e-02,  7.1638799e-01,  2.6618627e-01],
       [9.3952131e-01,  6.0392823e-02,  8.5930464e-05],
       [2.0174359e-03,  4.0127972e-01,  5.9670281e-01],
       [9.6077454e-01,  3.9198186e-02,  2.7247495e-05],
       [1.8846218e-02,  7.1935290e-01,  2.6180086e-01],
       [8.7638217e-04,  3.5333380e-01,  6.4578980e-01],
       [9.4799620e-01,  5.1942635e-02,  6.1134131e-05],
       [9.3504727e-01,  6.4859673e-02,  9.3109527e-05],
       ...
       [9.5420361e-01,  4.5758460e-02,  3.7999958e-05]], dtype=float32)
```

Which class does our network think each flower is? We can obtain the predicted class with the `np.argmax` , which finds the index of the maximum value in an array:

```
y_test_class = np.argmax(y_test, axis=1)
y_pred_class = np.argmax(y_pred, axis=1)
```

Let's check the classification report and confusion matrix that we have described in Chapter 3

As you may remember, `classification_report()` and the `confusion_matrix()` functions are found in `sklearn.metrics` package:

```
from sklearn.metrics import classification_report, confusion_matrix
```

To create a classification report, we'll run the `classification_report()` method, passing it the test class (the list that we created before of the *correct* labels for each dataum) and the `y_pred_class` (the list we just obtained of the predicted classes).

```
print(classification_report(y_test_class, y_pred_class))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	10
1	0.91	1.00	0.95	10
2	1.00	0.90	0.95	10
avg / total	0.97	0.97	0.97	30

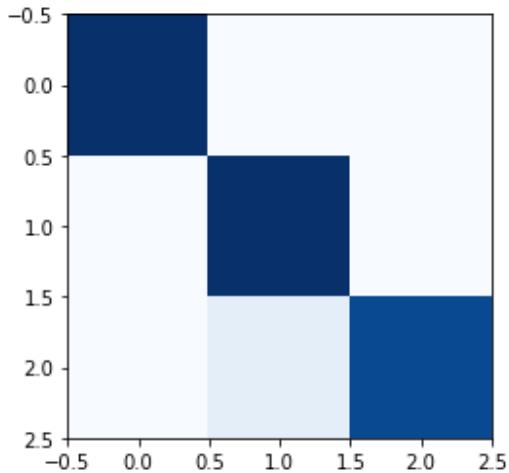
We get the confusion matrix by running the `confusion_matrix()` method passing it the same arguments as the classification report:

```
cm = confusion_matrix(y_test_class, y_pred_class)
pd.DataFrame(cm, index = target_names, columns = ['pred_'+c for c in target_names])
```

	pred_setosa	pred_versicolor	pred_virginica
setosa	10	0	0
versicolor	0	10	0
virginica	0	1	9

```
plt.imshow(cm, cmap='Blues')
```

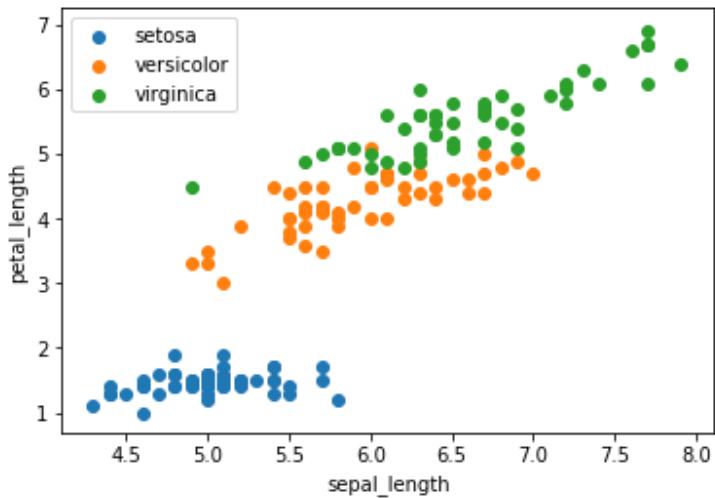
```
<matplotlib.image.AxesImage at 0x7f04c409f438>
```



Recall that the confusion matrix tells us how many examples from one class are predicted in each class. It's almost perfect, with the exception of one point in class `virginica` which gets predicted in class `versicolor`. Let's inspect the data visually to check why. Our data has 4 features, so we need to decide how to plot it. We could choose 2 features and plot just those:

```
plt.scatter(X.loc[y==0, 'sepal_length'], X.loc[y==0, 'petal_length'])
plt.scatter(X.loc[y==1, 'sepal_length'], X.loc[y==1, 'petal_length'])
plt.scatter(X.loc[y==2, 'sepal_length'], X.loc[y==2, 'petal_length'])
plt.xlabel('sepal_length')
plt.ylabel('petal_length')
plt.legend(target_names)
```

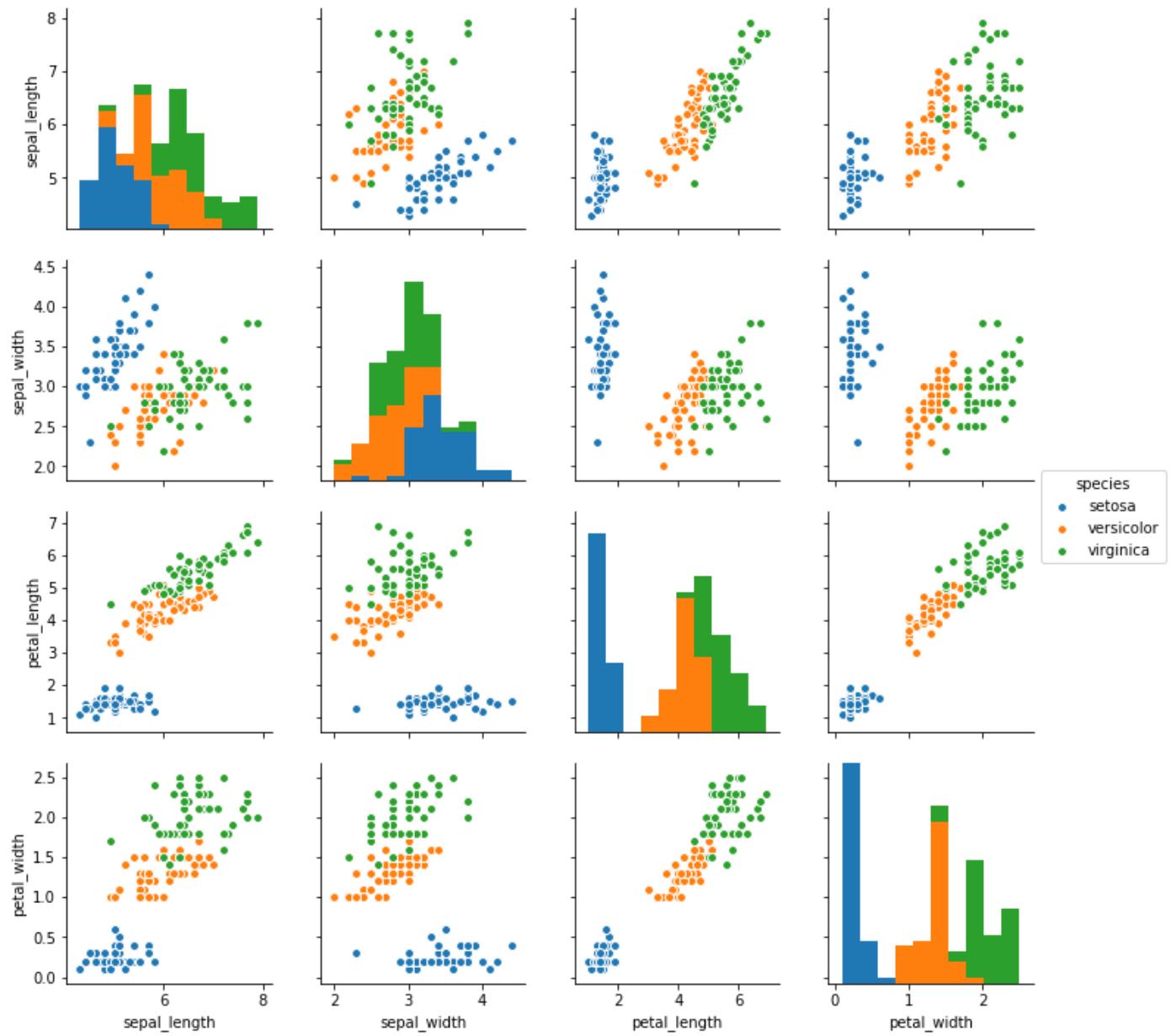
```
<matplotlib.legend.Legend at 0x7f04c41341d0>
```



Classes `virginica` and `versicolor` are slightly overlapping, which could explain why our model couldn't separate them too well. Is it true for every feature? We'll check that with a very cool visualization library called **Seaborn**. Seaborn improves Matplotlib with additional plots, for example the `pairplot`, which plots all possible pairs of features in a scatter plot:

```
import seaborn as sns  
  
sns.pairplot(df, hue="species")
```

```
<seaborn.axisgrid.PairGrid at 0x7f04907b8240>
```



As you can see `virginica` and `versicolor` overlap in all the features, which can explain why our model confuses them. Keep in mind that we used a shallow model to separate them instead of a deeper one.

CONCLUSION

In this chapter we have introduced fully connected deep Neural Networks and seen how they can be used to solve linear and nonlinear regression and classification problems. In the exercises we will apply them to predict the onset of diabetes in a population.

EXERCISES

Exercise 1

The [Pima Indians dataset](#) is a very famous dataset distributed by UCI and originally collected from the National Institute of Diabetes and Digestive and Kidney Diseases. It contains data from clinical exams for women age 21 and above of Pima Indian origins. The objective is to predict, based on diagnostic measurements, whether a patient has diabetes.

It has the following features:

- Pregnancies: Number of times pregnant
- Glucose: Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- BloodPressure: Diastolic blood pressure (mm Hg)
- SkinThickness: Triceps skin fold thickness (mm)
- Insulin: 2-Hour serum insulin (mu U/ml)
- BMI: Body mass index (weight in kg/(height in m)²)
- DiabetesPedigreeFunction: Diabetes pedigree function
- Age: Age (years)

The last column is the outcome, and it is a binary variable.

In this first exercise we will explore it through the following steps:

1. Load the `..data/diabetes.csv` dataset, use `pandas` to explore the range of each feature
 - For each feature draw a histogram. Bonus points if you draw all the histograms in the same figure.
 - Explore correlations of features with the outcome column. You can do this in several ways, for example using the `sns.pairplot` we used above or drawing a heatmap of the correlations.
 - Do features need standardization? If so what standardization technique will you use? MinMax? Standard?
 - Prepare your final `x` and `y` variables to be used by a ML model. Make sure you define your target variable well. Will you need dummy columns?

Exercise 2

Build a fully connected NN model that predicts diabetes. Follow these steps:

1. split your data in a train/test with a test size of 20% and a `random_state = 22`

- define a sequential model with at least one inner layer. You will have to make choices for the following things:
 - what is the size of the input?
 - how many nodes will you use in each layer?
 - what is the size of the output?
 - what activation functions will you use in the inner layers?
 - what activation function will you use at output?
 - what loss function will you use?
 - what optimizer will you use?
- fit your model on the training set, using a `validation_split` of 0.1
- test your trained model on the test data from the train/test split
- check the accuracy score, the confusion matrix and the classification report

Exercise 3

Compare your work with the results presented in [this notebook](#). Are your Neural Network results better or worse than the results obtained by traditional Machine Learning techniques?

- Try training a Support Vector Machine or a Random Forest model on the exact same train/test split. Is the performance better or worse?
- Try restricting your features to only 4 features like in the suggested notebook. How does model performance change?

Exercise 4

[Tensorflow playground](#) is a web based Neural Network demo. It is really useful to develop an intuition about what happens when you change architecture, activation function or other parameters. Try playing with it for a few minutes. You don't need to understand the meaning of every knob and button in the page, just get a sense for what happens if you change something. In the next chapter we'll explore these things in more detail.

Chapter 5: Deep Learning Internals

THIS IS A SPECIAL CHAPTER

In the last chapter we introduced the Perceptron with weights, biases and activation functions and fully connected Neural Networks. This chapter is a bit different from all the other chapters and it is meant for the reader who is interested in understanding the inner workings of a neural network.

In this chapter we learn about gradient descent and backpropagation. This sure much more technical and abstract than the rest of the book. There are mathematical formulas, weird symbols, derivatives, gradients and much more. We will try to make these concepts as intuitive and simple as possible, but these are complex topics and it is not possible to introduce them fully without going into some level of detail.

Let us first tell you: **you don't NEED to read this chapter.** This book is meant for the developer and practitioner that is interested in applying neural networks to solve great problems. As such, all the previous and following chapters are focused on the implementation of neural networks and their practical application to several problems. This chapter is different from all the others, you will not learn new applications here, you will not learn new commands or tricks nor we will introduce any new neural network architecture.

All this chapter does, is explain what happens when you run the function `model.fit`, i.e. break down how a neural network is trained. As we have already seen in chapters 3 and 4 after we define the model architecture we usually do 2 more steps:

1. we `.compile` the model specifying the optimizer and the cost function
- we `.fit` the model for a certain number of epochs using the training data

These two operations are executed by Keras for us and we don't have to worry about them too much. However, I'm sure you've been wondering why we choose a particular optimizer at compilation or what is actually happening during training.

This chapter explains exactly that.

In our opinion it is important to learn this for a few of reasons. First of all, understanding these concepts allow us to demystify what's actually happening under the hood with our network. Neural networks are not magic, and knowing these concepts can give us a better ability to judge where we can use them to solve problems and where we cannot. Secondly, knowing the internal mechanisms increases our abilities to understand which parameters can be tweaked and which optimization algorithms to choose.

So, let us re-iterate this once again: feel free to skip this chapter if your main goal is to learn how to use Keras and to apply neural networks. You won't find new code here, mostly a lot of maths and formulas.

On the other hand, if your goal is to understand how things actually work, then go ahead and read it. Chances are you will find the answers to some of your questions in this chapter.

Finally, if you are already familiar with derivatives and college calculus, you can probably skim through this large portions of this chapter quite quickly.

All that said, let's start by introducing derivatives and gradients. First let's import our usual libraries. By now you should be very familiar with all of them, but if in doubt on what they do, check back [Chapter 2](#) where we introduced them:

```
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
```

DERIVATIVES

As the name suggests a **derivative** is a function that derives from another function.

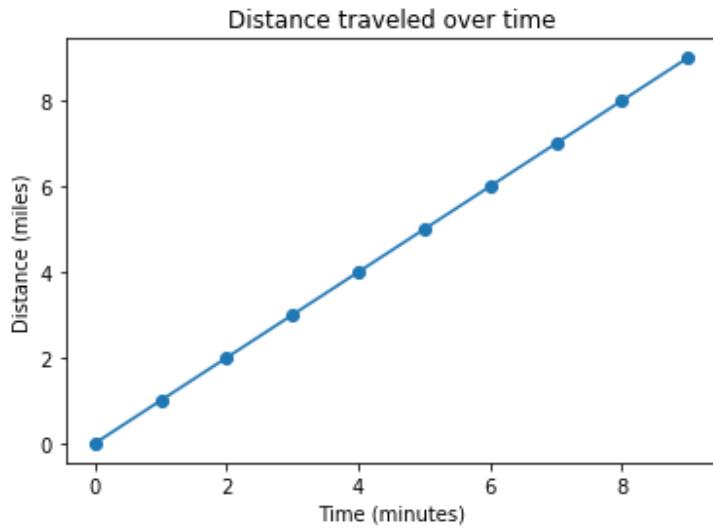
Let's start with an example. Imagine you are driving on the highway. As time goes by you mark your position along the highway, filling a table of values as a function of time. If your speed is 60 miles an hour, every minute your position will be increased by 1 mile.

Let's indicate your position as a function of time with the variable $x(t)$. Let's create an array of 10 minutes, called `t` and an array of your positions called `x` :

```
t = np.arange(10)
x = np.arange(10)
```

Now, let's make a plot to see the distance over time with respect to the distance traveled.

```
plt.plot(t, x, 'o-')
plt.title("Distance traveled over time")
plt.ylabel("Distance (miles)")
plt.xlabel("Time (minutes)")
plt.show()
```

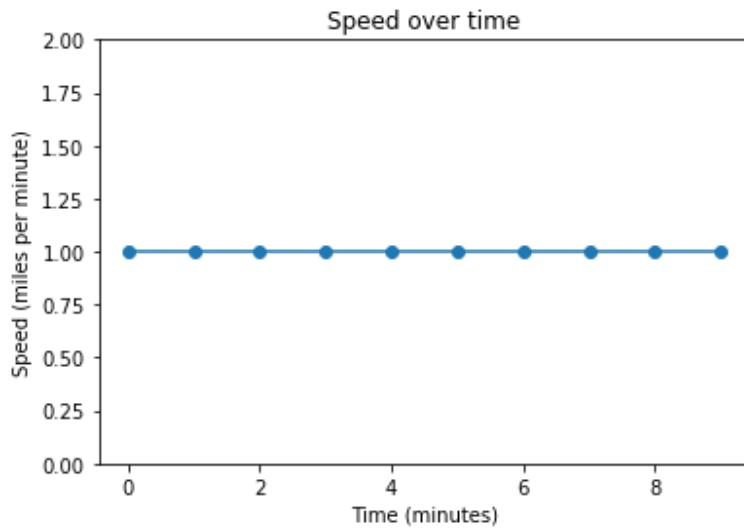


The derivative $x'(t)$ of this function is the **rate of change** in position with respect to time. In this example it is the speed of your car indicated by the odometer. In the example just mentioned, the derivative is a constant value of 60 miles per hour, or 1 mile per minute. Let's create an array containing the speed at each moment in time:

```
v = np.ones(10) # 1 mile per minute or 60 miles per hour
```

and let's plot it too:

```
plt.plot(t, v, 'o-')
plt.ylim(0, 2)
plt.title("Speed over time")
plt.ylabel("Speed (miles per minute)")
plt.xlabel("Time (minutes)")
plt.show()
```



In general, the derivative $x'(t)$ is itself a function of t that tells us the rate of change of the original function $x(t)$ at each point in time. This is why it is called a **derivative**. It can also be written explicitly as:

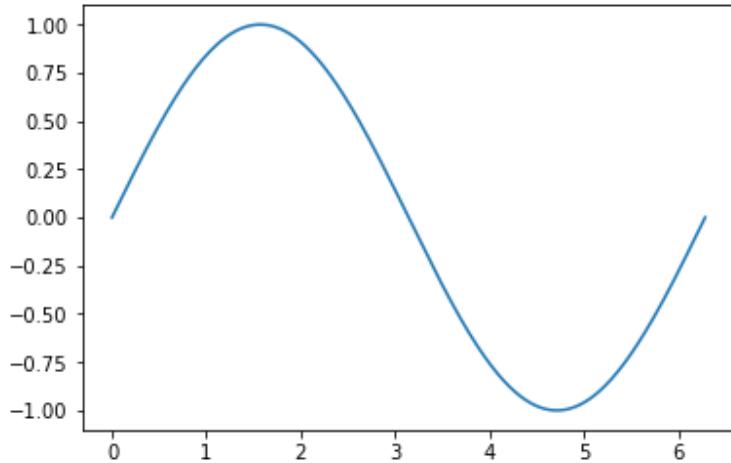
$$x'(t) := \frac{dx}{dt}(t)$$

Where the fraction $\frac{dx}{dt}$ indicates the ratio between a small change in x due to a small change in t . Let's look at a case where the derivative is not constant. Consider an arbitrary curve $f(t)$. Let's first create a slightly bigger time array:

```
t = np.linspace(0, 2*np.pi, 360)
```

Then let's take an arbitrary function and let's apply it to the array t . We will use the sine function, but that's just an example, any function would do:

```
f = np.sin(t)
plt.plot(t, f)
plt.show()
```



At each point along the curve $f(t)$, the derivative $f'(t)$ is equal to the rate of change in the function.

Finite differences

How do we calculate the value of the derivative at a particular point in t ? We can calculate its approximate value with the method of [finite differences](#):

$$\frac{df}{dt}(t_i) \approx \frac{\Delta f}{\Delta t}(t_i) = \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}}$$

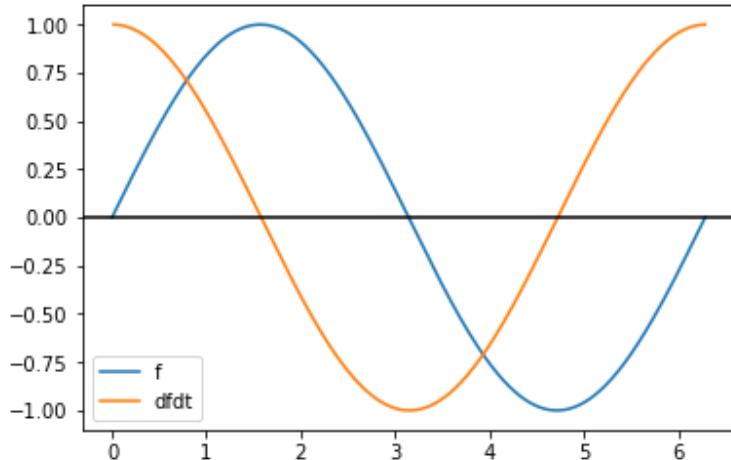
where we indicated with Δf the difference between two consecutive values of f .

We can calculate the value of the approximate derivative of the above function by using the function `np.diff` that calculates the difference between consecutive elements in an array:

```
dfdt = np.diff(f)/np.diff(t)
```

Let's plot it together with the original function:

```
plt.plot(t, f)
plt.plot(t[1:], dfdt)
plt.legend(['f', 'dfdt'])
plt.axhline(0, color='black')
plt.show()
```



If we read the figure from left to right, we notice that the value of the derivative is negative when the original curve is going downhill and it is positive when the original curve is going uphill. Finally, if we're at a minimum or at a maximum the derivative is 0 because the original curve is flat.

Let's define a simple helper function to plot the tangent line to our curve, i.e. the line that "just touches" the curve at that point:

```
def plot_tangent(i, color='r'):

    plt.plot(t, f)
    plt.plot(t[:-1], dfdt)
    plt.legend(['f', '$\\frac{df}{dt}$'])
    plt.axhline(0)

    ti = t[i]
    fi = f[i]
    dfdti = dfdt[i]

    plt.plot(ti, fi, 'o', color=color)
    plt.plot(ti, dfdti, 'o', color=color)
```

```

x = np.linspace(-0.75, 0.75, 20)
n = 1 + dfdti**2

plt.plot(ti + x/n, fi + dfdti*x/n, color, linewidth=3)

```

We can use this helper function to display the relationship between the inclination (slope) of our tangent line and the value of the derivative function. As you can see, a positive derivative corresponds to an uphill tangent while negative derivative corresponds to a downhill tangent line.

```

plt.figure(figsize=(14,5))

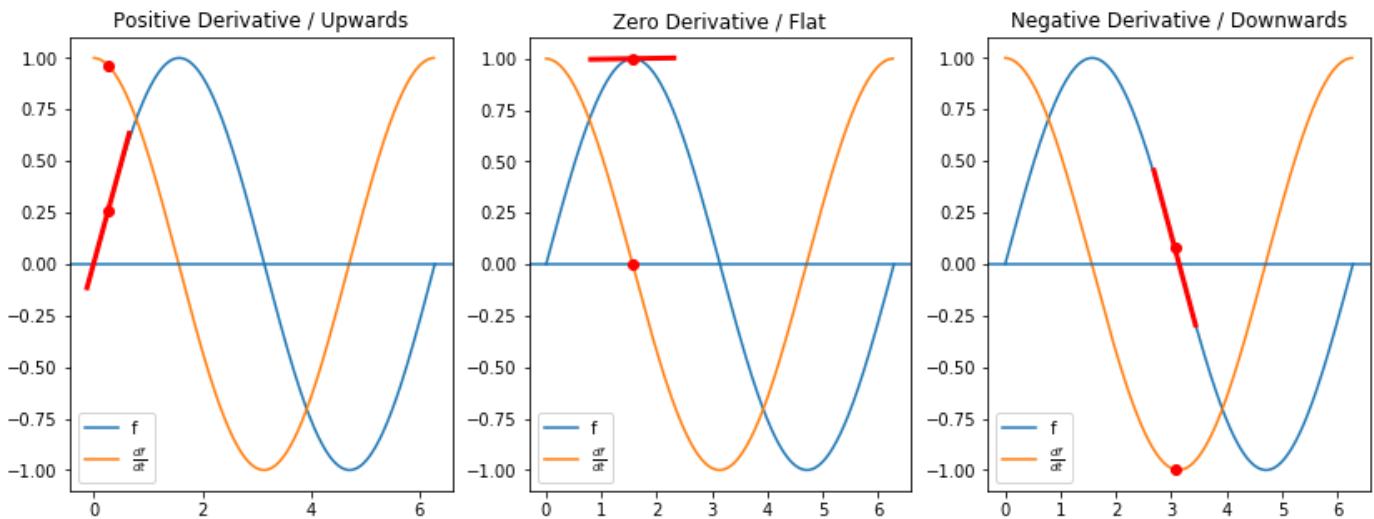
plt.subplot(131)
plot_tangent(15)
plt.title("Positive Derivative / Upwards")

plt.subplot(132)
plot_tangent(89)
plt.title("Zero Derivative / Flat")

plt.subplot(133)
plot_tangent(175)
plt.title("Negative Derivative / Downwards")

plt.show()

```



Although the finite differences method is useful to calculate the numerical value of a derivative, we don't need to use it. In fact, derivatives of simple functions are well known and we don't need to calculate them. Calculus is the branch of mathematics that deals with all this. For our purposes we will simply summarize here a few common functions and their derivatives:

Function Formula Derivative

constant	c	0
linear	x	1
power	x^n	nx^{n-1}
sine	$\sin(x)$	$\cos(x)$
cosine	$\cos(x)$	$-\sin(x)$
exponential	e^x	e^x
logarithm	$\log(x)$	$\frac{1}{x}$

Partial derivatives and the gradient

When our function has more than one input variable, we need to specify which variable we are using for derivation. For example, let's say we are measuring our elevation on a mountain as a function of our position. Our GPS position is defined by two variables: longitude and latitude, and therefore the elevation depends on two variables: $y = f(x_1, x_2)$.

We can calculate the **rate of change in elevation with respect to x_1 , and the rate of change with respect x_2** independently. These are called **partial derivatives**, because we only consider the change with respect to one variable. We will indicate them with a "curly d" symbol:

$$\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}$$

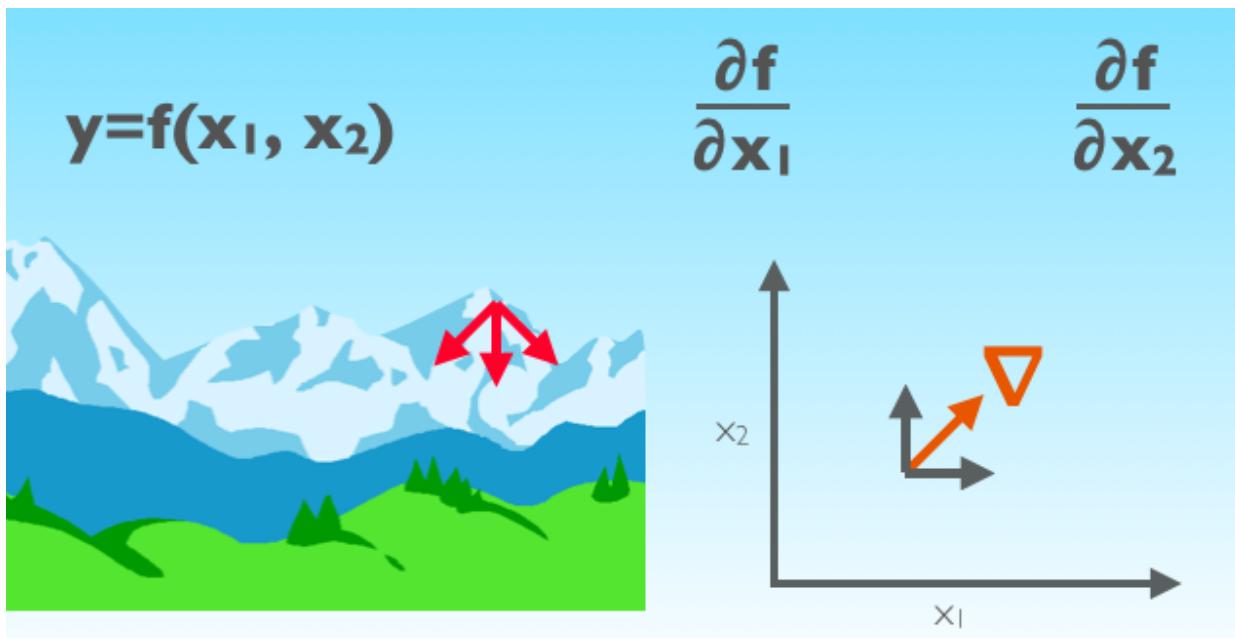
If we are on top of a hill the fastest route downhill will not necessarily be along any of the north-south or east-west directions, it will be in whatever direction the hill is more steeply descending down.

In the two dimensional plane of x_1 and x_2 , the direction of the most abrupt change will be a 2-dimensional vector whose components are the partial derivatives with respect to each variable. We call this vector the **Gradient**, and we indicate it with an inverted triangle called *del* or *nabla*: ∇ .

The gradient is an operation that takes a function of multiple variables and returns a vector. The components of this vector are all the partial derivatives of the function. Since the partial derivatives are functions of all variables, the gradient too is a function of all variables. To be precise, it is a vector function.

For each point (x_1, x_2) , the gradient returns the **vector in the direction of maximum steepness** in the graph of the original function. If we want to go downhill, all we have to do is walk in the direction opposite to the gradient. This will be our strategy for minimizing cost functions.

So we have an operation, the gradient, which takes a function of multiple variables and returns a vector in the direction of maximum steepness. Pretty cool!



Why is this neat? Why is it important? Well, it turns out that we can use this idea to train our networks.

BACKPROPAGATION INTUITION

Now that we have defined the gradient, let's talk about backpropagation.

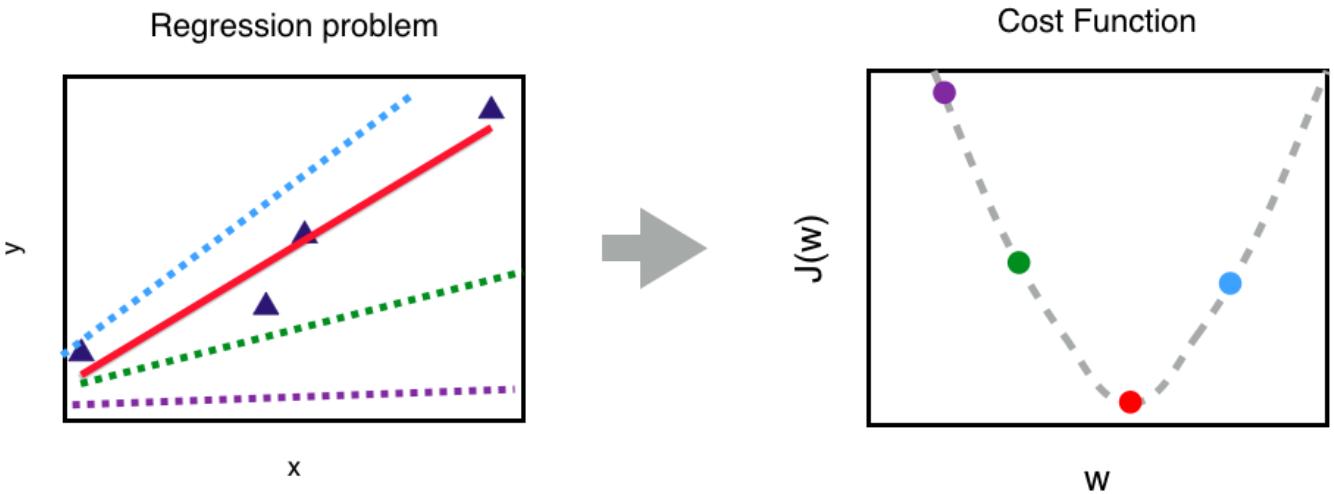
Backpropagation is a core concept in Machine Learning. The next several sections are dedicated to working through the math of backpropagation. As said at the beginning of this chapter, it is not necessary to understand the maths in order to be able to build and apply a deep learning model. However, the math is not very hard and with a little bit of exercise you'll be able to see that there is no mystery behind how Neural Networks function.

At a high-level, the backpropagation algorithm is a supervised learning method for training our networks. It uses the error between the model prediction and the true labels to modify the model weights in order to reduce the error in the next iteration.

The starting point for backpropagation is the [Cost Function](#) we have introduced in Chapter 3.

Let's consider a generic cost function of a network with just one weight, let's call this function $J(w)$. For every value of the weight w , the function calculates a value of the cost $J(w)$.

A different cost corresponds to each value of w



The figure shows this situation for the case of a Linear Regression. As seen in Chapter 3 different lines correspond to different values of w . In the figure we represented them with different colors. Each line

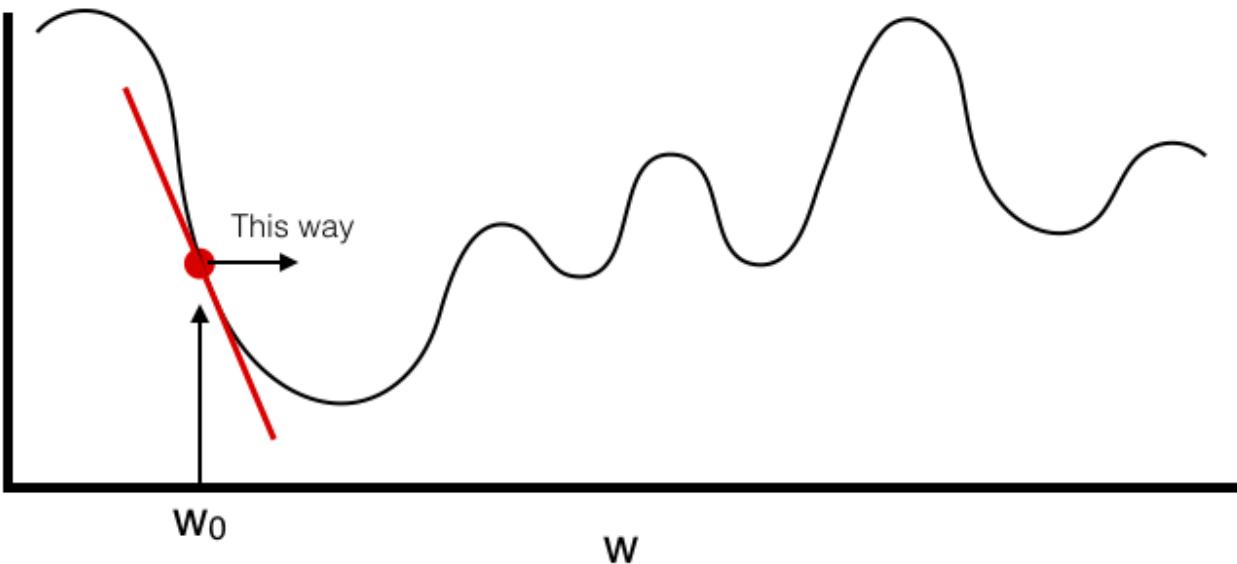
produces a different cost, here represented with a dot of different color and our goal is to find the value of w that corresponds to the minimum of $J(w)$.

The case of linear regression is easily solved with a bit of algebra, but how do we deal with the general case of a network with millions of weights and biases? The cost function J now depends on millions of parameters and it is not obvious how to search for a minimum value.

What is clear is that the shape of such a cost function is not going to be a smooth parabola like the one in the figure, and we will need a way to navigate a very complex landscape in search for a minimum value.

Let's say we are sitting at a particular point w_0 with corresponding cost $J(w_0)$, how do we move towards lower costs?

We would like to move in the direction of decreasing $J(w)$ until we reach the minimum, but we can only use local information. How do we decide where to go?



As we have seen when we talked about descending from a hill, the derivative indicates its slope at each point. So, in order to move towards lower values, we need to calculate the derivative at w_0 and then change our position by subtracting the value of the derivative from the value of our starting position w_0 .

Programmatically speaking, we can take one step in the direction where the descent algorithm is the lowest, i.e. the cost function is minimized.

Mathematically speaking, we can take one step following the rule:

$$w_0 - > w_0 - \frac{dJ}{dw}(w_0)$$

Let's check that this does move us towards lower values on the vertical axis.

If we are at w_0 like in the figure, the slope of the curve is negative and thus the quantity $-\frac{dJ}{dw}(w_0)$ is positive. So, the value of w_0 will increase, moving us towards the right on the horizontal axis.

The corresponding value on the vertical axis will decrease and we successfully moved towards a lower value of the function $f(w)$.

Vice versa, if we were to start at a point w_0 where the value of the slope is positive, we would subtract a positive quantity $\frac{dJ}{dw}(w_0)$ that is now negative. This would move w_0 to the left, and the corresponding values on the vertical axis would still decrease.

LEARNING RATE

The update rule we have just introduced one more modification. As it is, it suffers from two problems. If the cost function is very flat, the derivative will be very very small and with the current update rule, we will move very very slowly towards the minimum. Viceversa, if the cost function is very steep, the derivative will be very large and we might end up jumping beyond the minimum.

A simple solution to both problems is to introduce a tunable knob that allows us to decide how big should be the step to take in the direction of the gradient. This is called **learning rate**, and we will indicate it with the Greek letter η :

$$w_0 \rightarrow w_0 - \eta \frac{dJ}{dw}(w_0)$$

If we choose a small learning rate, we will move by tiny steps, if we choose a large learning rate, we will move by large steps.

However, we must be careful. If the learning rate is too large, we will actually run away from the solution. At each new step we move towards the direction of the minimum, but since the step is too large, we overshoot and go beyond the minimum, at which point we reverse course and repeat, going further and further away.

GRADIENT DESCENT

This way of looking for the minimum of a function is called **Gradient Descent** and it is the idea behind backpropagation. Given a function, we can move towards its minimum by following the path indicated by its derivative, or in the case of multiple variables, indicated by the gradient.

For a neural network, we define a cost function that depends on the values of the parameters, and we find the values of the parameters by minimizing such cost through gradient descent.

The **cost function** is the method for how we can optimize our networks. In fact, it's the backbone for a lot of different machine learning and deep learning techniques.

All we are really doing is taking the cost function, calculating its partial derivatives with respect to each parameter, and then using update rule to decrease the cost. We do this by subtracting the value of the negative gradient from the parameters themselves. This is what's called a parameter update.

GRADIENT CALCULATION IN NEURAL NETWORKS

Let's recap what we've learned so far.

We know that the gradient is a function that indicates the direction of maximum steepness. We also know that we can move towards the minimum of a function by taking consecutive steps in the direction of the gradient at each point we visit.

Let's see this with a programming example. We'll use an invented cost function. Let's start by defining an array x with 100 points in the interval $[-4, 4]$:

```
x = np.linspace(-4, 4, 100)
```

Then let's define an invented cost function $J(w)$ that depends on w in some weird way.

$$J(w) = 70.0 - 15.0w^2 + 0.5w^3 + w^4$$

```
def J(w):
    return 70.0 - 15.0*w**2 + 0.5*w**3 + w**4
```

Using the table of derivatives presented earlier we can also quickly calculate its derivative.

$$\frac{dJ}{dw}(w) = -30.0w + 1.5w^2 + 4w^3$$

```
def dJdw(w):
    return -30.0*w + 1.5*w**2 + 4*w**3
```

Let's plot both functions:

```
plt.subplot(211)
plt.plot(x, J(x))
plt.title("J(w)")

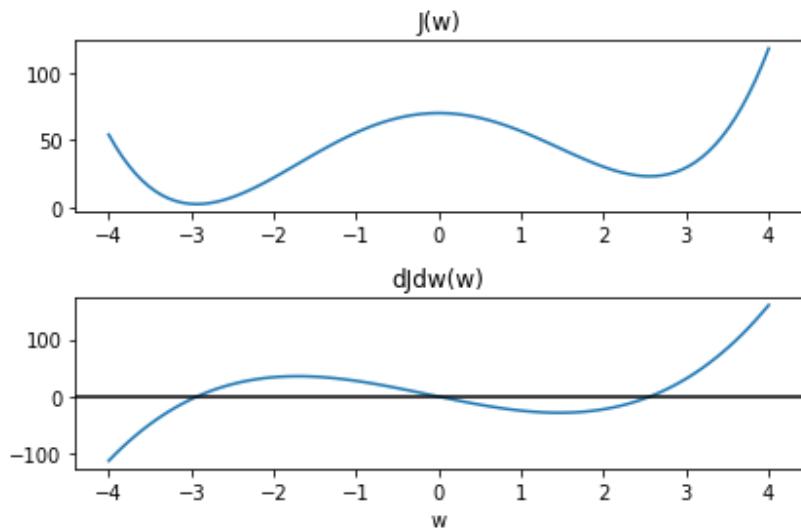
plt.subplot(212)
plt.plot(x, dJdw(x))
plt.axhline(0, color='black')
```

```

plt.title("dJdw(w)")
plt.xlabel("w")

plt.tight_layout()
plt.show()

```



Now let's find the minimum value of $J(w)$ by gradient descent. The function we have chosen has two minima, one is a local minimum, the other is the global minimum. If we apply plain gradient descent we will stop at the minimum that is nearest to where we started. Let's keep this in mind for later.

Let's start from a random initial value of $w_0 = -4$:

```
w0 = -4
```

and let's apply the update rule:

$$w_0 \leftarrow w_0 - \eta \frac{dJ}{dw}(w_0)$$

We will choose a small learning rate of $\eta = 0.001$ initially:

```
lr = 0.001
```

The update step is:

```
step = lr * dJdw(w0)
step
```

-0.112

and the new value of w_0 is:

```
w0 - step
```

-3.888

i.e. we moved to the right, towards the minimum!

Let's do 30 iterations and see where we get:

```
iterations = 30

w = w0

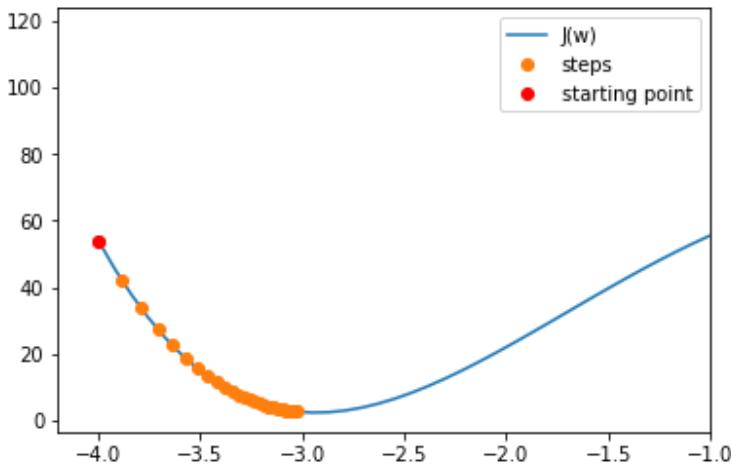
ws = [w]

for i in range(iterations):
    step = lr * dJdw(w)
    w -= step
    ws.append(w)

ws = np.array(ws)
```

Let's visualize our descent, zooming in the interesting region of the curve:

```
plt.plot(x, J(x))
plt.plot(ws, J(ws), 'o')
plt.plot(w0, J(w0), 'or')
plt.legend(["J(w)", "steps", "starting point"])
plt.xlim(-4.2, -1)
plt.show()
```



As you can see, we proceed with small steps towards the minimum, and there we stop. Try to modify the starting point and re-run the code above to fully understand how this works.

Why is this relevant to Neural Networks?

Remember that a Neural Network is just a function that connects our inputs X to our outputs y . We'll refer to this function as $\hat{y} = f(X)$. This function depends on a set of weights w that modulate the output of a layer when transferring it to the next layer, and on a set of biases b .

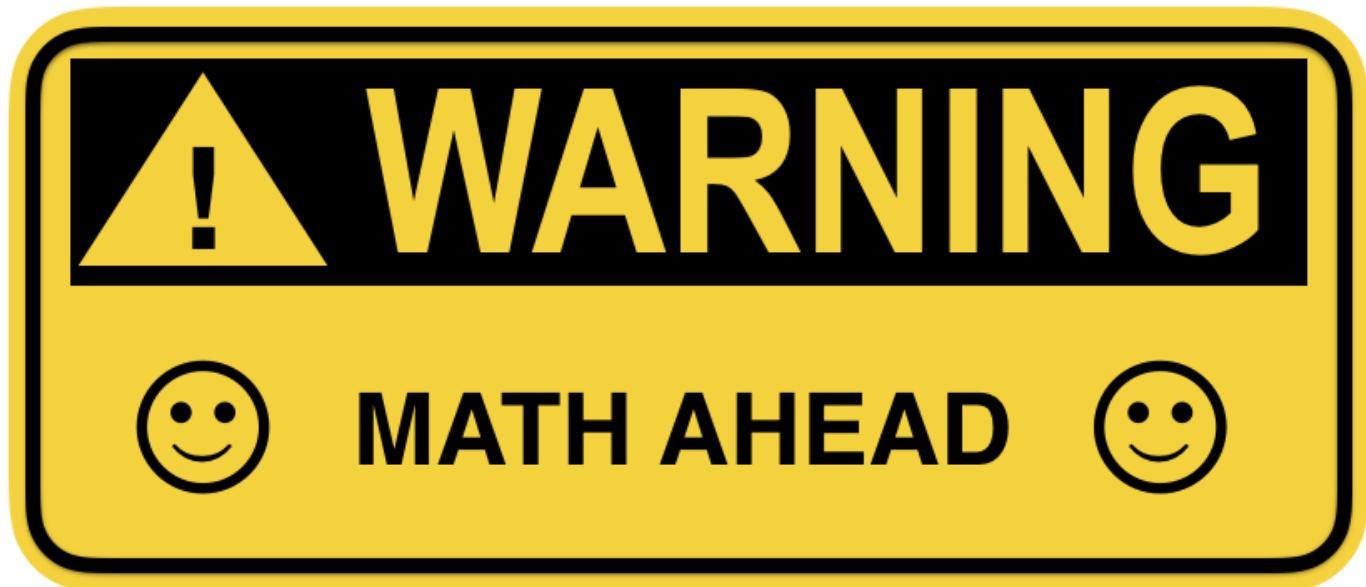
Also, remember that we defined a cost $J(\hat{y}, y) = J(f(X, w, b), y)$ that is calculated over the training set. So, for fixed training data, the cost J is a function of the parameters w and b .

The best model is the one that minimizes the cost. We can therefore **use gradient descent on the cost function** to update the values of the parameters w and b . The gradient will tell us in which direction to update our parameters, and it is crucial to learning the optimal values of our network parameters.

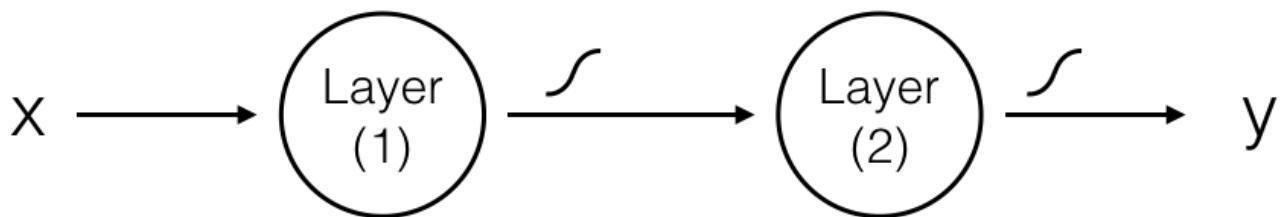
First we calculate the gradient with respect to each weight (and bias): $\frac{\partial J}{\partial w}$ and then we update each weight using the learning rate we have just introduced: $w_0 \rightarrow w_0 - \eta \frac{\partial J}{\partial w}$.

All we need to do at this point is to learn how to calculate the gradient $\frac{\partial J}{\partial w}$.

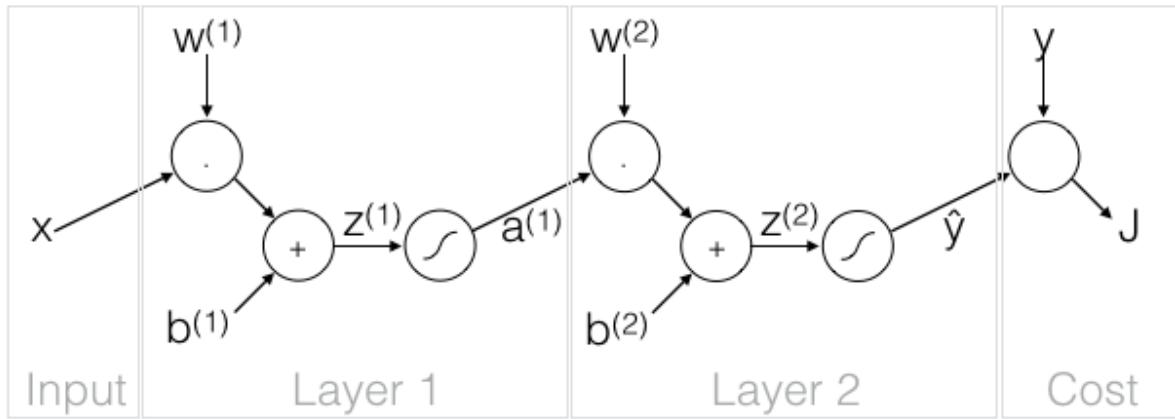
THE MATH OF BACKPROPAGATION



In this section we will work through the calculation of the gradient for a very simple Neural Network. We are going to use equations and maths. As said previously, feel free to skim through this part if you're focused on applications, you can always come back later to go deeper in the subject. We will start with a network with only one input, one inner node and one output. This will make our calculations easier to follow.



In order to make the math easier to follow we will break down this graph and highlight the operations involved:



Starting from the left, the input is multiplied with the first weight $w^{(1)}$, then the bias $b^{(1)}$ is added and the sigmoid activation function is applied. This completes the first layer. Then we multiply the output of the first layer by the second weight $w^{(2)}$, we add the second bias $b^{(2)}$ and we apply another sigmoid activation function. This gives us the output \hat{y} . Finally we use the output \hat{y} and the labels y to calculate the cost J .

Forward Pass

Let's formalize the operations described above with math. The forward pass equations are written as follows:

$$z^{(1)} = xw^{(1)} + b^{(1)}$$

$$a^{(1)} = \sigma(z^{(1)})$$

$$z^{(2)} = a^{(1)}w^{(2)} + b^{(2)}$$

$$\hat{y} = a^{(2)} = \sigma(z^{(2)})$$

$$J = J(\hat{y}, y)$$

The input-sum $z^{(1)}$ is obtained through a linear transformation of the input x with weight $w^{(1)}$ and bias $b^{(1)}$. In this case we only have one input, so there really is no weighted "sum", but we still call it input-sum to remind ourselves of the general case where multiple inputs and multiple weights are present.

The activation $a^{(1)}$ is obtained by applying the sigmoid function to the input-sum $z^{(1)}$. This is indicated by that letter σ (pronounced *sigma*). A similar set of equations holds for the second layer with input-sum $z^{(2)}$ and activation $a^{(2)}$, which is equivalent to our predicted output in this case.

The cost function J is a function of the true labels y and the predicted values \hat{y} , which contain all the parameters of the network.

The equations described above allow us to calculate the prediction of the network for a given input and the cost associated with such prediction. Now we want to calculate the gradients in order to update the weights and biases and reduce the cost.

Weight updates

Our goal is to calculate the derivative of the cost function with respect to the parameters of the model, i.e. weights and biases. Let's start by calculating the derivative of the cost function with respect to $w^{(2)}$, the last weight used by the network.

$$\frac{\partial J}{\partial w^{(2)}}$$

$w^{(2)}$ appears inside $z^{(2)}$, which is itself inside the sigmoid function, so we need a way to calculate the derivative of a nested function.

The technique is actually pretty easy and it's called **chain rule**. If you need a refresher of how it works, we have an example of this in the [Appendix](#).

We can look at the graph above to determine which terms will appear in the chain rule and see that J depends on $w^{(2)}$ through \hat{y} and $z^{(2)}$.

If we apply the chain rule we see that this derivative is the product of *three terms*.

$$\frac{\partial J}{\partial w^{(2)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}}$$

Wow! This may look to you pretty complicated! Wouldn't it be nice to have a simplified notation for all this? It turns out we can introduce this simpler notation, following the course by Roger Grosse at University of Toronto.

In particular we will use a long line over a variable to indicate the *derivative of the cost function with respect to that variable*. E.g.:

$$\overline{w^{(2)}} := \frac{\partial J}{\partial w^{(2)}}$$

Besides being easier to read, this notation emphasizes the fact that those derivatives are evaluated at a certain point, i.e. they are numbers, not functions.

Using this notation, we can rewrite the above equation as:

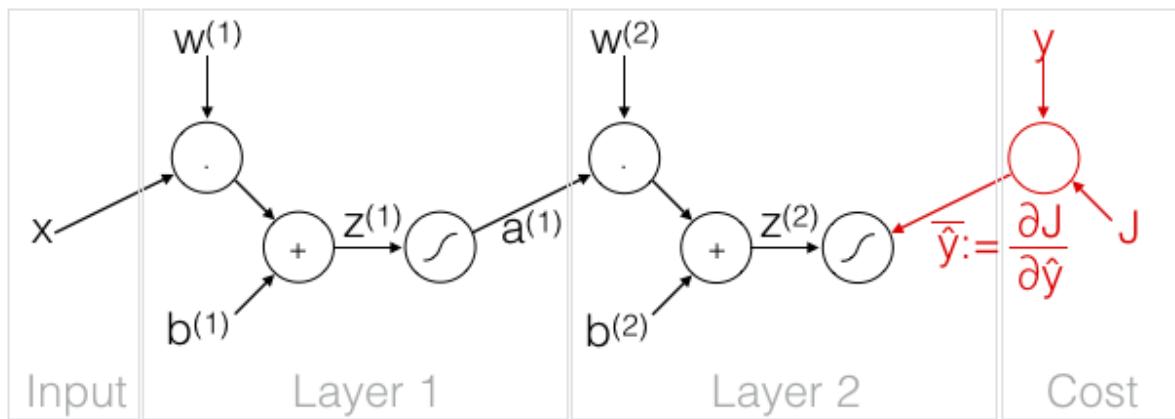
$$\overline{w^{(2)}} = \overline{z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}} = \overline{\hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}}$$

And we can start to see why it is called backpropagation: in order to get to calculate $w^{(2)}$ we will need to first calculate the derivatives of the terms that follow $w^{(2)}$ in the graph, and then propagate their contributions back to calculate $w^{(2)}$.

Step 1: $\hat{y} = \frac{\partial J}{\partial \hat{y}}$

The first term is just the derivative of the cost function with respect to \hat{y} . This term will depend on the exact form of the cost function, but it is well defined, and it can be calculated for a given training set. For example, in the case of the Mean Squared Error $\frac{1}{2}(\hat{y} - y)^2$ this term is simply: $(\hat{y} - y)$.

Looking at the graph above, we can highlight in red the terms involved in the calculation of \hat{y} which is only the labels and the predictions:

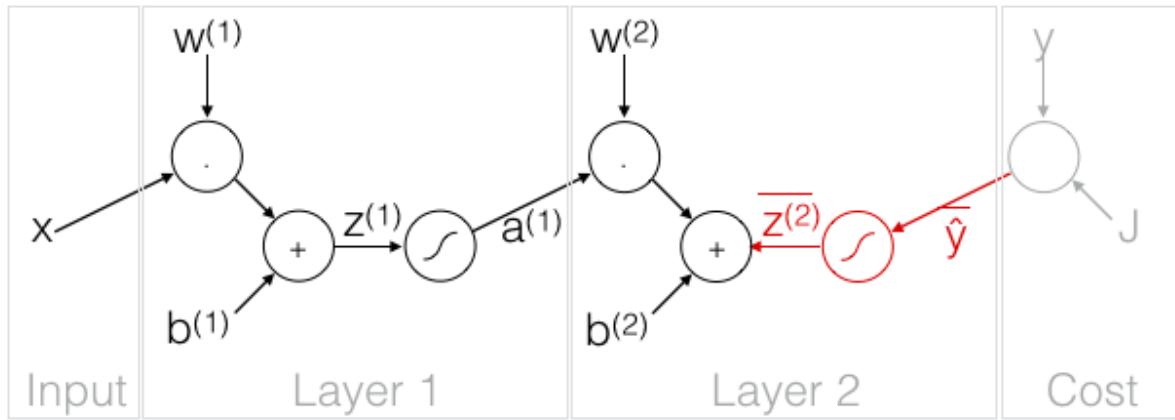


Step 2: $z^{(2)} = \frac{\partial J}{\partial z^{(2)}}$

As noted before, the chain rule tells us that $z^{(2)}$ is the product of the derivative of the sigmoid with the term we just calculated \hat{y} :

$$\overline{z^{(2)}} = \overline{\hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} = \overline{\hat{y}} \sigma'(z^{(2)})$$

Notice how information is propagating backwards in the graph:



Since we have already calculated \hat{y} we don't need to calculate it again, the only term we need is the derivative of the sigmoid. This is easy to calculate and we'll just indicate it with σ' .

$$\text{Step 3: } w^{(2)} = \frac{\partial J}{\partial w^{(2)}}$$

Now we can calculate $w^{(2)}$.

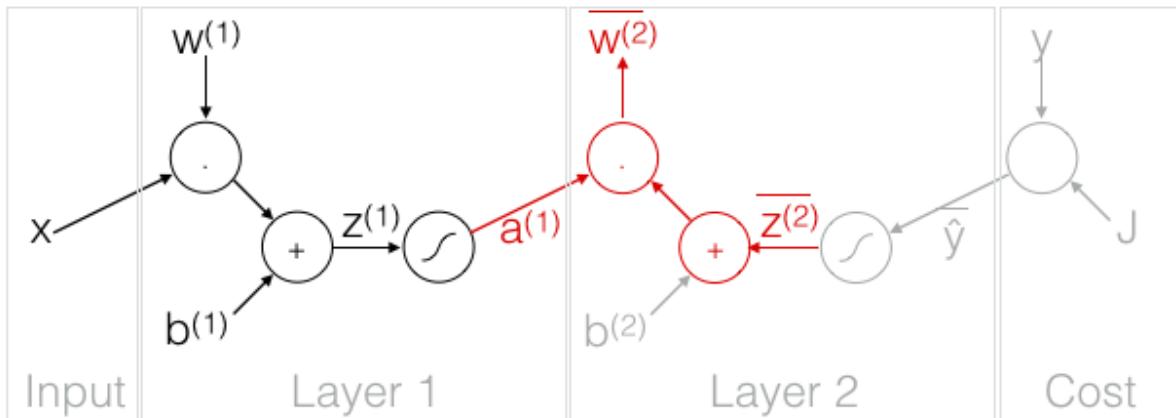
Looking at the formulas above, we know that:

$$\overline{w^{(2)}} = \overline{z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(2)}}$$

Since we have already calculated $z^{(2)}$ we only need to calculate $\frac{\partial z^{(2)}}{\partial w^{(2)}}$, which is equal to $z^{(2)} a^{(1)}$

So we have:

$$\overline{w^{(2)}} = \overline{z^{(2)} z^{(2)}} a^{(1)}$$



This last formula is really interesting because it tells us that the update to the weights $w^{(2)}$ is proportional to the input $a^{(1)}$ received by those weights.

This equation sometimes is also written as:

$$\overline{w^{(2)}} = \delta^{(2)} a^{(1)}$$

where $\delta^{(2)}$ is calculated using parts of the network that are downstream with respect to $w^{(2)}$ and it corresponds to the derivative of the cost with respect to the input sum $z^{(2)}$.

The important aspect here is that $\delta^{(2)}$, i.e. $z^{(2)}$, is a constant, representing the downstream contribution of the network to the error.

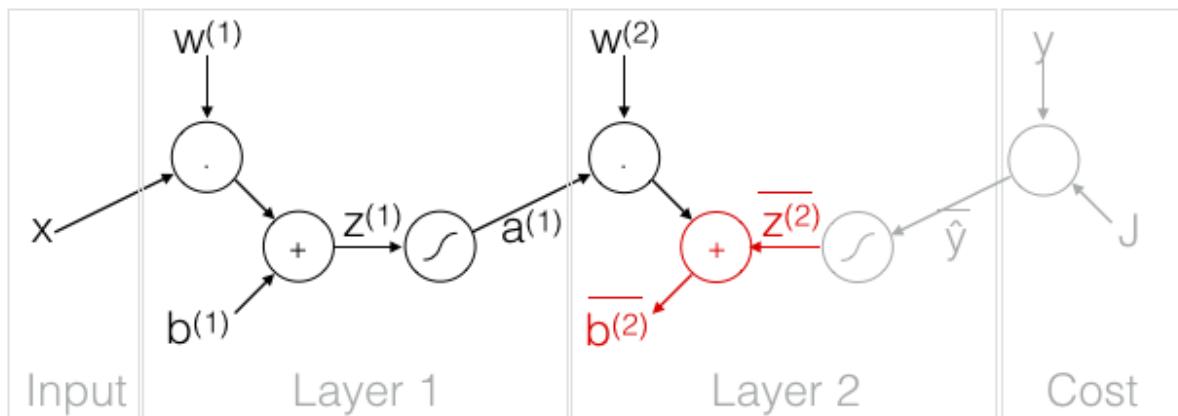
Using the same procedure we can calculate the corrections to the bias $b^{(2)}$ as well:

$$\text{Step 4: } b^{(2)} = \frac{\partial J}{\partial b^{(2)}}$$

We can apply the chain rule again and obtain:

$$\overline{b^{(2)}} = \overline{z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}} = \overline{z^{(2)}}$$

Since the $\frac{\partial z^{(2)}}{\partial b^{(2)}} = 1$

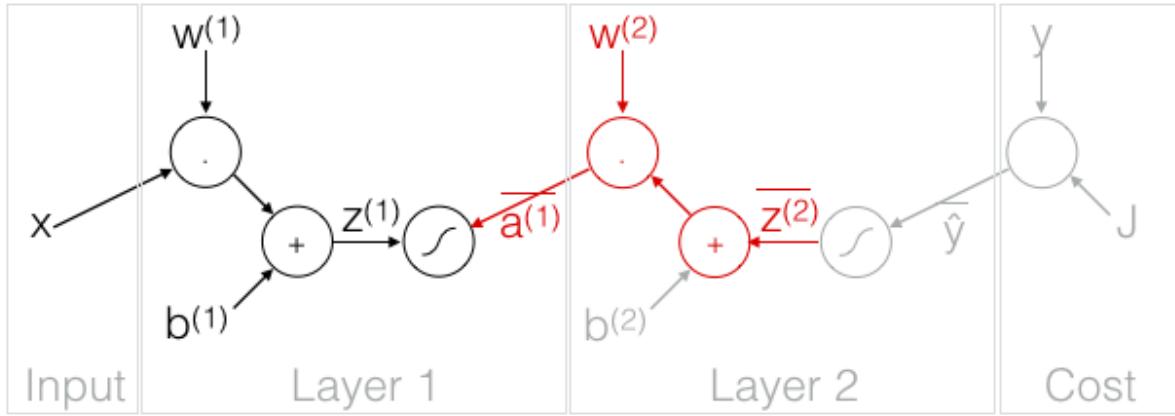


Following a similar procedure we can keep propagating the error back and calculate the corrections to $w^{(1)}$ and $b^{(1)}$. Proceeding backwards, the next term we need to calculate is $a^{(1)}$.

$$\text{Step 5: } a^{(1)} = \frac{\partial J}{\partial a^{(1)}}$$

Looking at the formulas for the forward pass we notice that $a^{(1)}$ appears inside $z^{(2)}$, so we apply the chain rule and obtain:

$$\overline{a^{(1)}} = \overline{z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} = \overline{z^{(2)}} w^{(2)}$$



At this point the calculation of the other terms is mechanical, and we will just summarize them all here:

$$\begin{aligned}\overline{\hat{y}} &= \frac{\partial J}{\partial \hat{y}} \\ \overline{z^{(2)}} &= \overline{\hat{y}} \sigma'(z^{(2)}) \\ \overline{b^{(2)}} &= \overline{z^{(2)}} \\ \overline{w^{(2)}} &= \overline{z^{(2)}} a^{(1)} \\ \overline{a^{(1)}} &= \overline{z^{(2)}} w^{(2)} \\ \overline{z^{(1)}} &= \overline{a^{(1)}} \sigma'(z^{(1)}) \\ \overline{b^{(1)}} &= \overline{z^{(1)}} \\ \overline{w^{(1)}} &= \overline{z^{(1)}} x\end{aligned}$$

As you can see each term relies on previously calculated terms, which means we don't have to calculate them twice. This is why it's called **backpropagation**: because the error terms are propagated back starting from the cost function and walking along the network graph in reverse order.

Wow! What a journey! Congratulations! you have completed the hardest part. We hope this was insightful and useful. In the next section we will extend these calculations to fully connected networks where there are

many nodes for each layer. As you will see, it's basically the same thing, only we will deal with matrices instead of just numbers.

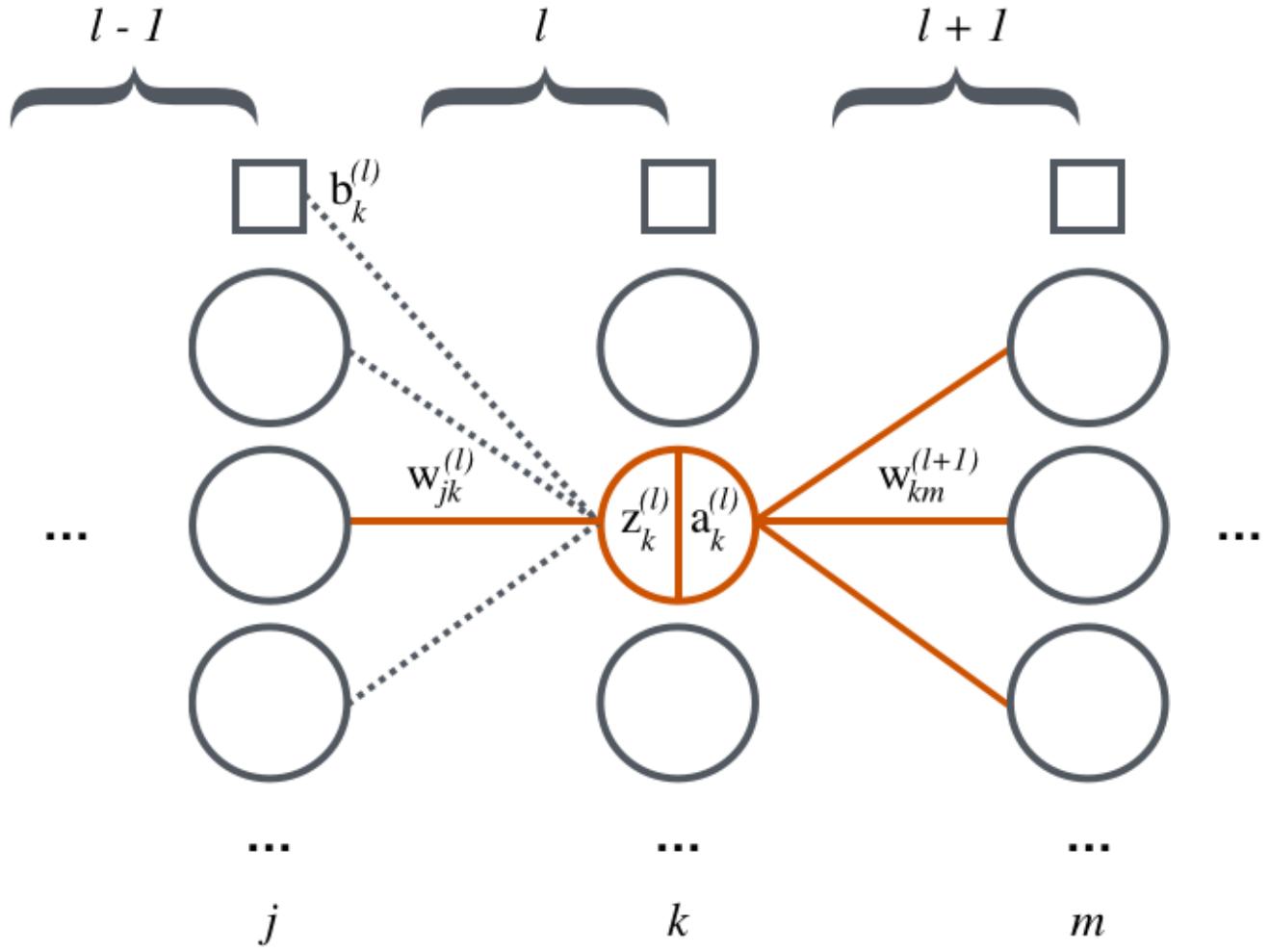
FULLY CONNECTED BACKPROPAGATION

Let's see how we can expand the calculation to a fully connected Neural Network.

In a fully connected network, each layer contains several nodes, and each node is connected to all of the nodes in the previous and in the next layers. The weights in layer l can be organized in a matrix $W^{(l)}$ whose elements are identified by two indices j and k . The index k indicates the receiving node and the index j indicates the emitting node. So, for example, the weight connecting node 5 in layer 2 to node 4 in layer 3 is going to be indicated as $w_{54}^{(3)}$ etc.

The input sum at layer l and node k , $z_k^{(1)}$ is the weighted sum of the activations of layer $l - 1$ plus the bias term of layer l :

$$z^{(l)}_k = \sum_j a^{(l-1)}_j w^{(l)}_{jk} + b^{(l)}_k$$



Forward Pass

The forward pass equations can be written as follows:

$$z_k^{(l)} = \sum_j a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)}$$

$$a_k^{(l)} = \sigma(z_k^{(l)})$$

The activations \$a_k^{(l)}\$ are obtained by applying the sigmoid function to the input-sums \$z_k^{(l)}\$ coming out from node \$k\$ at layer \$l\$.

Let's indicate the last layer with the capital letter \$L\$. The equations for the output are:

$$z_s^{(L)} = \sum_r a_r^{(L-1)} w_{rs}^{(L)} + b_s^{(L)}$$

$$\hat{y}_s = \sigma(z_s^{(L)})$$

$$J = \sum_s J(\hat{y}_s, y_s)$$

The cost function J is a function of the true labels y and the predicted values \hat{y} , which contain all the parameters of the network. We indicated it with a sum to include the case where more than one output node is present.

If the above formulas are hard to read in maths, here's a code version of them. We allocate an array w with random values for the weights $w_{jk}^{(l)}$. In this particular example, imagine a set of weights connecting a layer with 4 units to a layer with 2 units:

```
w = np.array([[ -0.1,  0.3],
              [ -0.3, -0.2],
              [ 0.2,  0.1],
              [ 0.2,  0.8]])
```

We also need an array for the biases, with as many elements as there are units in the receiving layer, i.e. 2:

```
b = np.array([0., 0.])
```

The output of the layer with 4 elements is represented by the array a , whose elements are $a_j^{(l-1)}$:

```
a = np.array([0.5, -0.2, 0.3, 0.])
```

Then, the layer l performs the operation:

```
z = np.dot(a, w) + b
```

returning the array of z with elements $z_k^{(l)} = \sum_j a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)}$:

```
z
```

```
array([0.07, 0.22])
```

z is indexed by the letter k . There are 2 entries, one for each of the units in the receiving layer. Similarly you can write code examples for the other equations.

Backpropagation

Although they may seem a bit more complicated, the only thing that changed is that now each node takes multiple inputs, each with its own weight and so the input sums z are actually summing up the contributions of the nodes in the previous layer.

The backpropagation formulas are calculated as before. Here is a summary of all of the terms:

$$\begin{aligned}\overline{\hat{y}_s} &= \frac{\partial J}{\partial \hat{y}_s} \\ \overline{z_s^{(L)}} &= \overline{\hat{y}_s} \sigma'(z_s^{(L)}) \\ \overline{b_s^{(L)}} &= \overline{z_s^{(L)}} \\ \overline{w_{rs}^{(L)}} &= \overline{z_s^{(L)}} a_r^{(L-1)} \\ &\dots \\ &\dots \\ \overline{a_k^{(l)}} &= \sum_m w_{km}^{(l+1)} \overline{z_m^{(l+1)}} \\ \overline{z_k^{(l)}} &= \overline{a_k^{(l)}} \sigma'(z_k^{(l)}) \\ \overline{b_k^{(l)}} &= \overline{z_k^{(l)}} \\ \overline{w_{jk}^{(l)}} &= \overline{z_k^{(l)}} a_j^{(l-1)}\end{aligned}$$

These equations are equivalent to the ones for the unidimensional case, with only **one major difference**.

The term $a_k^{(l)}$, indicating the change in cost due to the activation at node k in layer l needs to take into account all the errors in the nodes downstream at layer $l + 1$. Since the activation $a_k^{(l)}$ is part of the input of each node in the next layer $l + 1$, we have to apply the chain rule to each of them and sum all their contributions together.

Everything else is pretty much the same as the unidimensional case, with just a bunch of indices to keep track of.

MATRIX NOTATION

We can simplify the above notation a bit by using vectors and matrices to indicate all the ingredients in the network.

Forward Pass

The equations for the forward pass read:

$$\begin{aligned} \dots \\ \mathbf{z}^{(l)} &= \mathbf{a}^{(l-1)} \mathbf{W}^{(l)} + \mathbf{b}^{(l)} \\ \mathbf{a}^{(l)} &= \sigma(\mathbf{z}^{(l)}) \\ \dots \end{aligned}$$

Backpropagation

The equations for the backpropagation read:

$$\begin{aligned} \dots \\ \overline{\mathbf{a}^{(l)}} &= \mathbf{W}^{(l+1) T} \overline{\mathbf{z}^{(l+1)}} \\ \overline{\mathbf{z}^{(l)}} &= \overline{\mathbf{a}^{(l)}} \odot \sigma'(\mathbf{z}^{(l)}) \\ \overline{\mathbf{b}^{(l)}} &= \overline{\mathbf{z}^{(l)}} \\ \overline{\mathbf{W}^{(l)}} &= \overline{\mathbf{a}^{(l-1)} \mathbf{z}^{(l) T}} \\ \dots \end{aligned}$$

Circle dot indicates the element-wise product and it is also called **Hadamard product**, whereas when we have two matrices next to each other, we indicate the matrix multiplication is taking place.

So we can summarize the backpropagation algorithm as follows:

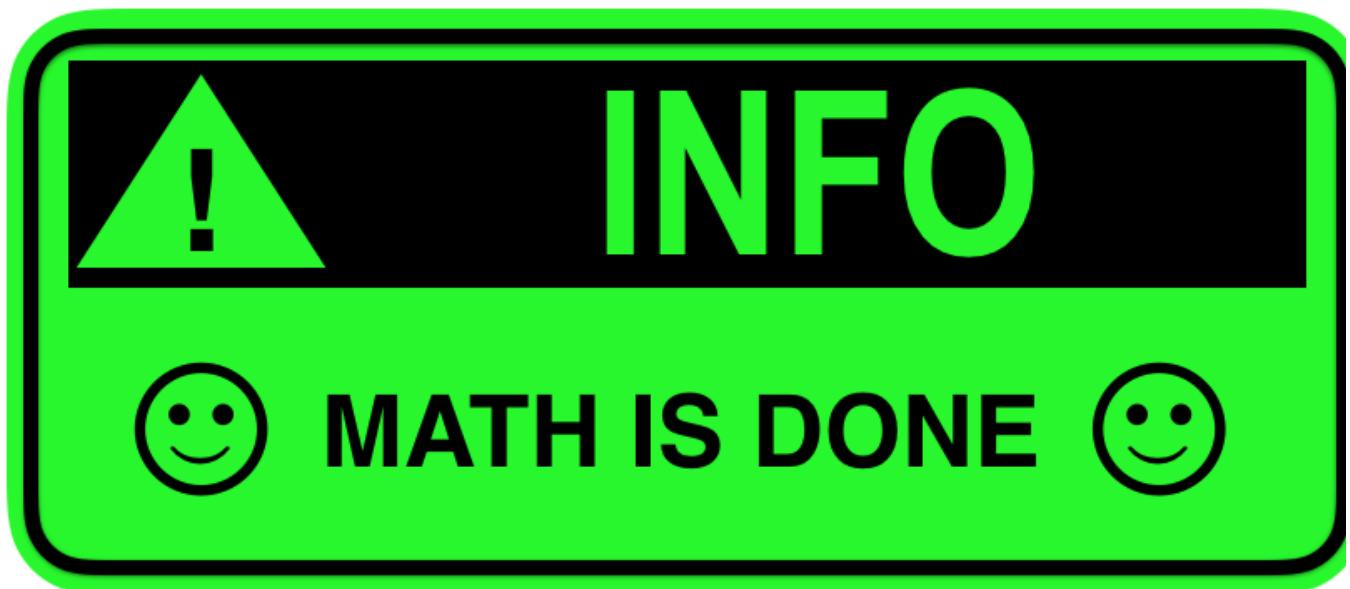
1. Forward pass: we calculate the input-sum and activation of each neuron proceeding from input to output.
 - We calculate the error signal of the final layer, by obtaining the gradient of the cost function with respect to the outputs of the network. This expression will depend on the training data and training labels, as well as the chosen cost function, but it is well defined for given training data and cost.

- We propagate the error backwards at each operation by taking into account the error signals at the outputs affected by that operation as well as the kind of operation performed by that specific node.
- We proceed back till we get to the weights multiplying the input, at which point we are done.

A couple of observations:

- The gradient of the cost function with respect to the weights is a matrix with the same shape as the weight matrix.
- The gradient of the cost function with respect to the biases is a vector with the same shape as the biases.

Congratulations! You've now gone through the back propagation algorithm and hopefully see that **it's just many matrix multiplications**. The bigger the network, the bigger your matrices will be and so the larger the matrix multiplication products. We will go back to this in a few sections. For now, give yourself a pat on the back: Neural Networks have no more mysteries for you!



GRADIENT DESCENT

How do backpropagation and gradient descent work in practice in Deep Learning? Let's use a real world dataset to explore how this is done in detail.

Let's say you've just been hired by the government for a very important task. A group of counterfeiters is using fake banknotes and this is creating all sorts of problems. Luckily your colleague Agent Jones managed to get hold of a stack of fake banknotes and bring them to the lab for inspection. You've scanned true and fake notes and extracted four spectral features. Let's build a classifier that can distinguish them.



First of all let's load and inspect the dataset:

```
df = pd.read_csv('.../data/banknotes.csv')
df.head()
```

	variance	skewness	kurtosis	entropy	class
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

The four features come from the images (see [UCI database](#) for details) and they are like a *fingerprint* of each image. Another way to look at it is to say that feature engineering has already been done and we have now 4 numbers representing the relevant properties of each image. The `class` column indicates if a banknote is true or fake, with 0 indicating true and 1 indicating fake.

Let's see how many banknotes we have in each class:

```
df['class'].value_counts()
```

```
0    762
1    610
Name: class, dtype: int64
...
Name: class, dtype: int64
...
Name: class, dtype: int64
```

We can also calculate the fraction of the larger class by dividing the first row by the total number of rows:

```
df['class'].value_counts()[0]/len(df)
```

```
0.5553935860058309
```

The larger class amounts to 55% of the total, so we if we build a model it needs to have an accuracy superior to 55% in order to be useful.

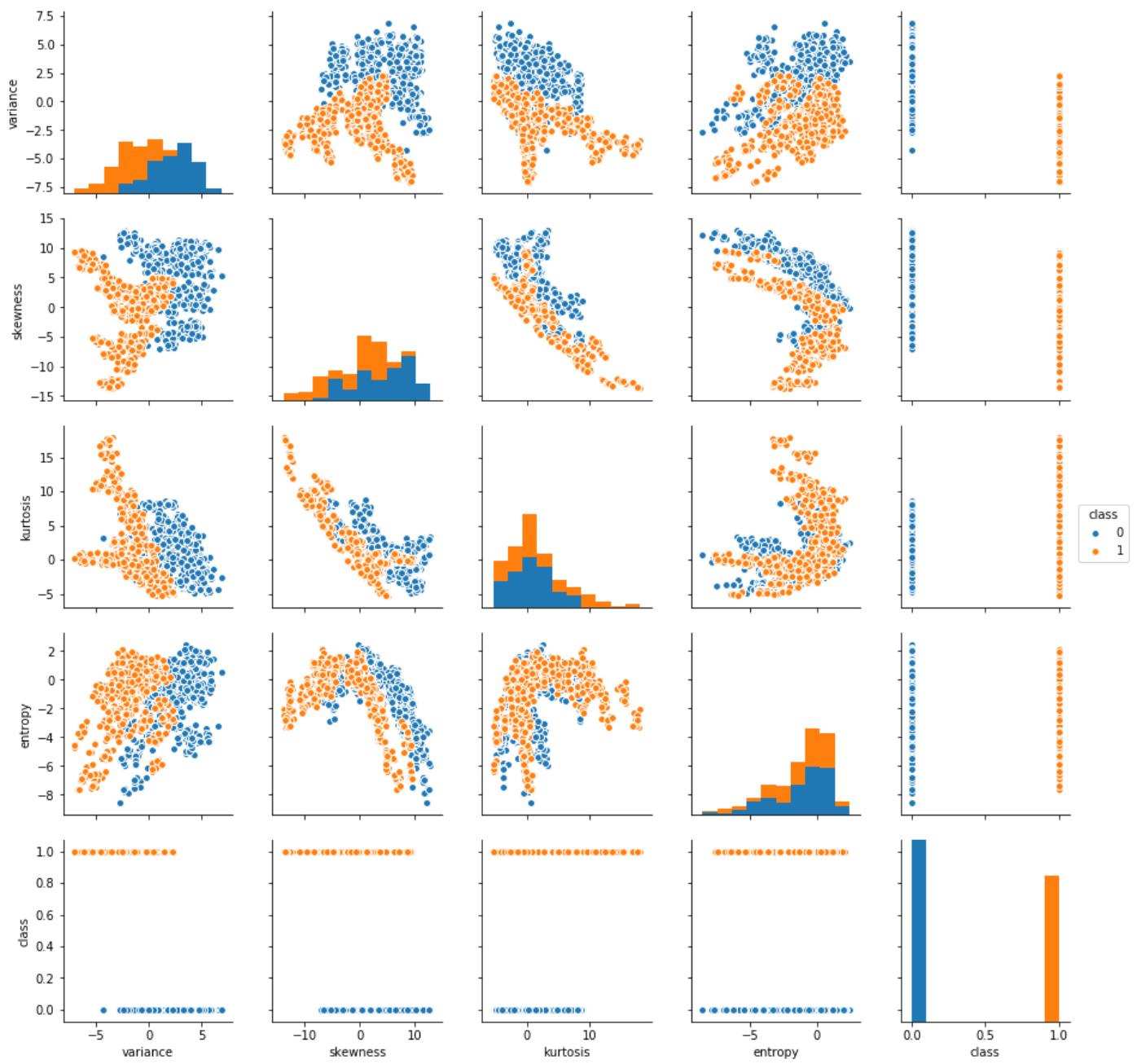
Let's use `seaborn.pairplot` for a quick visual inspection of the data. First we load the library:

```
import seaborn as sns
```

Then we plot the whole dataset using a `pairplot` like we did for the Iris flower dataset in the previous chapters. A pairplot allows us to look at how pairs of features are correlated, as well as how each feature is correlated with the labels. Also, a pairplot displays the histogram of each feature along the diagonal and we can use the `hue` parameter to color the data using the labels. Pretty nice!

```
sns.pairplot(df, hue="class")
```

```
<seaborn.axisgrid.PairGrid at 0x7fae700702e8>
```



We can see from the plot that the two sets of banknotes seem quite well separable. In other words the orange and the blue scatters are not completely overlapped. This induces us to think that we will manage to build a good classifier and bust the counterfeiters.

Let's start by building a reference model using `Scikit-Learn`. As we have seen in [Chapter 3, Scikit-Learn](#) is a great Machine Learning library for Python. It implements many classical algorithms like **Decision Trees**, **Support Vector Machines**, **Random Forest** and more. It also has many preprocessing and model evaluation routines, so we strongly encourage you to learn to use it well.

For the purpose of this Chapter, we would like a model that trains fast, that does not require too much pre-processing and feature engineering and that is known to give good results.

Luckily for us such model exists and it's called **Random Forest**.

Random Forest

Random Forest is an ensemble learning method for classification, regression and other tasks, that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. You can think of it as a Decision Tree on steroids!

Scikit Learn provides `RandomForestClassifier` ready to use in the `sklearn.ensemble` module.

For the purpose of this Chapter it is not fundamental that you understand the internals of how the Random Forest classifier works. The important point here is that it's a model that works quite well and so we will use it for comparison.

Let's start by loading it:

```
from sklearn.ensemble import RandomForestClassifier
```

and let's create an instance of the model with default parameters:

```
model = RandomForestClassifier()
```

Now let's separate the features from labels as usual:

```
X = df.drop('class', axis=1).values
y = df['class'].values
```

and we are ready to train the model. In order to be quick and effective in judging the performance of our model we will use a 3-fold cross validation as done many times in [Chapter 3](#). First we load the `cross_val_score` function:

```
from sklearn.model_selection import cross_val_score
```

And then we run it with the model, features and labels as arguments. This function will return 3 values for the test accuracy, one for each of the 3 folds.

```
cross_val_score(model, X, y)
```

```
array([0.99344978, 0.97811816, 0.99562363])
```

The Random Forest model seems to work really well on this dataset. We obtain an accuracy score higher than 99% with a 3-fold cross-validation. This is really good and it also shows us how in some cases traditional ML methods are very fast and effective solutions.

We can also get the score on a train/test split fixed set in order to compare it later with a Neural Network based model.

```
from sklearn.model_selection import train_test_split
```

Let's split up our data using the `train_test_split` function:

```
x_train, x_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=42)
```

Let's train our model and check the accuracy score now:

```
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

```
0.9951456310679612
```

The accuracy on the test set is still very high.

Logistic Regression Model

Let's build a Logistic Regression model in Keras and train it. As we have seen, the parameters of the model are updated using the gradient calculated from the cost function evaluated on the training data.

$$\frac{d J(y, \hat{y}(w, X))}{d w}$$

X and y here, indicate a pair of training features and labels.

In principle, we could feed the training data one point at a time. For each pair of features and label, calculate the cost and the gradient and update the weights accordingly. This is called **Stochastic Gradient Descent** (also SGD). Once our model has seen each training data once, we say that an **Epoch** has completed, and we start again from the first training pair with the following epoch. Let's manually run one epoch on this simple model.

Then let's create a model as we have done in the previous chapters.

Notice that since this is a Logistic Regression we will only have one Dense layer, with an output of 1 and a sigmoid activation function. By now you should be very familiar with all this, but in case you have doubts you may go back to [Chapter 4](#) where we explained `Dense` layers in more detail.

Let's start with a few imports:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/_init__.py:36: Fu
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

And then let's define the model. We will initialize the weights to one for this time, using the `kernel_initializer` parameter. It's not a good initialization, but it will guarantee that we all get the same results without any artifacts due to random initialization:

```
model = Sequential()
model.add(Dense(1, kernel_initializer='ones', input_shape=(4,), activation='sigmoid'))
```

Then we compile the model as usual. Notice that, since we only have 1 output node with a `sigmoid` activation, we will have to use the `binary_crossentropy` loss, also introduced in [Chapter 4](#).

TIP: As a reminder, binary crossentropy has the formula:

$$J(\hat{y}, y) = - (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

and it can be implemented in code as:

```
def binary_crossentropy(y, y_hat):
    if y == 1:
        return - np.log(y_hat)
    else:
        return - np.log(1 - y_hat)
```

We compile the model using the `sgd` optimizer, which stands for **Stochastic Gradient Descent**. We will discuss this optimizer along with other more powerful ones [later in this chapter](#), so stay tuned.

Finally, we will compile the model requesting that the `accuracy` metric is also calculated at each iteration.

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

Finally we save the random weights so that we can always reset the model to this starting point.

```
weights = model.get_weights()
```

The method `.train_on_batch` performs a single gradient update over one batch of samples, so we can use it to train the model on a single data point at a time and then visualize how the loss changes at each point.

Normally we train models one batch at a time, passing several points at once and calculating the average gradient correction. The next plot will make it very clear why.

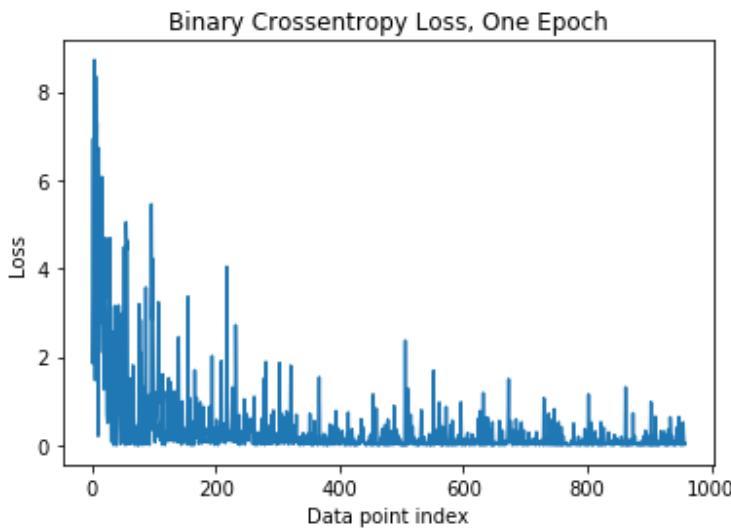
Let's train the model one point at a time first, for one epoch, i.e. passing all of the training data once:

```
losses = []
idx = range(len(X_train) - 1)
for i in idx:
    loss, _ = model.train_on_batch(X_train[i:i+1], y_train[i:i+1])
    losses.append(loss)
```

Let's plot the losses we have just calculated. As you will see the value of the loss changes greatly from one update to the next:

```
plt.plot(losses)
plt.title('Binary Crossentropy Loss, One Epoch')
plt.xlabel('Data point index')
plt.ylabel('Loss')
```

```
Text(0, 0.5, 'Loss')
```



As you can see in the plot, passing one data point at a time results in a very **noisy** estimation of the gradient. We can improve the estimation of the gradient by averaging the gradients over a few points contained in a **mini-batch**.

Common choices for the mini-batch size are 16, 32, 64, 128, 256, generally powers of 2. With mini-batch gradient descent, we do N/B weight updates per epoch, with N equals to the number of points in the training set and B equals to the number of points in a mini-batch.

Let's reset the model weights to their initial random values:

```
model.set_weights(weights)
```

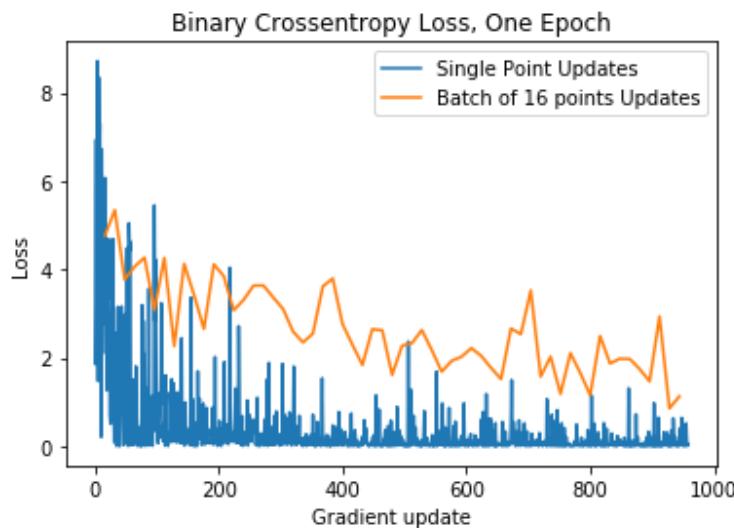
Now let's train the model with batches of 16 points each:

```
B = 16

batch_idx = np.arange(0, len(X_train) - B, B)
batch_losses = []
for i in batch_idx:
    loss, _ = model.train_on_batch(X_train[i:i+B], y_train[i:i+B])
    batch_losses.append(loss)
```

Now let's plot the losses calculated with mini-batch gradient descent over the losses calculated at each point. As you will see, the loss decreases in a much smoother fashion:

```
plt.plot(idx, losses)
plt.plot(batch_idx + B, batch_losses)
plt.title('Binary Crossentropy Loss, One Epoch')
plt.xlabel('Gradient update')
plt.ylabel('Loss')
plt.legend(['Single Point Updates', 'Batch of 16 points Updates'])
plt.show()
```



The min-batch method is what `keras` automatically does for us when we invoke the `.fit` method. When we run `model.fit` we can specify the number of `epochs` and the `batch_size`, like we have been doing many times:

```
model.set_weights(weights)

history = model.fit(X_train, y_train, batch_size=16, epochs=20)
```

```
Epoch 1/20
960/960 [=====] - 0s 132us/step - loss: 2.6466 - acc: 0.2552
Epoch 2/20
960/960 [=====] - 0s 130us/step - loss: 0.9773 - acc: 0.5510
Epoch 3/20
960/960 [=====] - 0s 133us/step - loss: 0.4814 - acc: 0.7938
Epoch 4/20
960/960 [=====] - 0s 131us/step - loss: 0.3281 - acc: 0.8833
Epoch 5/20
...
Epoch 5/20
```

Now that we've trained the model, we can evaluate its performance on the test set using the `model.evaluate` method. This is somewhat equivalent to the `model.score` method in Scikit-Learn . It returns a dictionary with the loss, and all the other metrics we passed when we executed `model.compile` .

```
result = model.evaluate(X_test, y_test)
"Test accuracy: {:.2f} %".format(result[1]*100)
```

```
412/412 [=====] - 0s 88us/step
```

'Test accuracy: 97.82 %'

With 20 epochs of training the logistic regression model does not perform as well as the Random Forest model yet. Let's see how we can improve it. One direction that we can explore to improve a model is to tune the hyperparameters. We will start from the most obvious one, which is the **Learning Rate**.

Learning Rates

Let's explore what happens to the performance of our model if we change the learning rate. We can do this with a simple loop where we perform the following steps:

1. We recompile the model with a different learning rate.
2. We reset the weights to the initial value.
3. We retrain the model and append the results to a list.

```
from keras.optimizers import SGD
```

```
dflist = []

learning_rates = [0.01, 0.05, 0.1, 0.5]

for lr in learning_rates:

    model.compile(loss='binary_crossentropy',
                  optimizer=SGD(lr=lr),
                  metrics=['accuracy'])

    model.set_weights(weights)
    h = model.fit(X_train, y_train, batch_size=16, epochs=10, verbose=0)

    dflist.append(pd.DataFrame(h.history, index=h.epoch))
    print("Done: {}".format(lr))
```

```
Done: 0.01
Done: 0.05
Done: 0.1
Done: 0.5
```

We can concatenate all our results in a single file for easy visualization using the `pd.concat` function along the columns axis.

```
historydf = pd.concat(dflist, axis=1)
```

```
historydf
```

	acc	loss	acc	loss	acc	loss	acc	lc
0	0.255208	2.631684	0.656250	0.966480	0.817708	0.608429	0.904167	0.3418
1	0.548958	0.980156	0.941667	0.176473	0.971875	0.114553	0.981250	0.0542

	acc	loss	acc	loss	acc	loss	acc	lc
2	0.792708	0.484010	0.969792	0.124777	0.975000	0.087007	0.982292	0.0453
3	0.885417	0.329009	0.977083	0.102571	0.979167	0.073091	0.983333	0.0397
4	0.907292	0.257602	0.977083	0.090085	0.983333	0.064990	0.988542	0.0350
5	0.930208	0.217109	0.975000	0.081961	0.983333	0.059179	0.986458	0.0361
6	0.936458	0.190043	0.977083	0.074622	0.983333	0.054149	0.982292	0.0367
7	0.944792	0.171013	0.982292	0.070410	0.985417	0.050862	0.987500	0.0322
8	0.950000	0.156974	0.983333	0.065660	0.985417	0.047986	0.984375	0.0319
9	0.956250	0.145814	0.982292	0.062593	0.986458	0.045536	0.987500	0.0319

And we can add information about the learning rate in a secondary column index using the `pd.MultiIndex` class.

```
metrics_reported = dflist[0].columns
idx = pd.MultiIndex.from_product([learning_rates, metrics_reported],
                                 names=['learning_rate', 'metric'])

historydf.columns = idx
```

```
historydf
```

learning_rate	0.01		0.05		0.10		0.50
metric	acc	loss	acc	loss	acc	loss	acc
0	0.255208	2.631684	0.656250	0.966480	0.817708	0.608429	0.90410
1	0.548958	0.980156	0.941667	0.176473	0.971875	0.114553	0.9812!

learning_rate	0.01		0.05		0.10		0.50
metric	acc	loss	acc	loss	acc	loss	acc
2	0.792708	0.484010	0.969792	0.124777	0.975000	0.087007	0.982291
3	0.885417	0.329009	0.977083	0.102571	0.979167	0.073091	0.983310
4	0.907292	0.257602	0.977083	0.090085	0.983333	0.064990	0.988540
5	0.930208	0.217109	0.975000	0.081961	0.983333	0.059179	0.986410
6	0.936458	0.190043	0.977083	0.074622	0.983333	0.054149	0.982291
7	0.944792	0.171013	0.982292	0.070410	0.985417	0.050862	0.987500
8	0.950000	0.156974	0.983333	0.065660	0.985417	0.047986	0.984310
9	0.956250	0.145814	0.982292	0.062593	0.986458	0.045536	0.987500

Now we can display the behavior of loss and accuracy as a function of the learning rate.

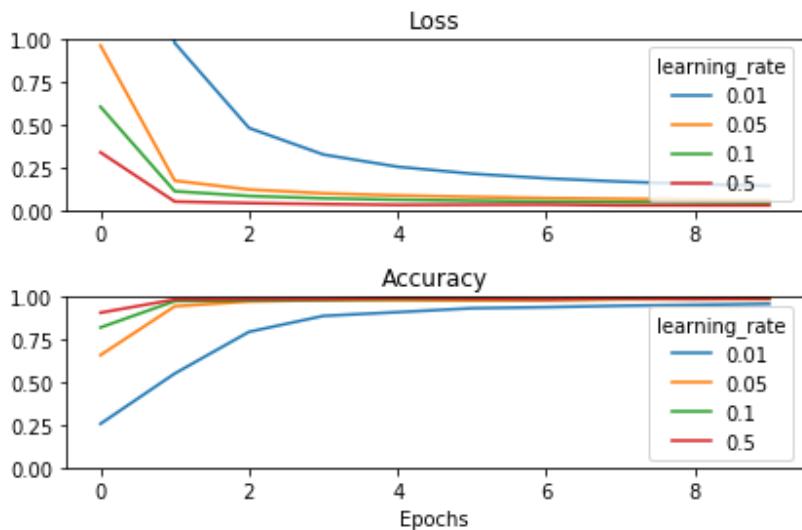
```

ax = plt.subplot(211)
historydf.xs('loss', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
plt.title("Loss")

ax = plt.subplot(212)
historydf.xs('acc', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
plt.title("Accuracy")
plt.xlabel("Epochs")

plt.tight_layout()

```



As expected a small learning rate gives a much slower decrease in the loss. Another hyperparameter we can try to tune is the **Batch Size**. Let's see how changing batch size affects the convergence of the model.

Batch Sizes

Let's loop over increasing batch sizes from a single point up to 128 .

```
dflist = []

batch_sizes = [1, 4, 16, 32, 64, 128]

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

for batch_size in batch_sizes:
    model.set_weights(weights)

    h = model.fit(X_train, y_train,
                  batch_size=batch_size,
                  verbose=0, epochs=20)

    dflist.append(pd.DataFrame(h.history, index=h.epoch))
    print("Done: {}".format(batch_size))
```

```
Done: 1
Done: 4
```

```
Done: 16
Done: 32
Done: 64
Done: 128
```

Like we did above we can arrange the results in a `Pandas Dataframe` for easy display. Notice how we are using the `pd.MultiIndex.from_product` function to create a multi-index for the columns so that the data is organized by batch size and by metric.

```
historydf = pd.concat(dflist, axis=1)
metrics_reported = dflist[0].columns
idx = pd.MultiIndex.from_product([batch_sizes, metrics_reported],
                                 names=['batch_size', 'metric'])
historydf.columns = idx
```

```
historydf
```

batch_size	1		4		16		32
metric	acc	loss	acc	loss	acc	loss	acc
0	0.868750	0.406626	0.619792	1.112657	0.259375	2.647322	0.181250
1	0.972917	0.090896	0.929167	0.209316	0.558333	0.974283	0.325000
2	0.979167	0.069797	0.960417	0.142752	0.800000	0.481099	0.470833
3	0.982292	0.058750	0.972917	0.116789	0.877083	0.328252	0.639583
4	0.986458	0.052041	0.977083	0.101782	0.911458	0.257927	0.767708
5	0.985417	0.047568	0.976042	0.092023	0.927083	0.216849	0.832292
6	0.987500	0.043949	0.977083	0.084184	0.939583	0.190174	0.876042
7	0.987500	0.041580	0.977083	0.078409	0.942708	0.171082	0.893750
8	0.987500	0.039204	0.979167	0.073869	0.950000	0.157188	0.907292

batch_size	1		4		16		32
metric	acc	loss	acc	loss	acc	loss	acc
9	0.988542	0.038099	0.980208	0.069746	0.957292	0.145811	0.917708
10	0.988542	0.036487	0.983333	0.066377	0.960417	0.136964	0.926042
11	0.987500	0.035369	0.983333	0.063510	0.967708	0.129707	0.932292
12	0.988542	0.034837	0.983333	0.061041	0.967708	0.123597	0.935417
13	0.989583	0.034106	0.985417	0.058504	0.973958	0.118205	0.939583
14	0.989583	0.032867	0.985417	0.056769	0.973958	0.113811	0.940625
15	0.988542	0.032381	0.984375	0.054960	0.975000	0.109682	0.946875
16	0.987500	0.031833	0.986458	0.053088	0.978125	0.106059	0.950000
17	0.988542	0.031050	0.985417	0.051858	0.980208	0.102737	0.953125
18	0.987500	0.030541	0.986458	0.050489	0.980208	0.099895	0.955208
19	0.990625	0.030472	0.985417	0.049346	0.979167	0.097130	0.957292

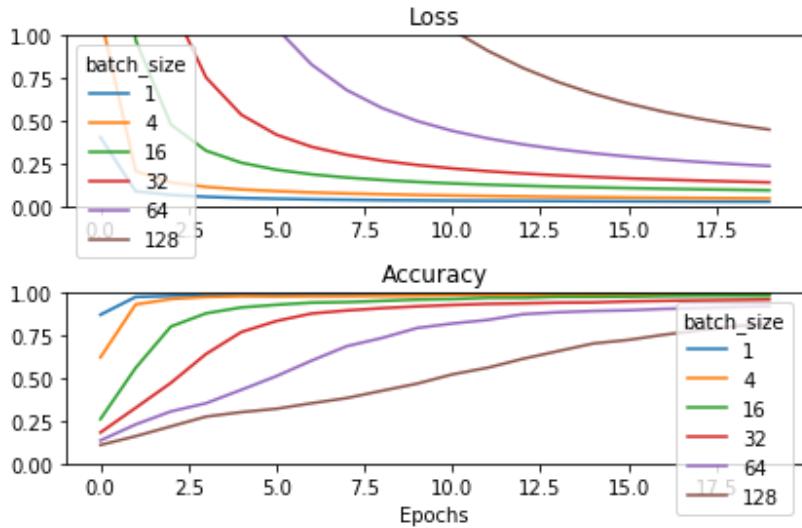
```

ax = plt.subplot(211)
historydf.xs('loss', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
plt.title("Loss")

ax = plt.subplot(212)
historydf.xs('acc', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
plt.title("Accuracy")
plt.xlabel("Epochs")

plt.tight_layout()

```



Smaller batches allow for more updates in a single epoch, on the other hand, they take much longer to run a single epoch, so there's a trade-off between speed of training (measured as number of gradients updates) and speed of convergence (measured as number of epochs). In practice a batch size of 16 or 32 data points is often used.

A [very recent research article](#) suggests to start with a small batch size and the increase it gradually. We encourage you to try and experiment with that strategy as well.

OPTIMIZERS

The **optimizer** is the algorithm used internally by Keras to update the weights and move the model towards lower values of the cost function. Keras implements several optimizers that go by fancy names like: SGD, Adam, RMSProp and many more. Despite these smart sounding names, the optimizers are all variations of the same concept, which is the **Stochastic Gradient Descent** or SGD.

SGD is so fundamental that we have invented an acronym to help you remember it. If you find it hard to remember Stochastic Gradient Descent, just think **Simply Go Down**, which is exactly what SGD does!

TIP: In the next pages you will find some mathematical symbols when the algorithms are explained. We highlighted the algorithms pseudo-code parts with a blue box like this:

Here's the algorithm

Feel free to skim through these if maths is not your favourite thing, you'll find a practical comparison of optimizers just after this section.

Stochastic Gradient Descent (or Simply Go Down) and its variations

Let's begin our discovery of optimizers with a review of the SGD algorithm. SGD only needs one hyper-parameter: the learning rate. Once we know the learning rate, we proceed in a loop by:

1. Sampling a minibatch from the training set.
2. Computing the gradients.
3. Updating the weights by subtracting the gradient times the learning rate.

Using a bit more formal language, we can write SGD as:

SGD

- Choose an initial vector of parameters w and learning rate η - Repeat until stop rule:
 - Extract a random batch from the training set, with corresponding training labels
 - Evaluate the average cost function $J(y, \hat{y})$ using the points in the batch
 - Evaluate the gradient $g = \nabla_w J(w)$ using the points in the batch and the current value of the parameters w
 - Apply the update rule: $w \leftarrow w - \eta g$

The stop rule could be a fixed number of updates or epochs as well a condition on the amount of change in the cost function. For example, we could decide to stop the training loop if the value of the cost is not changing too much.

Momentum

In recent years, several improvements have been proposed to this formula. In particular, we would like to have a

A first improvement of the SGD is to add momentum. **Momentum** means that we accumulate the gradient corrections in a variable v called *velocity*, that basically serves as a smoothed version of the gradient.

- Like SGD, choose an initial vector of parameters w , a learning rate η and a momentum parameter μ
- Repeat until stop rule:
 - Same 3 steps as SGD (get batch, evaluate cost, evaluate gradient)
 - Accumulate gradients into velocity: $v = \mu v - \eta g$
 - Apply the update rule: $w \rightarrow w - v$

Applying momentum is like saying: if you are going down in a direction, then you should keep going more or less in that direction minus a small correction given by the new gradients. It's as if instead of walking downhill, we would roll down like a ball. The name comes from physics, in case you're curious.

AdaGrad

SGD and SGD + momentum keep the learning rate constant for each parameter. This can be problematic if the parameters are sparse (i.e. most of them are zero except a few ones).

Adaptive algorithm, like **AdaGrad** overcome this problem by accumulating the square of the gradient into a normalization variable for each of the parameter. The result of this is that each parameter will have a personalized learning rate. Parameters whose gradient is large, will have a learning rate that decreases fast, while parameters that have small gradients will have a large learning rate.

This has proven to converge faster than pure SGD.

- Like SGD, choose an initial vector of parameters w , a learning rate η , a small constant $\delta = 10^{-7}$ to avoid division by 0
- Repeat until stop rule:
 - Same 3 steps as SGD (get batch, evaluate cost, evaluate gradient)
 - Accumulate the square of the gradient: $r \rightarrow r + g \cdot g$
 - Compute update: $\Delta w = \eta \frac{1}{\sqrt{r}} g$
 - Apply the update rule $w \rightarrow w - \Delta w$

Let's break down the above equation for the update so that we understand it fully. Both the accumulation step and the update step are computed element by element, so we can focus on a single parameter.

- For a single parameter w_i , $\|g\|^2$ is equivalent to r_i^2 , so we are accumulating the square of the gradient in a variable r_i for each parameter.
- $\frac{\eta}{\delta + \sqrt{r}} \odot g$ may look a bit daunting at first, so let's break it down. η is the learning rate, no surprises here. For a single parameter w_i we are dividing the value of the gradient g_i by the square root of the accumulated square gradients r_i^2 . If the gradients are large we will be dividing by a large quantity. On the other hand, if the gradients are small, we will be dividing by a small quantity. This yields a practically constant update step size, multiplied by the learning rate. The δ in the denominator is a numerical regularization constant, so that we do not risk dividing by zero if r becomes too small.

RMSProp: Root Mean Square Propagation (or Adagrad with EWMA)

RMSProp is also adaptive, but it allows to choose the fraction of squared gradients to accumulate, using an [Exponentially Weighted Moving Average \(or EWMA\)](#) decay in the accumulation formula. If you're not familiar with how EWMA works, we strongly encourage you to review the [Appendix](#). EWMA is *the most important algorithm of your life!*

- Like SGD, choose an initial vector of parameters w , a learning rate η , a small constant $\delta = 10^{-7}$ to avoid division by zero and an EWMA mixing factor ρ between 0 and 1, this is also called decay rate - Repeat until stop rule: - Same 3 steps as SGD (get batch, evaluate cost, evaluate gradient) - Accumulate EWMA of the square of the gradient: $r \rightarrow \rho r + (1-\rho) \|g\|^2$ - Same update rules as Adagrad

Adam: Adaptive Moment Estimation (or EWMA everywhere)

Finally, let's introduce Adam. This algorithm improves upon RMSProp by applying EWMA to the gradient update as well as the square of the gradient.

- Like SGD, choose an initial vector of parameters w , a learning rate η , a small constant $\delta = 10^{-7}$ to avoid division by zero and an EWMA mixing factors ρ_1 and ρ_2 between 0 and 1 (usually chosen as 0.9 and 0.999 respectively) - Repeat until stop rule: - Same 3 steps as SGD (get batch, evaluate cost, evaluate gradient) - Accumulate EWMA of the gradient: $v \rightarrow \rho_1 v + (1-\rho_1) g$ - Accumulate EWMA the square of the gradient: $r \rightarrow \rho_2 r + (1-\rho_2) \|g\|^2$ - Correct bias 1: $\hat{v} = \frac{v}{1 - \rho_1^t}$ - Correct bias 2: $\hat{r} = \frac{r}{1 - \rho_2^t}$ - Compute update: $\Delta w = \eta \frac{\hat{v}}{\hat{r}} + \sqrt{\hat{r}}$ - Apply the update rule $w \rightarrow w - \Delta w$

This formula may also appear to be a bit complicated, so let's walk through it step by step.

- EWMA is applied to both the gradient and its square. We are taking inspiration from both the momentum and the RMSProp formulas.
- The only other novelty is the bias correction. We take the current value of the accumulated quantity and divide it by $(1 - \rho^t)$. Since both decay rates are almost 1, the normalization is very small initially, and it increases as time goes by. This seems to work in practice really well.

In summary, we have seen few of the most popular optimization algorithms. You are probably wondering how to choose the best one. Unfortunately there is no best one, and each of them performs better in some conditions. What is true though, is that a good choice of the hyper parameters is key for an algorithm to perform well, and we encourage you to familiarize yourself with one algorithm and understand the effects of changing hyper parameter.

Let's compare the performance of few optimizers in `keras`. Optimizers are available in the `keras.optimizer` module, so let's start by importing them:

```
from keras.optimizers import SGD, Adam, Adagrad, RMSprop
```

We then set the learning rate to be the same for each of them and run the training for 5 epochs each:

```
dflist = []

optimizers = ['SGD(lr=0.01)',
              'SGD(lr=0.01, momentum=0.3)',
              'SGD(lr=0.01, momentum=0.3, nesterov=True)',
              'Adam(lr=0.01)',
              'Adagrad(lr=0.01)',
              'RMSprop(lr=0.01)']

for opt_name in optimizers:
    model.compile(loss='binary_crossentropy',
                  optimizer=eval(opt_name),
                  metrics=['accuracy'])

    model.set_weights(weights)

    h = model.fit(X_train, y_train, batch_size=16, epochs=5, verbose=0)

    dflist.append(pd.DataFrame(h.history, index=h.epoch))
    print("Done: ", opt_name)
```

```
Done: SGD(lr=0.01)
Done: SGD(lr=0.01, momentum=0.3)
Done: SGD(lr=0.01, momentum=0.3, nesterov=True)
Done: Adam(lr=0.01)
Done: Adagrad(lr=0.01)
Done: RMSprop(lr=0.01)
```

We can aggregate the results like we did previously:

```
historydf = pd.concat(dflist, axis=1)
metrics_reported = dflist[0].columns
idx = pd.MultiIndex.from_product([optimizers, metrics_reported],
                                 names=['optimizers', 'metric'])
historydf.columns = idx
```

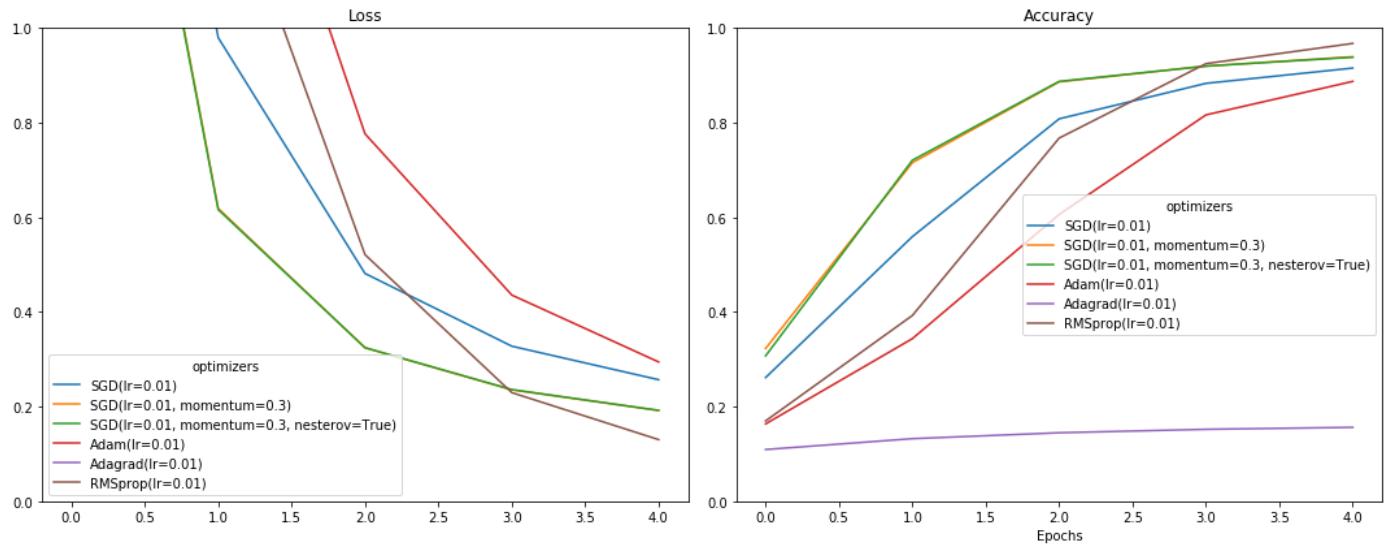
And plot them for comparison:

```
plt.figure(figsize=(15, 6))

ax = plt.subplot(121)
historydf.xs('loss', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
plt.title("Loss")

ax = plt.subplot(122)
historydf.xs('acc', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
plt.title("Accuracy")
plt.xlabel("Epochs")

plt.tight_layout()
```



As you can see, in this particular case, some optimizers converge a lot faster than others. This could be due to the particular combination of hyper-parameters chosen as well as to the their better performance on this particular problem. We encourage you to try out different optimizers on your problems, as well as trying different hyper-parameter combinations.

INITIALIZATION

So far we have explored the effect of learning rate, batch size and optimizers on the speed of convergence of a model. We have compared their effect starting from the same set of randomly initialized weights. What if we initialized weights in a different weight and kept everything else fixed? This may seem unimportant but it turns out that the initialization is actually critical. A model could not converge at all for some initialization and converge really quickly for some other initialization. While we don't really understand this fully, we have a few heuristic strategies available, that we can test, looking for the best one for our specific problem.

`keras` offers the possibility to initialize the weights in several ways including:

- Zeros, ones, constant: all weights are initialized to zero, to one or to a constant value. Generally these are not good choices, because they leave the model uncertain on which parameters to optimize first.

Initialization strategies try to "break the symmetry" by assigning random values to the parameters. The range and type of the random distribution can vary and several initialization schemes have been proposed:

- *Random uniform*: each weight receives a random value between 0 and 1, chosen with uniform probability.
- *Lecun_uniform*: like the above, but the values are drawn in the interval $[-\text{limit}, \text{limit}]$ where $\text{limit} = \frac{\sqrt{3}}{\text{# inputs}}$. Where # inputs indicates the number of inputs in the weight tensor for a specific layer.
- *Normal*: each weight receives a random value drawn from a normal distribution with mean 0 and standard deviation of 1.
- *He_normal*: like the previous one, but with standard deviation $\sigma = \sqrt{\frac{2}{\text{# in}}}$.
- *Glorot_normal*: like the previous one, but with standard deviation $\sigma = \sqrt{\frac{2}{\text{# in} + \text{# out}}}$.

You can read more about them [here](#). In order to see the effect of initialization we'll use a deeper network with more than just 5 weights.

```
import keras.backend as K
```

```
dflist = []

initializers = ['zeros', 'ones', 'uniform', 'lecun_uniform',
                'normal', 'he_normal', 'glorot_normal']
```

```

for init in initializers:

    K.clear_session()

    model = Sequential()
    model.add(Dense(10, input_shape=(4,),
                  kernel_initializer=init,
                  activation='tanh'))
    model.add(Dense(10, kernel_initializer=init,
                  activation='tanh'))
    model.add(Dense(10, kernel_initializer=init,
                  activation='tanh'))
    model.add(Dense(1, kernel_initializer=init,
                  activation='sigmoid'))

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

    h = model.fit(X_train, y_train, batch_size=16, epochs=10, verbose=0)

    dflist.append(pd.DataFrame(h.history, index=h.epoch))
    print("Done: ", init)

```

```

Done: zeros
Done: ones
Done: uniform
Done: lecun_uniform
Done: normal
Done: he_normal
Done: glorot_normal

```

Let's aggregate and plot the results

```

historydf = pd.concat(dflist, axis=1)
metrics_reported = dflist[0].columns
idx = pd.MultiIndex.from_product([initializers, metrics_reported],
                                 names=['initializers', 'metric'])

historydf.columns = idx

```

```
styles = ['-+', '-*', '-x', '-d', '-^', '-o', '-s']
```

```

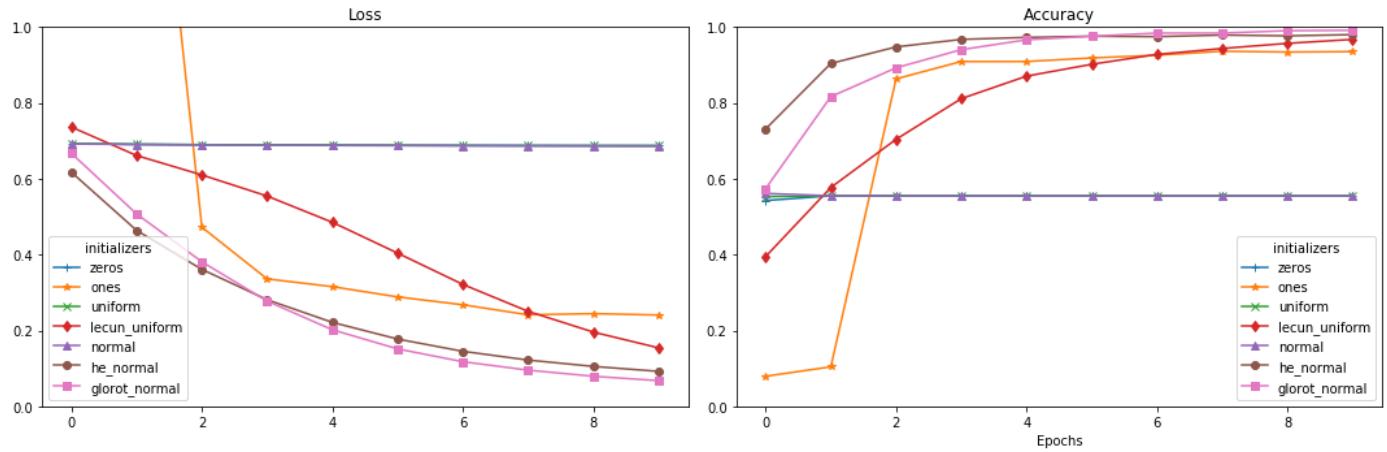
plt.figure(figsize=(15, 5))

ax = plt.subplot(121)
xs = historydf.xs('loss', axis=1, level='metric')
xs.plot(ylim=(0,1), ax=ax, style=styles)
plt.title("Loss")

ax = plt.subplot(122)
xs = historydf.xs('acc', axis=1, level='metric')
xs.plot(ylim=(0,1), ax=ax, style=styles)
plt.title("Accuracy")
plt.xlabel("Epochs")

plt.tight_layout()

```



As you can see some initializations don't even converge, while some do converge rather quickly. Initialization of the weights plays a very important role in large models, so it is important to try a couple of different initialization schemes in order to get the best results.

INNER LAYER REPRESENTATION

We conclude this dense chapter on how to train a Neural Network with a little treat. As mentioned previously, a Neural Network can be viewed as a general function between any input and any output. This is also true for any of the intermediate layers. Each layer learns a nonlinear transformation between its inputs and its outputs, so we can pull out the values at the output of any layer. This gives us a way to see how our network is learning to separate our data. Let's see how it's done. First of all we will re-train a network with 2 layers, the first with 2 nodes and the second with just 1 output node.

Let's clear the backend session first:

```
K.clear_session()
```

Then we define and compile the model:

```
model = Sequential()
model.add(Dense(2, input_shape=(4,), activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(lr=0.01),
              metrics=['accuracy'])
```

We then set the model weights to some random values. In order to get reproducible results, the random values are given for this particular run:

```
weights = [np.array([[ -0.26285839,   0.82659411],
                     [ 0.65099144,  -0.7858932 ],
                     [ 0.40144777,  -0.92449236],
                     [ 0.87284446,  -0.59128475]]),
           np.array([[ 0.,   0.]]),
           np.array([[-0.7150408 ], [ 0.54277754]]),
           np.array([[ 0.]]]

model.set_weights(weights)
```

And then we train the model

```
h = model.fit(X_train, y_train, batch_size=16, epochs=20,  
              verbose=1, validation_split=0.3)
```

```
Train on 672 samples, validate on 288 samples  
Epoch 1/20  
672/672 [=====] - 0s 410us/step - loss: 0.4868 - acc: 0.7649 - v  
Epoch 2/20  
672/672 [=====] - 0s 175us/step - loss: 0.3434 - acc: 0.8646 - v  
Epoch 3/20  
672/672 [=====] - 0s 176us/step - loss: 0.2326 - acc: 0.9405 - v  
Epoch 4/20  
672/672 [=====] - 0s 174us/step - loss: 0.1475 - acc: 0.9792 - v  
...  
672/672 [=====] - 0s 174us/step - loss: 0.1475 - acc: 0.9792 - v
```

Let's look at the layers using the `model.summary` function:

```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 2)	10
dense_2 (Dense)	(None, 1)	3

Total params: 13
Trainable params: 13
...
Trainable params: 13

The model only has 2 dense layers. One connecting the input to the 2 inner nodes and one connecting these 2 inner nodes to the output. The list of layers is accessible as an attribute of the model:

```
model.layers
```

```
[<keras.layers.core.Dense at 0x7face0c2acf8>,  
<keras.layers.core.Dense at 0x7face0c2a668>]
```

```
...
<keras.layers.core.Dense at 0x7face0c2a668>
...
<keras.layers.core.Dense at 0x7face0c2a668>
```

and the inputs and outputs of each layer are also accessible as attributes. Let's take the input of the first layer and the output of the first layer, the one with 2 nodes:

```
inp = model.layers[0].input
out = model.layers[0].output
```

These variables refer to objects from the keras kernel. This is Tensorflow by default but it can be switched to other kernels if needed.

```
inp
```

```
<tf.Tensor 'dense_1_input:0' shape=(?, 4) dtype=float32>
```

```
out
```

```
<tf.Tensor 'dense_1/Relu:0' shape=(?, 2) dtype=float32>
```

Both the input and the output are Tensorflow tensors. In the next chapter we will learn more about Tensors, so don't worry about them for now.

keras allows us to define a function between any tensors in a model as follows:

```
features_function = K.function([inp], [out])
```

Notice that `features_function` is a function itself, so `K.function` is a function that returns a function.

```
features_function
```

```
<keras.backend.tensorflow_backend.Function at 0x7facd9536400>
```

We can apply this function to the test data. Notice that the function expects a list of inputs and returns a list of outputs. Since our inputs list only has one element, so will the output list and we can extract the outputs by taking the first element:

```
features = features_function([X_test])[0]
```

The output tensor contains as many points as `X_test` each represented by 2 numbers, the output values of the 2 nodes in the first layer:

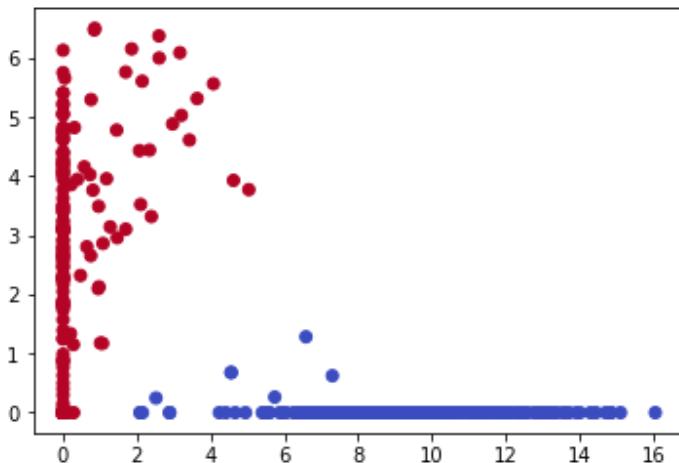
```
features.shape
```

```
(412, 2)
```

We can plot the data as a scatter plot, and we can see how the network has learned to represent the data in 2 dimensions in such a way that the next layer can separate the 2 classes more easily:

```
plt.scatter(features[:, 0], features[:, 1], c=y_test, cmap='coolwarm')
```

```
<matplotlib.collections.PathCollection at 0x7facd94fa9b0>
```



Let's plot the output of the second-to-last layer at each epoch in a training loop. First we re-initialize the model:

```
model.set_weights(weights)
```

Then we create a `K.function` between the input and the output of layer 0:

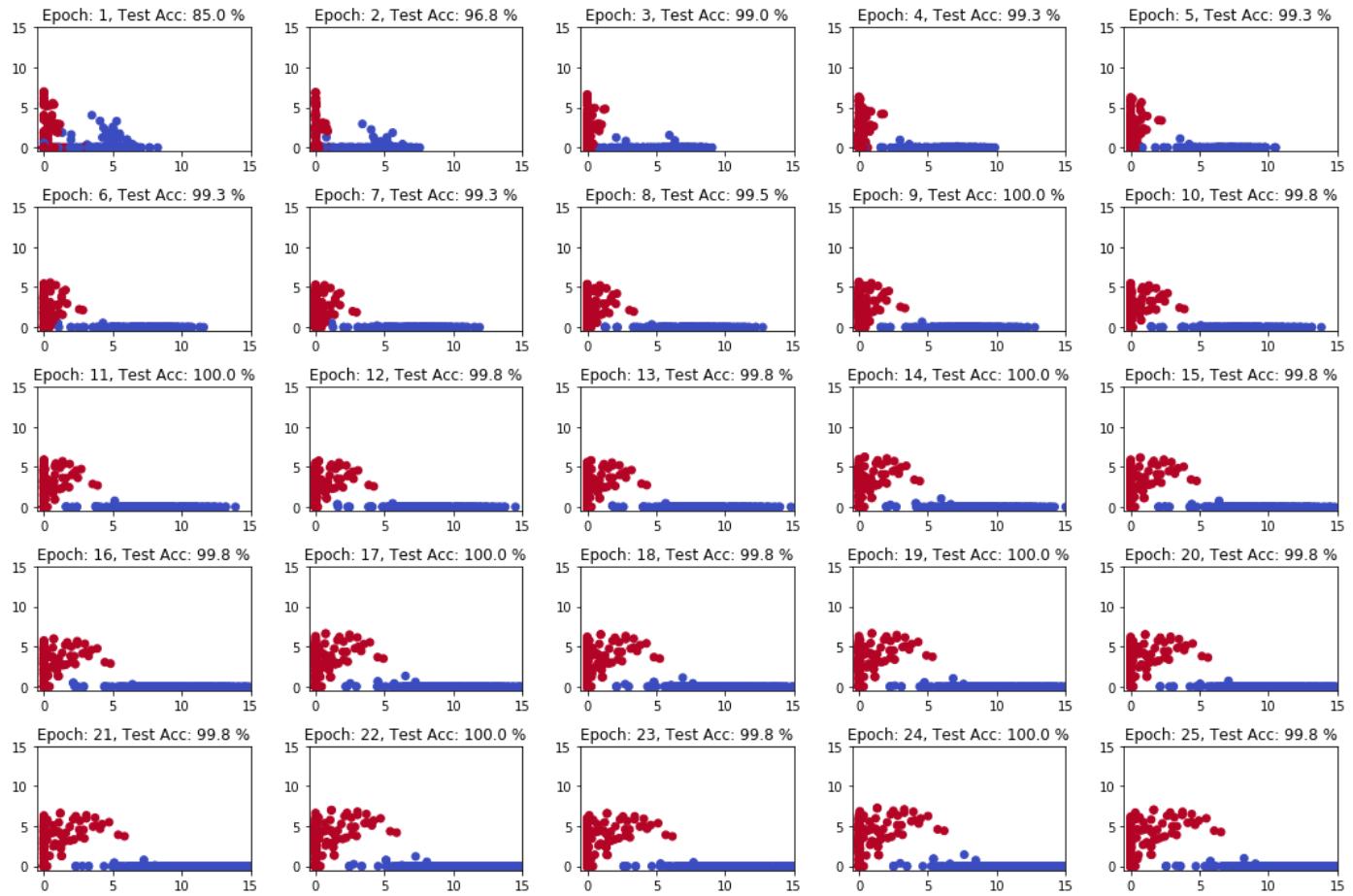
```
inp = model.layers[0].input
out = model.layers[0].output
features_function = K.function([inp], [out])
```

Then we train the model one epoch at a time, plotting the 2D representation of the data as it comes out from layer 0:

```
plt.figure(figsize=(15,10))

for i in range(1, 26):
    plt.subplot(5, 5, i)
    h = model.fit(X_train, y_train, batch_size=16, epochs=1, verbose=0)
    test_acc = model.evaluate(X_test, y_test, verbose=0)[1]
    features = features_function([X_test])[0]
    plt.scatter(features[:, 0], features[:, 1], c=y_test, cmap='coolwarm')
    plt.xlim(-0.5, 15)
    plt.ylim(-0.5, 15)
    plt.title('Epoch: {}, Test Acc: {:.1f} %'.format(i, test_acc * 100.0))

plt.tight_layout()
```



As you can see, at the beginning the network has no notion of the difference between the two classes. As the training progresses, the network learns to represent the data in a 2 dimensional space where the 2 classes are linearly separable, so that the final layer (which is basically a logistic regression) can easily separate them with a straight line.

This chapter was surely more intense and theoretical than the previous ones, but we hope it gave you a thorough understanding of the inner workings of how a Neural Network works and what you can do to improve its performance.

EXERCISES

Exercise 1

You've just been hired at a wine company and they would like you to help them build a model that predicts the quality of their wine based on several measurements. They give you a dataset with wine:

- load the `../data/wines.csv` into Pandas
- use the column called "Class" as target
- check how many classes are there in target, and if necessary use dummy columns for a Multiclass classification
- use all the other columns as features, check their range and distribution (using seaborn pairplot)
- rescale all the features using either MinMaxScaler or StandardScaler
- build a deep model with at least 1 hidden layer to classify the data
- choose the cost function, what will you use? Mean Squared Error? Binary Cross-Entropy? Categorical Cross-Entropy?
- choose an optimizer
- choose a value for the learning rate, you may want to try with several values
- choose a batch size
- train your model on all the data using a `validation_split=0.2`. Can you converge to 100% validation accuracy?
- what's the minimum number of epochs to converge?
- repeat the training several times to verify how stable your results are

Exercise 2

Since this dataset has 13 features we can only visualize pairs of features like we did in the Paired plot. We could however exploit the fact that a Neural Network is a function to extract 2 high level features to represent our data.

- build a deep fully connected network with the following structure:
 - Layer 1: 8 nodes
 - Layer 2: 5 nodes
 - Layer 3: 2 nodes

- Output: 3 nodes
- choose activation functions, initializations, optimizer and learning rate so that it converges to 100% accuracy within 20 epochs (not easy)
- remember to train the model on the scaled data
- define a Feature Function like we did above between the input of the 1st layer and the output of the 3rd layer
- calculate the features and plot them on a 2-dimensional scatter plot
- can we distinguish the 3 classes well?

Exercise 3

Keras functional API. So far we've always used the Sequential model API in Keras. However, Keras also offers a Functional API, which is much more powerful. You can find its documentation [here](#). Let's see how we can leverage it.

- define an input layer called `inputs`
- define two hidden layers as before, one with 8 nodes, one with 5 nodes
- define a `second_to_last` layer with 2 nodes
- define an output layer with 3 nodes
- create a model that connect input and output
- train it and make sure that it converges
- define a function between inputs and `second_to_last` layer
- recalculate the features and plot them

Exercise 4

Keras offers the possibility to call a function at each epoch. These are Callbacks, and their documentation is [here](#). Callbacks allow us to add some neat functionality. In this exercise we'll explore a few of them.

- Split the data into train and test sets with a `test_size = 0.3` and `random_state=42`
- Reset and recompile your model
- train the model on the train data using `validation_data=(X_test, y_test)`
- Use the `EarlyStopping` callback to stop your training if the `val_loss` doesn't improve
- Use the `ModelCheckpoint` callback to save the trained model to disk once training is finished
- Use the `TensorBoard` callback to output your training information to a `/tmp/` subdirectory

Chapter 6: Convolutional Neural Networks

INTRO

In the previous chapter we dove into Deep Learning, we built our first real model, and hopefully demystified a lot of the complicated stuff. Now it's time to start applying Deep Learning to a kind of data where it really shines: images!

At the root of it, what is an image anyway? The information in an image is encoded by the relations between nearby pixels. A slightly darker or lighter image still contains the same information. Similarly, it doesn't matter where an object is positioned exactly, in order to recognize it. **Convolutional Neural Networks** (CNN), as we will discover in this chapter, are able to encode a lot of information about relations between nearby pixels. This makes them great tools to work with images, as well as with sequences like sounds and movies.

In this section we will learn what convolutions are, how they can be used to filter images, and how Convolutional Neural Nets work. By the end of this section, we will train our first CNN to recognize handwritten digits. We will also introduce the core concept of **Tensor**. Are you ready? Let's go!

MACHINE LEARNING ON IMAGES WITH PIXELS

Image classification or image recognition is the process of identifying the object contained in an image. The classifier receives an image as input and it returns a label indicating the object represented in the image.

Consider this image:



humans quickly recognize the a cat, whereas the computer just sees a bunch of pixels and has no prior notion of what a cat is, nor that a cat is represented in this image. It may seem magic that neural networks are able to solve the image classification problem well, but we hope that, by the end of this chapter, how they do it will be quite clear!

In order to understand why it is so difficult for a computer to classify objects in images let's start from how images are represented, and in particular let's start with a black and white image.

A black and white image can be represented as a grid of points, each point with a binary value. These points on the grid are called **pixels**, and in a black and white image they only carry two possible values: 0 and 1.

Let's create a random Black and White image with Python. As always, we start by importing the usual libraries. By now they should be familiar, but if you need a reminder have a look at [Chapter 1](#):

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Let's use the `np.random.binomial` function to generate a 10x10 square matrix of random zeros and ones. Using the `np.random.binomial()` method will give us an approximately equal amount of zeros and ones.

TIP: according to the [documentation](#), `np.random.binomial` creates a random distribution where samples are drawn from a **binomial distribution**:

```
binomial(n, p, size=None)
Draw samples from a binomial distribution.
```

with n (≥ 0) is the number of trials and p (in the interval [0,1]) is the probability of success.

We will use the argument `size=(10, 10)` to specify that we want an array with 2 axes, each with 10 positions:

```
bw = np.random.binomial(1, 0.5, size=(10, 10))
```

Let's print out the array `bw`:

```
bw
```

```
array([[1, 0, 1, 0, 1, 1, 1, 1, 0],
       [0, 1, 0, 1, 0, 1, 0, 1, 0],
```

```

[0, 1, 0, 1, 1, 1, 1, 1, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 1, 1, 0, 0, 0, 0, 0, 1],
[0, 1, 0, 1, 0, 1, 1, 1, 0, 0],
[1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
[0, 0, 0, 1, 0, 0, 1, 1, 1, 0],
[0, 1, 1, 0, 0, 0, 1, 1, 1, 1],
...
[1, 1, 0, 1, 0, 1, 0, 1, 1, 0]])

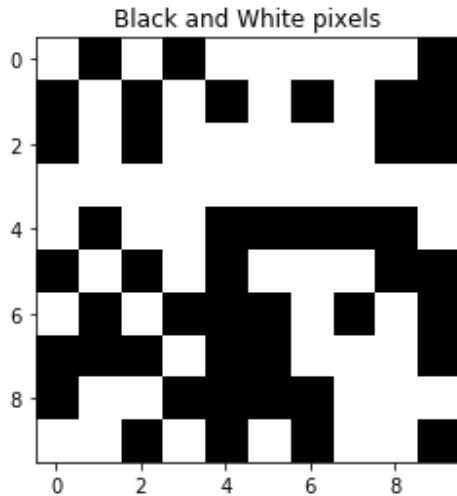
```

As promised, it's a random set of zeros and ones. We can also use the function `matplotlib.pyplot.imshow` to visualize it as an image. Let's do it:

```

plt.imshow(bw, cmap='gray')
plt.title("Black and White pixels")
plt.show()

```



Awesome! We have just learned how to create a Black and White image with Python. Let's now generate a grayscale image.

To generate a grayscale image we simply allow the pixels to carry values that are intermediate between 0 and 1. Actually, since we do not really care about infinite possible shades of gray, we normally use unsigned integers with 8 bits, i.e. the numbers from 0 to 255.

A 10x10 grayscale image with 8-bit resolution is a grid of numbers, each-of-which is an integer between 0 and 255.

Let's draw one such image. In this case we will use the `np.random.randint` function, which generates random integers uniformly distributed between a low and a high extremes. Here's a snippet from the documentation:

TIP: from the documentation of `np.random.randint` :

```
randint(low, high=None, size=None, dtype='l')
```

Return random integers from `low` (inclusive) to `high` (exclusive).
`_Low_` and `_high_` are the lowest (signed) and largest integer to be drawn from the distrib

```
gs = np.random.randint(0, 256, size=(10, 10))
```

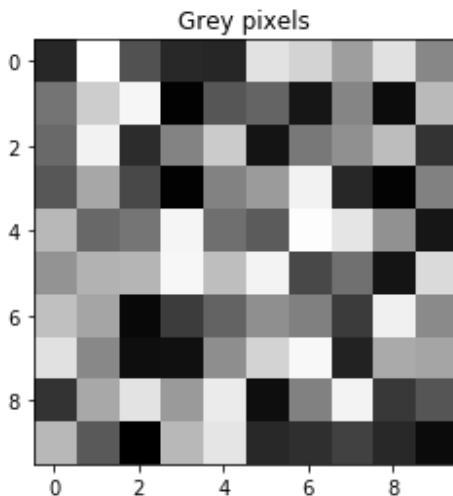
Let's print out the array `gs` :

```
gs
```

```
array([[ 45, 255, 86, 46, 44, 226, 213, 161, 225, 138],
       [120, 206, 246, 9, 91, 104, 29, 136, 19, 188],
       [110, 242, 51, 135, 204, 27, 124, 147, 191, 57],
       [ 92, 169, 77, 9, 133, 158, 242, 45, 10, 132],
       [186, 108, 121, 245, 115, 96, 253, 229, 148, 29],
       [150, 180, 183, 247, 192, 244, 77, 116, 27, 219],
       [193, 167, 15, 67, 103, 146, 131, 65, 240, 141],
       [225, 139, 21, 23, 145, 212, 248, 40, 172, 167],
       [ 57, 170, 228, 157, 235, 21, 132, 243, 62, 90],
       ...
       [186, 94, 7, 186, 229, 46, 53, 71, 47, 18]])
```

As expected it's a 10x10 grid of random integers between 0 and 255. Let's visualize it as an image:

```
plt.imshow(gs, cmap='gray')
plt.title("Grey pixels")
plt.show()
```



Wonderful! In image classification problems we have to think of images as the input into the algorithm, therefore, this 2D array with 100 numbers, corresponds to one data point in a classification task. How could we train a Machine Learning algorithm on such data? Let's say we have many such gray-scale images representing handwritten digits. How do we feed them to a Machine Learning model?

MNIST

The [MNIST database](#) is a very famous dataset of handwritten digits and it has become a benchmark for image recognition algorithms. It consists of 70000 images of 28 pixel by 28 pixels, each representing a handwritten digit.

TIP: Think of how many real world applications involve recognition of handwritten digits:

- zipcodes
- tax declarations
- student tests
- ...

The target variables are the 10 digits from 0 to 9.

Keras has its built-in dataset for MNIST, so we will load it from there using the `load_data` function

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

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Let's check the shape of the arrays of the data we received for the training and test sets:

```
X_train.shape
```

```
(60000, 28, 28)
```

```
X_test.shape
```

```
(10000, 28, 28)
```

The loaded data is a numpy array of order 3. It's like a 3-dimensional matrix, whose elements are identified by 3 indices. We'll discuss these more in detail later in this chapter.

For now, it is sufficient to know that the first index (running from 0 to 59999 for `X_train`) locates a specific image in the dataset, while the other two indices locate a certain pixel in the image, i.e. they run from 0 to the height and width of the image.

For instance, we can select the first image in the training set and take a look at its shape by using the first index:

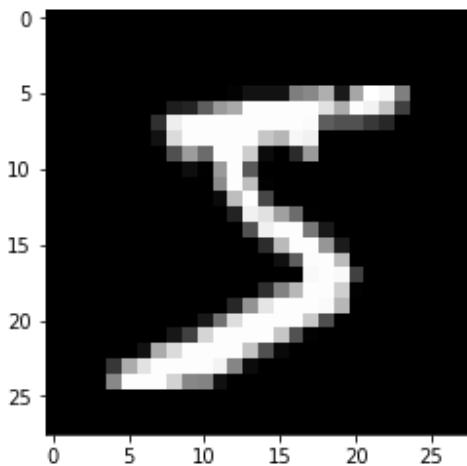
```
first_img = X_train[0]
```

This image is a 2D array of numbers between 0 and 255, like this:

Let's use `plt.imshow` once again to display the image:

```
plt.imshow(first_img, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f7556001c88>
```



Notice that with the `gray` colormap, zeros are displayed as black pixels while higher numbers are displayed as lighter pixels.

Pixels as features

How can we use this whole image as an input to a classification algorithm?

So far our input datasets have always been 2D tabular sets, where table columns refer to different features and each data point occupies a row. In this case, each data point is itself a 2D table (an image) and so we need to decide how to map it to features.

The simplest way to feed images to a machine learning algorithm is to use each pixel in the image as an individual feature. If we do this, we will have $28 \times 28 = 784$ independent features, each one being an integer between 0 and 255, and our dataset will become tabular once again. Each row in the tabular dataset will represent a different image, and each of the 784 columns will represent a specific pixel.

The `reshape` method of a numpy array allows us to reshape any array to a new shape. For example, let's reshape the training dataset to be a tabular dataset with 60000 rows and 784 columns:

```
x_train_flat = x_train.reshape(60000, 784)
```

We can check that the operation worked by printing the shape of `x_train_flat`:

```
x_train_flat.shape
```

```
(60000, 784)
```

Wonderful! Another valid syntax for `reshape` is to just specify the size of the dimensions we care about and let the method figure out the other dimension, like this:

```
x_test_flat = x_test.reshape(-1, 28*28)
```

Again, let's print the shape to be sure:

```
x_test_flat.shape
```

```
(10000, 784)
```

Great! Now we have 2 tabular datasets like the ones we are familiar with. The features contain values between 0 and 255:

```
x_train_flat.min()
```

```
0
```

```
x_train_flat.max()
```

```
255
```

As already seen in [Chapter 3](#), Neural Network models are quite sensitive to the absolute size of the input features, and hence they like features that are normalized to be somewhat near 1.

We should rescale the values of our features to be between 0 and 1. Lets do it by dividing them by 255 so they will have values between 0 and 1. Notice that we need to convert the the data type to `float32` because under the hood numpy arrays are implemented in C and therefore are strongly typed.

```
x_train_sc = x_train_flat.astype('float32') / 255.0
x_test_sc = x_test_flat.astype('float32') / 255.0
```

Great! We now have 2D data that we can use to train a fully connected Neural Network!

Multiclass output

Since our goal is to recognize a digit contained in an image, our final output is a class label between 0 and 9. Let's inspect `y_train` to look at the target values we want to train our network to learn:

```
y_train
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

We can use the `np.unique` method to check what are the unique values for the labels, these should be the digits from 0 to 9:

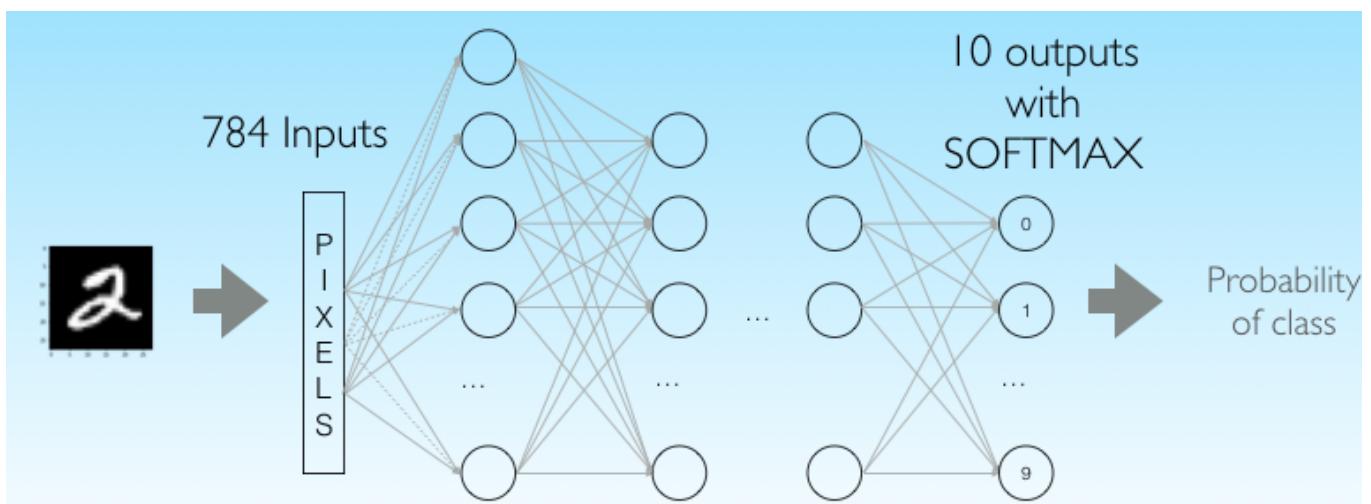
```
np.unique(y_train)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

`y_train` is an array of target digits, and it contains values between 0 and 9.

Since there are 10 possible output classes, this is a Multiclass classification problem where the outputs are mutually exclusive. As we have learned in the [Chapter 4](#), we need to convert the labels to a matrix of binary columns. In doing so, we communicate to the network that the labels are distinct and it should learn to predict the probability of an image to correspond to a specific label.

In other words, our goal is to build a network with 784 inputs and 10 output, like the one represented in this figure:



so that for a given input image the network learns to indicate to which label it corresponds. Therefore we need to make sure that the shape of the label array matches the output of our network.

We can convert our labels to binary arrays using the `to_categorical` utility function from `keras`. Let's import it

```
from keras.utils.np_utils import to_categorical
```

and let's convert both `y_train` and `y_test`:

```
y_train_cat = to_categorical(y_train)
y_test_cat = to_categorical(y_test)
```

Let's double check what's going on. As we have seen before, the first element of `x_train` is a handwritten number 5. So the corresponding label should be a 5.

```
y_train[0]
```

5

The corresponding binary version of the label is the following array:

```
y_train_cat[0]
```

```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

As you can see, this is an array of 10 numbers, zero everywhere except at position 5 (remember we start counting from 0) indicating which of the 10 classes our image should be classified as.

As seen in [Chapter 3](#) for features and in [Chapter 4](#) for labels, this type of encoding is called *one-hot* encoding, meaning we encode classes as an array with as many elements as the number of distinct classes, zero everywhere except for a 1 at the corresponding class.

What is *one-hot* mean here? It's the first time we're seeing this phrase `one-hot`. Either we should remove the phrase or (my preference) explain what it means. Either way, we can't just saying it without defining it. >>> Added internal reference to Ch. 3

Great! Finally let's check the shape of `y_train_cat`. This should have as many rows as we have training examples and 10 columns for the 10 binary outputs:

```
y_train_cat.shape
```

```
(60000, 10)
```

Let's check our test dataset to make sure it matches as well.

```
y_test_cat.shape
```

```
(10000, 10)
```

Fantastic! We can now train a fully connected Neural Network using all what we've learned in the previous chapters.

Fully connected on images

To build our network, let's import the usual Keras classes as seen in [Chapter 1](#). Once again we build a `Sequential` model, i.e. we add the layers one by one, using fully connected layers, i.e. `Dense` :

```
from keras.models import Sequential
from keras.layers import Dense
```

Now let's build the model. As we have done in [Chapter 4](#), we will build this network layer by layer, making sure that the sizes of the input/outputs.

The network configuration will be the following:

- Input: 784 features
- Layer 1: 512 nodes with Relu activation
- Layer 2: 256 nodes with Relu activation
- Layer 3: 128 nodes with Relu activation
- Layer 4: 32 nodes with Relu activation
- Output Layer: 10 nodes with Softmax activation

Notice a couple of things:

1. We specify the size of the input in the definition of the first layer through the parameter `input_dim=784` .

- The choice of the number of layers and the number of nodes per layer is arbitrary. Feel free to experiment with different architectures and observe:
 - if the network performs better or worse
 - if the training takes longer or shorter (number of epochs to reach a certain accuracy)
- The last layer added to the stack is also the output layer. This may be sometimes confusing, so make sure that the number of nodes in the last layer in the stack corresponds to the number of categories in your dataset
- The last layer outputs has a `softmax` activation function. As seen in [Chapter 4](#) this is needed when the classes are mutually exclusive. In this case, an image of a digit cannot be of 2 different digits at the same time, and we need to let the model know about it.
- Finally, the model is compiled using the `categorical_crossentropy` loss, which is the correct one for classifications with many mutually exclusive classes.

```
model = Sequential()

model.add(Dense(512, input_dim=784, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax')) # output layer

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

Let's print out the model summary:

```
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_1 (Dense)	(None, 512)	401920
dense_2 (Dense)	(None, 256)	131328
dense_3 (Dense)	(None, 128)	32896

...

As you can see, the model has about half a million parameters, namely 570,602.

Let's train it on our data for 10 epochs with 128 images per batch. We will need to pass the *scaled and reshaped* inputs and outputs.

Also, let's use a `validation_split` of 10%, meaning we will train the model on 90% of the training data, and evaluate its performance on the remaining 10%. This is like an internal train/test split done on the training data. It's useful when we plan to change the network and tune its architecture to maximize its ability to generalize. We will keep the actual test set for a final check once we have committed to the best architecture.

```
h = model.fit(X_train_sc, y_train_cat, batch_size=128, epochs=10, verbose=1, validation_s
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [=====] - 3s 53us/step - loss: 0.2932 - acc: 0.9088
Epoch 2/10
54000/54000 [=====] - 2s 28us/step - loss: 0.1011 - acc: 0.9692
Epoch 3/10
54000/54000 [=====] - 2s 28us/step - loss: 0.0674 - acc: 0.9794
Epoch 4/10
54000/54000 [=====] - 2s 28us/step - loss: 0.0498 - acc: 0.9849
...
54000/54000 [=====] - 2s 28us/step - loss: 0.0498 - acc: 0.9849
```

The model seems to be doing very well on the training data (as we can see by the `acc` output).

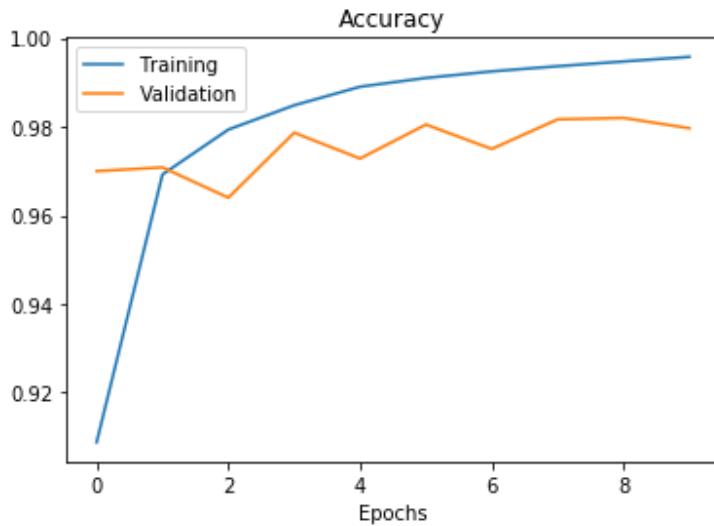
Let's check if it is overfitting, i.e. if it is just memorizing the answers instead of learning general rules about the training examples

TIP: if you need to refresh your knowledge of overfitting have a look at [Chapter 3](#) as well as [this Wikipedia article](#).

Let's plot the history of the accuracy and compare the training accuracy with the validation accuracy.

```
plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
```

```
plt.title('Accuracy')
plt.xlabel('Epochs')
plt.show()
```



We already notice that while the training accuracy increases, the validation accuracy does not seem to increase as well. Let's check the performance on the test set:

```
test_accuracy = model.evaluate(X_test_sc, y_test_cat)[1]
test_accuracy
```

```
10000/10000 [=====] - 0s 32us/step
```

0.9782

and let's compare it with the performance on the training set:

```
train_accuracy = model.evaluate(X_train_sc, y_train_cat)[1]
train_accuracy
```

```
60000/60000 [=====] - 2s 32us/step
```

0.9937666666666667

The performance on the test set is lower than the performance on the training set.

TIP: one question you may have is "When is a difference between the test and train scores significant". We can answer this question by running a cross-validation to see what the standard deviation of each score is. Then we can compare their difference between the two scores with the standard deviation and see if their difference is much greater than the statistical fluctuations of each score.

This difference between the train and test scores may indicate we are overfitting.

This makes sense, because the model is trained using the individual pixels as features. This implies that two images which are similar but slightly rotated or shifted have completely different features.

In order to go beyond "pixels as features" we need to extract better features from the images.

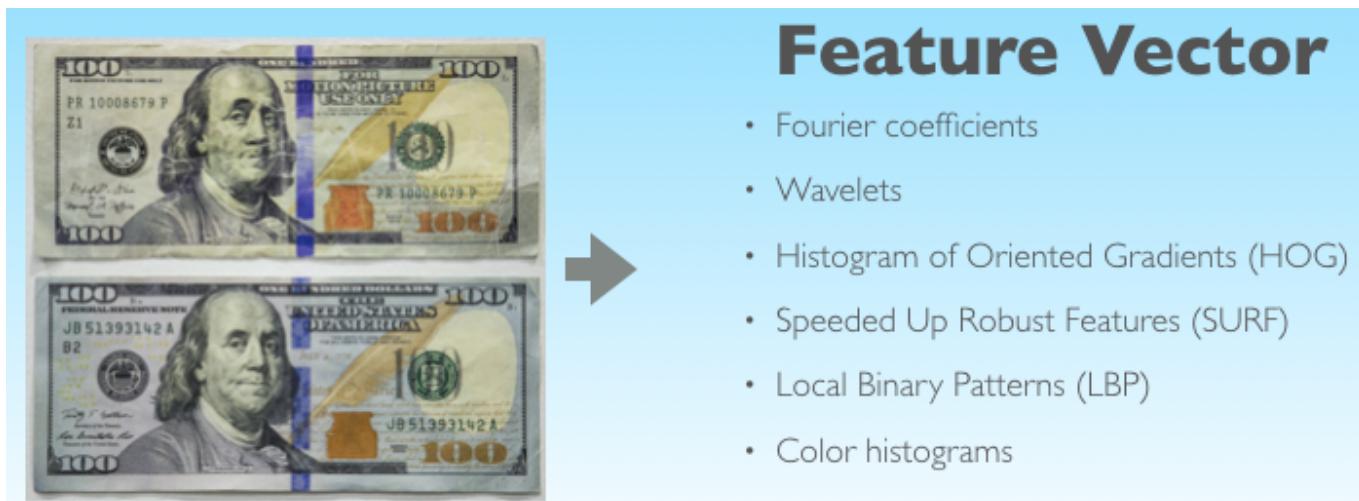
BEYOND PIXELS AS FEATURES

In the previous example we trained a model to recognize handwritten digits using the raw values of the pixels as input features. The model performed pretty well on the training data, but had some trouble generalizing to the test set. Intuitively it's quite clear where the problem is: the absolute value of each pixel is not a great feature to describe the content of an image. To understand this, ask yourself if you would still recognize the digits if black turned to gray and white turned to a lighter gray. Yes you would, despite the fact that the value of each pixel has changed. This is because an image carries information in the arrangements of nearby pixels, not just in the value of each pixel.

It is legitimate to wonder if there is a better way to extract information from images, and there is.

The process of going from an image to a vector of pixels is just the simplest case of **feature extraction** from an image. There are many other methods to extract features from images, including **Fourier transforms**, **Wavelet transforms**, **Histograms of oriented gradients (HOG)** and many others. These are all methods that take an image in input and return a vector of numbers that can be used as features.

The banknotes dataset we used in the previous chapter is an example of features extracted from images with these methods.



Although really powerful, these methods require very deep domain knowledge and each was developed over time to solve a specific problem in image recognition. It would be great if we could avoid using these special methods and just learn the best features from the image problem itself.

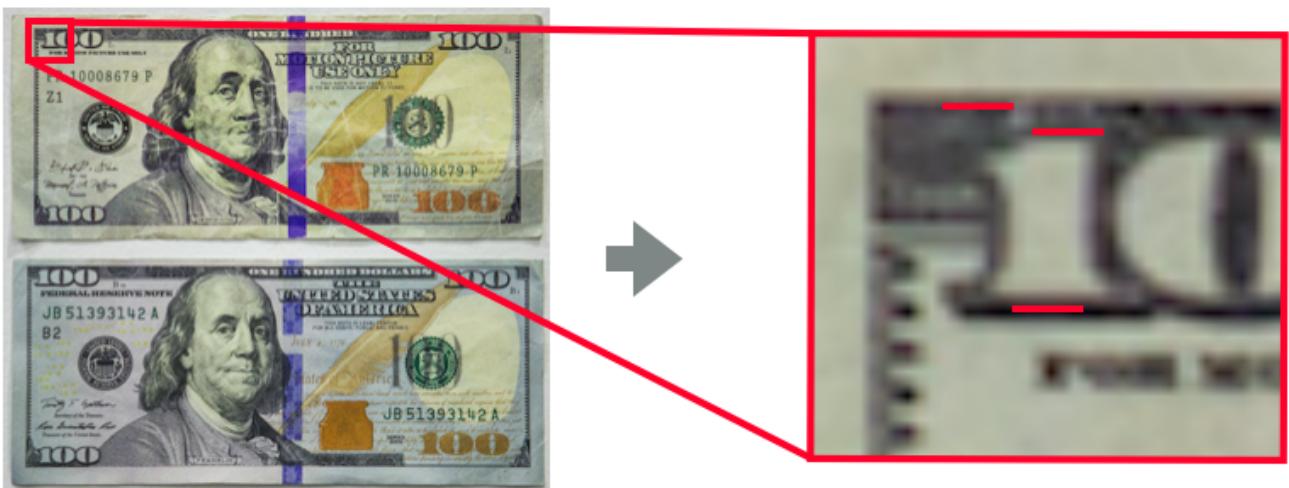
This case is a general issue with feature engineering: identifying features that correctly represent the type of information we are trying to capture from a rich data point (like an image), is a time consuming and complex effort, often involving several Ph.D. students doing their thesis on it.

Let's see if we can use a different approach.

Using local information

Let's consider an image more in detail. What makes an image different from a vector of numbers is that the values of pixels are correlated both horizontally and vertically. It's the 2D pattern that carries the information about what's represented in the image and these 2D patterns, like for example horizontal and vertical contrast lines, are specific to an image or to a set of images. It would be great to have a technique that is able to capture them automatically.

Additionally, if all we care about is recognizing an object, we should strive to be insensitive to the position of the object in the image, and our features should rely more on local patterns of pixels arranged in the form of the object, than on the position of such pixels on the grid.



The mathematical operation that allows us to look for local patterns is called **convolution**. But before we learn about it, we have to take a moment and learn about **Tensors**.

Images as tensors

In this section we talk about tensors. Tensors were originally introduced in Physics and they are a very powerful mathematical tool, they are so important that Einstein used them to describe general relativity. Yeah!

That's right, space-time curvature and gravity are described by tensors!

Despite this, the tensors used in Machine Learning are not the same as the ones used in physics. **Tensors in Machine Learning** really is just a synonym of **Multi-dimensional array**. This is somewhat misleading and has generated a bit of a debate ([see here](#)), but we will proceed as the mainstream convention and use the word tensors to refer to multi-dimensional arrays.

In this sense the **order** or **rank** of a tensor refers to the number of axes in the array.

TIP: people tend to use the word **dimension** to indicate the rank of a tensor (number of axes) as well as the length of a specific axis. We will call the former rank or order, saving the word dimension for the latter. More on this later, however.

You may wonder why you should learn about tensors. The answer is, they allow you to apply machine learning to multi-dimensional data like images, movies, text and so on. Tensors are a great way to extend our skills beyond the tabular datasets we've used so far!

Let's start with **scalars**. Scalars are just numbers, everyday numbers we are used to. They have no dimension.

5

5

Vectors can be thought of as lists of numbers. The number of elements in a vector is also called **vector length** and sometimes **number of dimensions**. As already seen many times, in `python` we can create vectors using the `np.array` constructor:

```
v = np.array([1, 3, 2, 1])
v.shape
```

(4,)

We've just created a vector with 4 elements.

TIP: In our terminology this is a vector of dimension 4, which is still a tensor of order 1, since it only has 1 axis.

The numbers in the list are the coordinates of a point in a space with the same number of dimensions as the number of entries in the list.

Going up one level, we encounter tensors of order 2, which are called matrices. **Matrices** are tables of numbers with rows and columns, i.e. they have 2 axes.

```
M = np.array([[1, 3, 2, 2],  
             [0, 1, 3, 2]])  
M.shape
```

(2, 4)

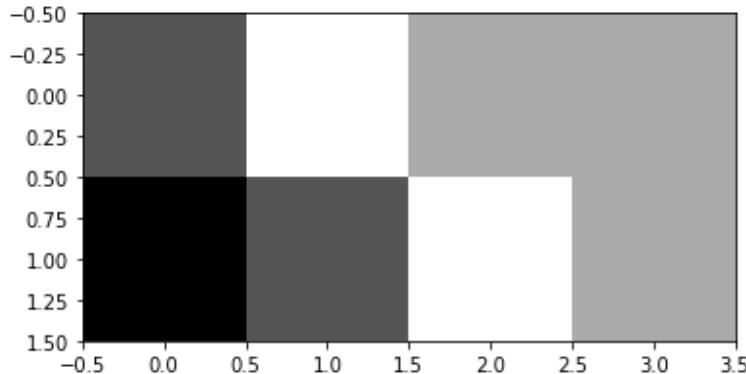
The first axis of `M` has length 2, which is the number of rows in the matrix, the second axis has length 4 and it corresponds to the columns in the matrix.

A grayscale image, as we saw, is a 2D matrix where each entry in the matrix corresponds to a pixel.

TIP: notice that `plt.imshow` takes care of normalizing the values of the matrix so that they can be displayed in gray-scale.

```
plt.imshow(M, cmap='gray')
```

<matplotlib.image.AxesImage at 0x7f74fc15abe0>



Notice also that a matrix can be thought of as a list of vectors of the same length, each representing a row of pixels. In the same way, as a vector can be seen as a list of scalars. So if we extract the first element of the

matrix, this is a vector:

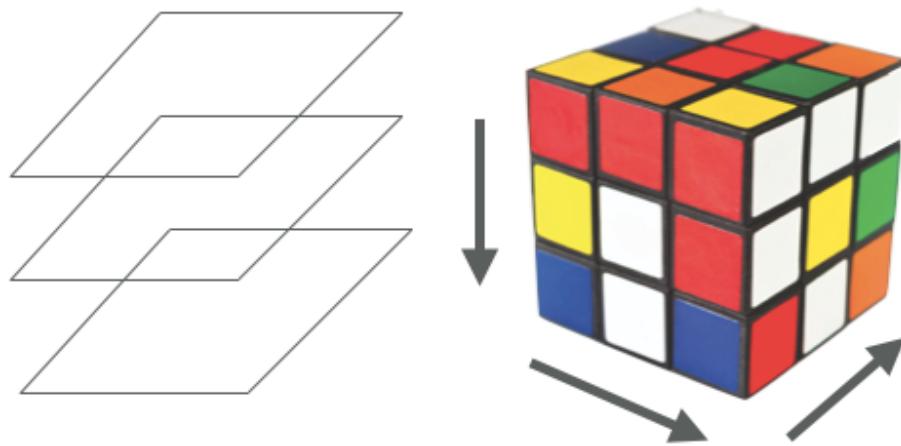
```
M[0].shape
```

(4,)

This recursive construction allows us to organize them in the larger family of tensors. Tensors can be understood as nested lists of objects of the previous order, all with the same shape.

Order	Name	Example
0	Scalar	3
1	Vector	[4, 5, 0, 3, 1, 4, 5]
2	Matrix	[[0, 1, 0], [5, 0, 2]]
3	Tensor	[[[0, 1, 0], [5, 0, 2]], [[1, 2, 4], [8, 3, 1]]]

So for example, a tensor of order 3 can be thought-of as an array of matrices, which are tensors of order two. Since all of these matrices have the exact same number of rows and columns, the tensor is actually like a cuboid of numbers.



Each number is located by the row, the column and the depth where it's stored.

The shape of a tensor tells us how many objects there are when counting along a particular axis. So for example, a vector has only one axis and a matrix has two axes, indicating the number of rows and the number of columns. Since most of the data (images, sounds, texts, etc.) we will use are stored as tensors, it is very important to know the dimensions of these objects for a proper use.

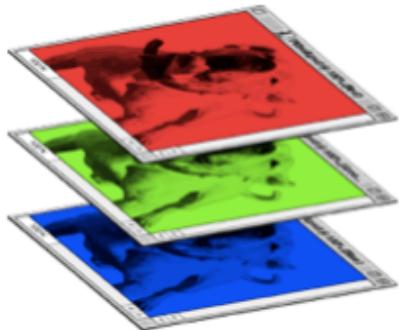
Colored images

A colored image is actually a set of gray-scale images each corresponding to a primary color channel. So, in the case of **RGB**, we have three channels (Red, Blue and Green), each containing the pixels of the image in that particular channel.

This image is an order three tensor and there are two major ordering conventions. If we think of the image as a list of three single color images, then the axis order will be channel first, then height, and then width.

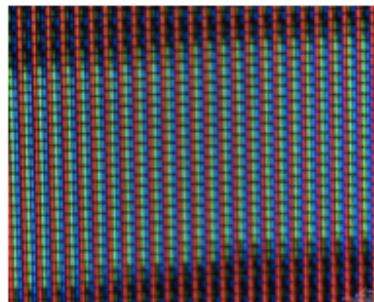
On the other hand, we can also think of the tensor as an order two list of vector pixels, where each pixel contains three numbers, one for each of the colors. This is called "channel last" and it's the convention used in the rest of the book.

(C, H, W)



(H, W, C)

VS



Let's create and display a random color image by creating a list of random pixels between 0 and 255:

```
img = np.random.randint(255, size=(4, 4, 3), dtype='uint8')
img
```

```
array([[[105, 153, 30],
       [247, 69, 237],
       [106, 67, 2],
       [106, 26, 97]],

      [[151, 0, 152],
       [112, 63, 29],
       [12, 162, 251],
       [78, 13, 13]],

      ...,
      [100, 213, 135]]], dtype=uint8)
```

Now let's display it as a figure, showing each of the dominant pixels in each list.

```
plt.figure(figsize=(5, 5))
plt.subplot(221)
plt.imshow(img)
plt.title("All Channels combined")

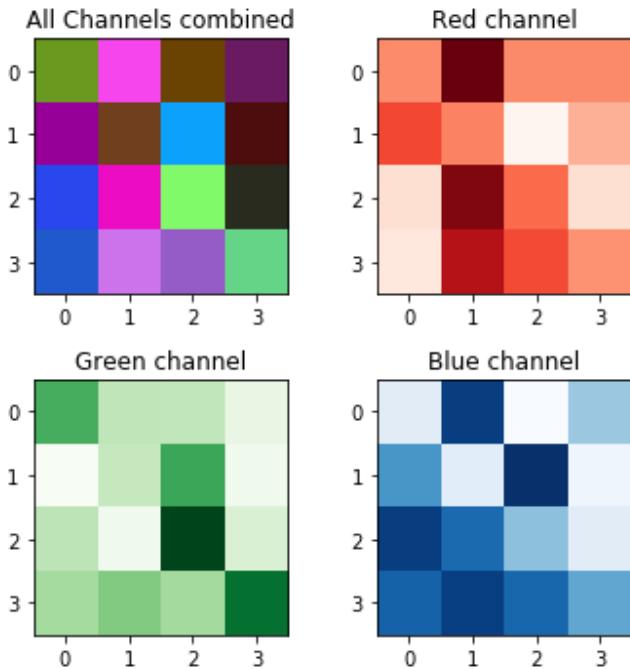
plt.subplot(222)
plt.imshow(img[:, :, 0], cmap='Reds')
plt.title("Red channel")
```

```

plt.subplot(223)
plt.imshow(img[:, :, 1], cmap='Greens')
plt.title("Green channel")

plt.subplot(224)
plt.imshow(img[:, :, 2], cmap='Blues')
plt.title("Blue channel")
plt.tight_layout()

```



Pause here for a second and observe how the colors of the pixels in the colored image reflect the combination of the colors in the three channels.

Now that we know how to represent images using tensors, we are ready to introduce convolutional neural networks.

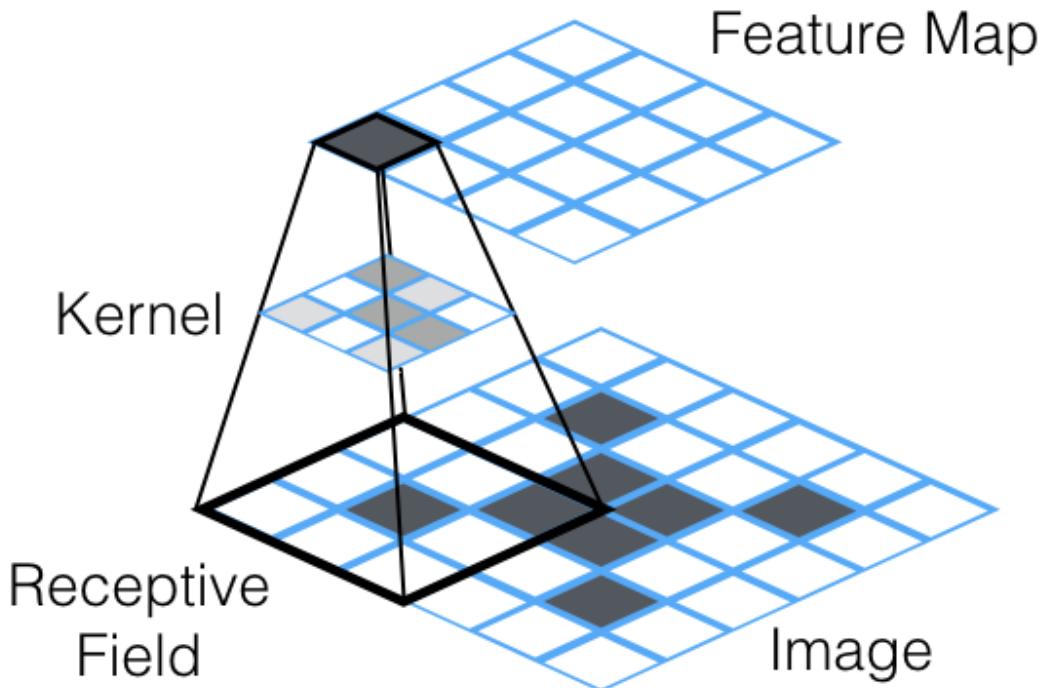
TIP: If you'd like to know a bit more about Tensors and how they work, we displayed a few operations in the [Appendix](#).

CONVOLUTIONAL NEURAL NETWORKS

Simply stated, convolutional Neural Networks are Neural Networks that replace the matrix multiplication operation ($X \cdot w$) with the [convolution](#) operation ($X * w$) in at least one of their layers.

TIP: if you need a refresher about convolutions and how they work, have a look at the [Appendix](#).

For the purpose of this chapter, all we need to know is that an image can be convolved with a **filter** or **kernel**, which is basically a smaller image. The convolution of an image with a kernel generates a new images, also called a **feature map**, whose pixels represent the "degree of matching" of the corresponding **receptive field** with the kernel.



So, if we take many filters and arrange them in a **convolutional layer** the output of the convolution of an image will be as many feature maps (convolved images) as there are filters. Since all of these images have the same size, we can arrange them in a tensor, where the number of channels corresponds to the number of filters used. In fact, let's use tensors to describe everything: inputs, layers and outputs.

We can arrange the input data as a tensor of order four. A single image is an order-3 tensor as we know, but since we have many input images in a batch, and they all have the same size, we might as well stack them in

an order-4 tensor where the first axis indicates the number of samples.

So the 4 axis are respectively: number of images in the batch, height of the image, width of the image and number of color channels in the image.

For example, in the MNIST training dataset we have 60000 images, each with 28x28 pixel and only one color channel, because they are grayscale. This gives us an order-4 tensor with the shape $(60000, 28, 28, 1)$.

Input: order 4 tensor

$$(N, H, W, C)$$

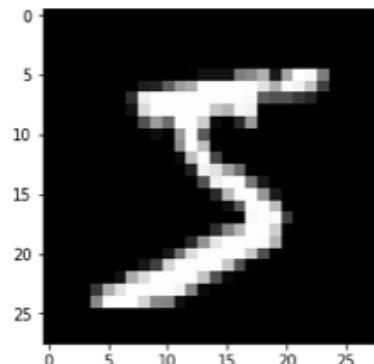
N: Number of images

H: Height of image

W: Width of image

C: Number of color channels

MNIST training set



$$(60000, 28, 28, 1)$$

Similarly, we can stack the filters in the convolutional layer as an order-4 tensor. We will use the first two axes for the height and the weight of the filter. The third axis will correspond to the number of color channels in the input, while the last axis is for the number nodes in the layer, i.e. the number of different filters we are going to learn. This is also called number of **output channels** sometimes, you'll soon see why.

Let's do an example where we build a convolutional layer with four 3×3 filters. The order-4 tensor has a shape of $(3, 3, 1, 4)$, i.e. four filters of 3×3 pixels each with a single input color channel each.

CONV: order 4 tensor

(Hf, Wf, Ci, Co)

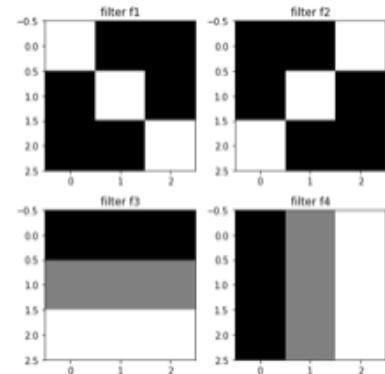
Hf: Height of filter patch

Wf: Width of filter patch

Ci: Channels in input

Co: Channels in output (# filters)

4 filters, 3x3x1



(3, 3, 1, 4)

When we convolve each input image with the convolutional layer, we still obtain an order-4 tensor.

Output: order 4 tensor

(N, H, W, C)

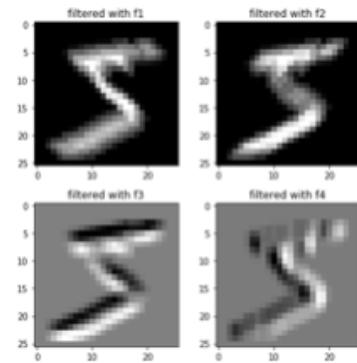
N: Number of images

H: Height of image

W: Width of image

C: Number of color channels

4 output channels



(60000, 26, 26, 4)

The first axis is still the number of images in the batch or in the dataset. The other three axes are for the image height, width and number of color channels in the output. Notice that this is also the number of filters in the layer, four in the case of this example.

Notice that since the output is an order-4 tensor, we could feed it to a new convolutional layer, provided we make sure to match the number of channels correctly.

Convolutional Layers

Convolutional layers are available in `keras.layers.Conv2D`. Let's apply a convolutional layer to an image and see what happens.

First, let's import the `conv2D` layer from keras:

```
from keras.layers import Conv2D
```

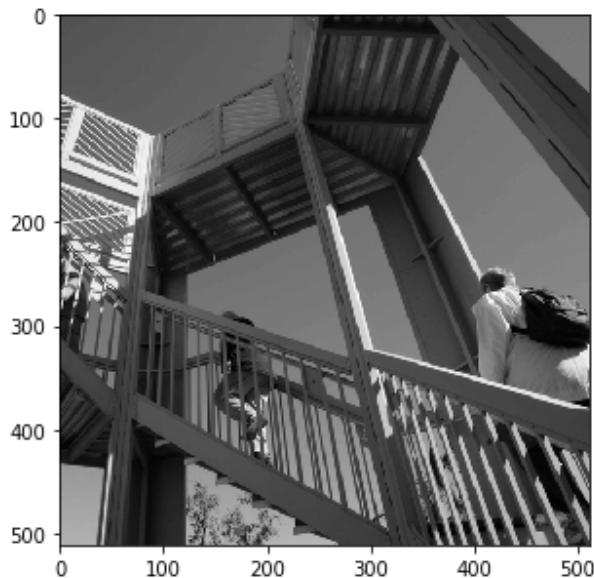
Now let's load an example image from `scipy.misc`:

```
from scipy import misc
```

```
img = misc.ascent()
```

and let's display it:

```
plt.figure(figsize=(5, 5))
plt.imshow(img, cmap='gray')
plt.show()
```



Let's check the shape of `img`:

```
img.shape
```

(512, 512)

A convolutional layer wants an order-4 tensor as input, so first of all we need to reshape our image so that it has 4 axes and not 2.

We can just add 1 axis of length 1 for the color channel (which is a grayscale pixel value between 0 and 255) and 1 axis of length 1 for the dataset index.

```
img_tensor = img.reshape((1, 512, 512, 1))
```

Let's start by applying a large flat filter of size 11x11 pixels, this should result in a blurring of the image because the pixels are averaged.

The syntax of Conv2D is:

```
Conv2D(filters, kernel_size, ...)
```

so we will specify 1 for the filter and (11, 11) for the kernel_size . We will also initialize all the weights to one by using kernel_initializer='ones' . Finally we will need to pass the input shape, since this is the first layer in the network. This is the shape of a single image, which in this case is (512, 512, 1) .

```
model = Sequential()
model.add(Conv2D(1, (11, 11), kernel_initializer='ones', input_shape=(512, 512, 1)))
model.compile('adam', 'mse')
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 502, 502, 1)	122

Total params: 122
Trainable params: 122
Non-trainable params: 0

...

We have a model with one convolutional layer, so the number of parameters is equal to $11 \times 11 + 1$ where the +1 comes from the bias term. We can apply the convolution to the image by running a forward pass:

```
img_pred_tensor = model.predict(img_tensor)
```

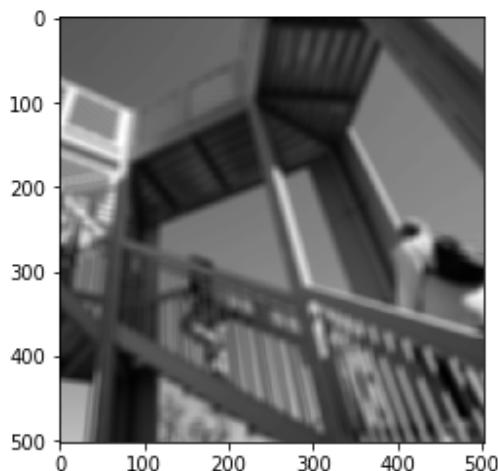
To visualize the image we extract it from the tensor.

```
img_pred = img_pred_tensor[0, :, :, 0]
```

and we can use `plt.imshow` as before:

```
plt.imshow(img_pred, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f74a0172f60>
```



As you can see the image is blurred, as we expected.

TIP: try to change the initialization of the convolutional layer to something else. Then re-run the convolution and notice how the output image changes.

Great! We have just demonstrated that the convolution with a kernel will produce a new image, whose pixels will be a combination of the original pixels in a receptive field and the values of the weights in the kernel.

These weights are not decided by the user, they are learned by the network through backpropagation! This allows a Neural Network to adapt and learn any pattern that is relevant to solving the task.

There are two additional arguments to consider when building a convolutional layer with keras: `padding` and `stride`.

Padding

If you've been paying attention, you may have noticed that the convolved image is slightly smaller than the original image:

```
img_pred_tensor.shape
```

```
(1, 502, 502, 1)
```

This is due to the default setting of `padding='valid'` in the `Conv2D` layer and it has to do with how we treat the data at the boundaries. Each pixel in the convolved image is the result of the contraction of the receptive field with the kernel. Since in this case the kernel has a size of 11x11, if we start at the top left corner and slide to the right there are only 502 possible positions for the receptive field. In other words we lose 5 pixels on the right and 5 pixels on the left.

If we would like to preserve the image size, we need to offset the first receptive field so that its center falls on the top left corner of the input image. We can fill the empty parts with zeros. This is called padding.

0	0	0	0	0	0	0	0	0	0	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
0	0	0	0	0	0	0	0	0	0	0

In keras we have two padding modes:

- valid which means **no padding**
- same which means **pad** to keep the same image size.

Let's check that *padding same* works as expected:

```
model = Sequential()
model.add(Conv2D(1, (11, 11), padding='same',
                kernel_initializer='ones', input_shape=(512, 512, 1)))
model.compile('adam', 'mse')

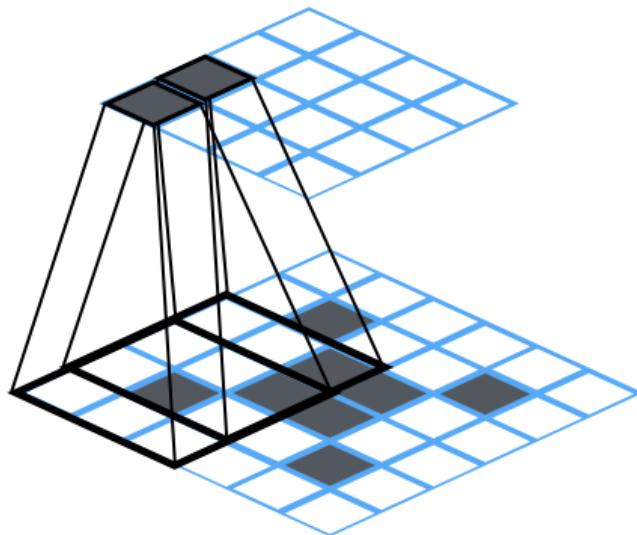
model.predict(img_tensor).shape
```

(1, 512, 512, 1)

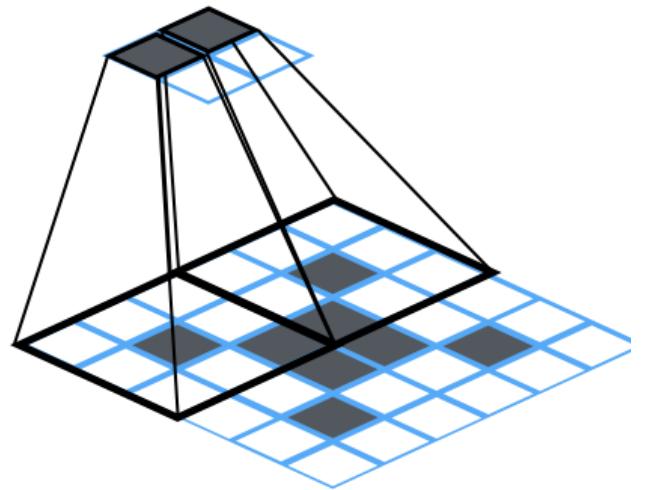
Awesome! We know how padding works. Why use padding? We can use padding if we think that the pixels at the border contain useful information to solve the classification task.

Stride

The **stride** is the number of pixels that we use that separate one receptive field from the next. It's like the step size in the convolution. A stride of $(1, 1)$ means we slide the receptive field by one pixel horizontally and one vertically. Looking at the figure:



Stride = 1



Stride = 3

The input image has size 6×6 , the filter (not shown) is 3×3 and so is the receptive field. If we perform a convolution with no padding and stride of 1, the output image will lose one pixel on each side, resulting in a 4×4 image. Increasing the stride means skipping a few pixels between one receptive field and the next, so for example a stride of $(3, 3)$, will produce an output image of 2×2 .

We can also stride of different length in the two directions, which will produce a rectangular convolved output image.

Finally, if we don't want to lose the borders during the convolution we can pad the image with zeros and obtain an image with the same size as the input.

The default value for the stride is 1 to the right and 1 down, but we can jump by larger amounts, for example if the image resolution is too high.

This will produce output images that are smaller. For example, let's jump by 5 pixels in both directions in our example:

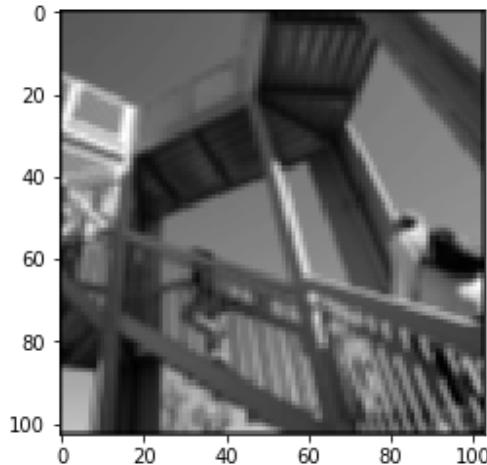
```
model = Sequential()
model.add(Conv2D(1, (11, 11), strides=(5, 5), padding='same',
                kernel_initializer='ones', input_shape=(512, 512, 1)))
model.compile('adam', 'mse')

small_img_tensor = model.predict(img_tensor)
small_img_tensor.shape
```

```
(1, 103, 103, 1)
```

```
plt.imshow(small_img_tensor[0, :, :, 0], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f74a0026c18>
```



The image is still present, but its resolution is now much lower. We can also choose asymmetric strides, if we believe the image has more resolution in one direction than another:

```
model = Sequential()
model.add(Conv2D(1, (11, 11), strides=(11, 5), padding='same',
                kernel_initializer='ones', input_shape=(512, 512, 1)))
model.compile('adam', 'mse')
```

```
asym_img_tensor = model.predict(img_tensor)
asym_img_tensor.shape
```

```
(1, 47, 103, 1)
```

Pooling layers

Another layer we need to learn about is the pooling layer.

Pooling reduces the size of the image by discarding some information. For example, max-pooling only preserves the maximum value in a patch and stores it in the new image, while discarding the values in the other pixels.

Also, pooling patches usually do not overlap, so that the size of the image is actually reduced.

If we apply pooling to the feature maps, we end up with smaller feature maps, that still retain the highest matches of our convolutional filters with the input.

Average pooling is similar, only using average instead of max.

These layers are available in `keras` as `MaxPooling2D` and `AveragePooling2D`.

```
from keras.layers import MaxPooling2D, AveragePooling2D, GlobalMaxPooling2D
```

Let's add a `MaxPooling2D` layer in a simple network (containing this single layer):

```
model = Sequential()
model.add(MaxPooling2D(pool_size=(5, 5), input_shape=(512, 512, 1)))
model.compile('adam', 'mse')
```

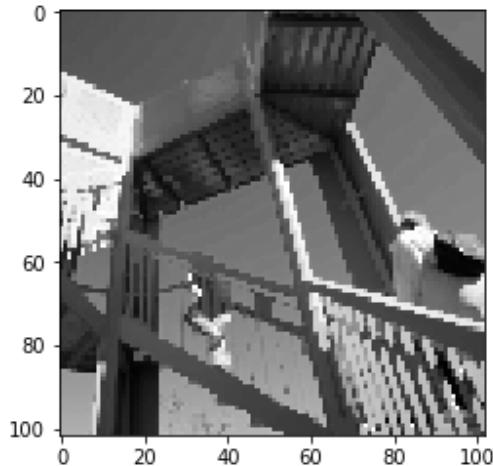
and let's apply it to our example image:

```
img_pred = model.predict(img_tensor)[0, :, :, 0]
img_pred.shape
```

(102, 102)

```
plt.imshow(img_pred, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f74681c32e8>
```



Max-pooling layers are useful in tasks of object recognition, since pixels in feature maps represent the "degree of matching" of a filter with a receptive field, keeping the max keeps the highest matching feature.

On the other hand, if we are also interested in the location of a particular match, then we shouldn't be using max-pooling, because location information will be lost in the pooling operation.

Thus, for example if we are using a convolutional Neural Network to read the state of a video game from a frame we need to know the exact positions of players and thus using max-pooling is not recommended.

Finally `GlobalMaxPooling2D` calculates the global max in the image, so it returns a single value for the image:

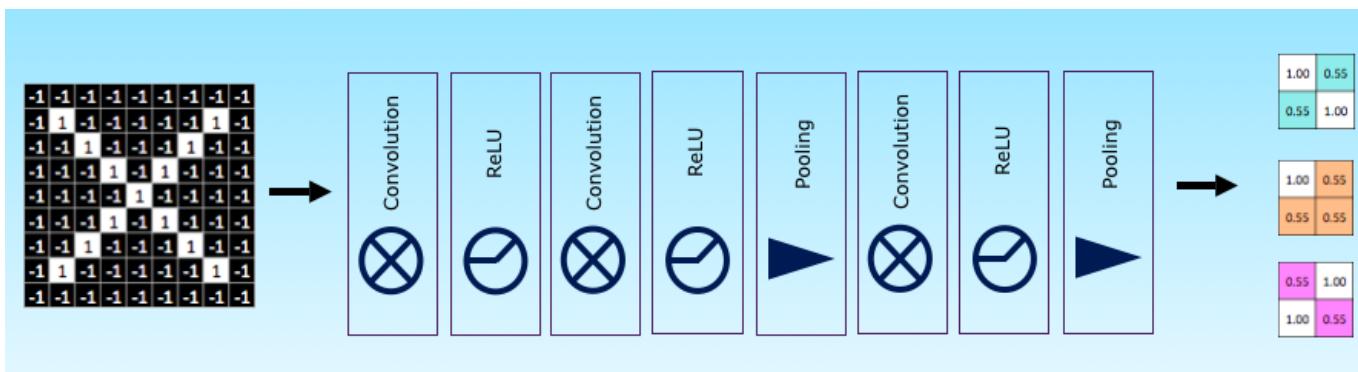
```
model = Sequential()
model.add(GlobalMaxPooling2D(input_shape=(512, 512, 1)))
model.compile('adam', 'mse')
```

```
img_pred_tensor = model.predict(img_tensor)
img_pred_tensor.shape
```

(1, 1)

Final architecture

Convolutional, pooling and activation layers can be stacked together. The output of one layer can be fed into the next resulting in an feature extraction pipeline that will gradually transform an image into a tensor with more channels and less pixels:



The value of each "pixel" in the last feature map is influenced by a large regions of the original image and it will have learned to recognize complex patterns.

In fact, that's the beauty of stacking convolutional layers. The first layers will learn patterns of pixels in the original image, while deeper layers will learn more complex patterns that are combinations of the simpler patterns.

In practice, early layers will specialize to recognize contrast lines in different orientations, while deeper layers will combine those contrast lines to recognize parts of objects. The typical example of this is the face recognition task where middle layers recognize facial features like eyes, noses and mouths while deeper nodes specialize on individual faces.

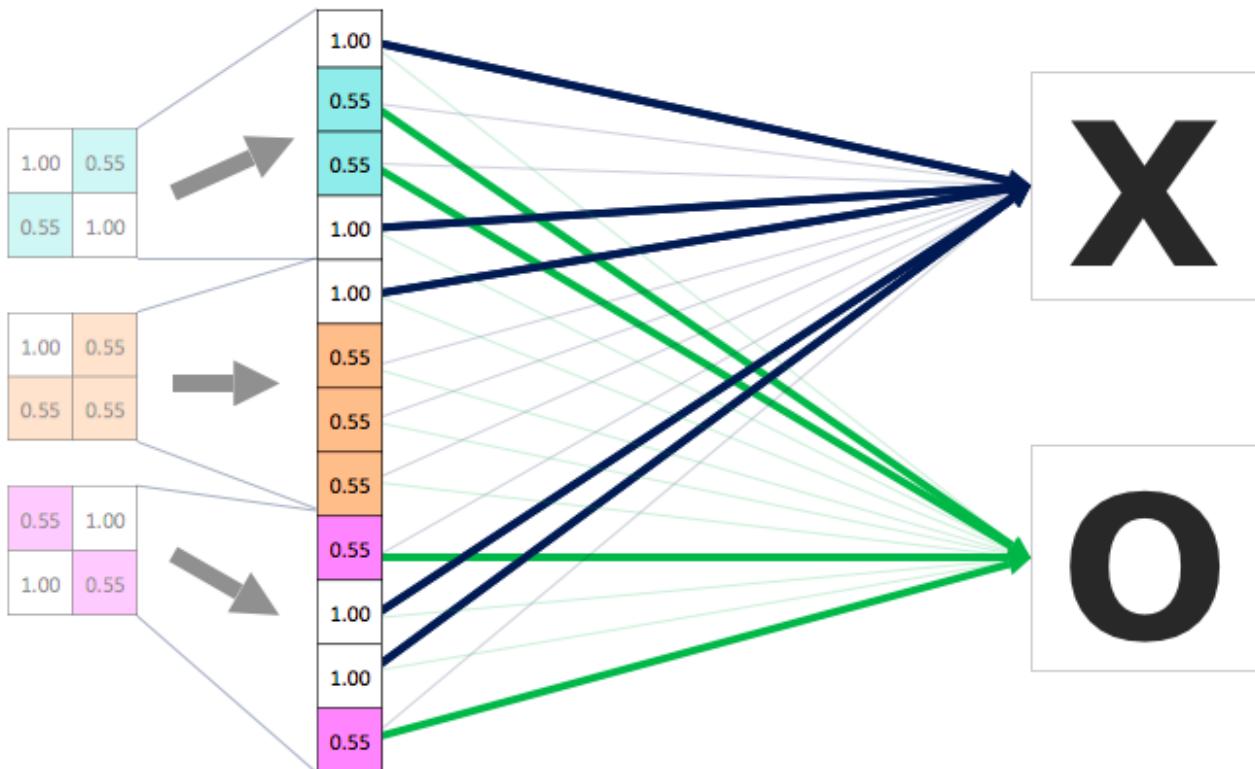
The convolutional stack behaves like an optimized feature extraction pipeline that is trained to optimally solve the task at hand.

In order to complete the pipeline and solve the classification task we can pipe the output of the feature extraction pipeline into a fully connected final stack of layers.

We will need to unroll the output tensor into a long vector like we did initially for the MNIST data, and connect this vector to the labels using a fully connected network.

Flatten

Fully connected



We can also stack multiple fully connected layers if we want. Our final network is like a pancake of many layers, the convolutional part dealing with feature extraction and the fully connected part handling the classification.

The deeper we go in the network the richer and more unique are the patterns matched and so more robust the classification will be.

Convolutional network on images

Let's build our first convolutional Neural Network to classify the MNIST data. First of all we need to reshape the data as order-4 tensors. We will store the reshaped data into new variables called `x_train_t` and `x_test_t`.

```
X_train_t = X_train_sc.reshape(-1, 28, 28, 1)
X_test_t = X_test_sc.reshape(-1, 28, 28, 1)
```

```
x_train_t.shape
```

```
(60000, 28, 28, 1)
```

Then we import the `Flatten` and `Activation` layers:

```
from keras.layers import Flatten, Activation
```

Let's now build a simple model with the following architecture:

- A `conv2D` layer with 32 filters of size 3x3.
- A `MaxPooling2D` layer of size 2x2.
- An activation layer with a `ReLU` activation function.
- A couple of fully connected layers leading to the output of 10 classes corresponding to the digits.

Notice that between the convolutional layers and the fully connected layers we will need `Flatten` to reshape the feature maps into feature vectors.

In order to speed up the convergence we initialize the convolutional weights drawing from a random normal distribution. Later in the book we will discuss intializations more in detail.

Also notice that we need to pass `input_shape=(28, 28, 1)` to let the model know our input images are grayscale 28x28 images:

```
model = Sequential()

model.add(Conv2D(32, (3, 3), kernel_initializer='normal', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
```

```
        optimizer='rmsprop',
        metrics=['accuracy'])

model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_5 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 32)	0
activation_1 (Activation)	(None, 13, 13, 32)	0
<hr/>		
...		
<hr/>		

This model has 300k parameters, that's almost half of the the fully connected model we designed at the beginning of this chapter. Let's train it for 5 epochs. Notice that we pass the tensor data we created above:

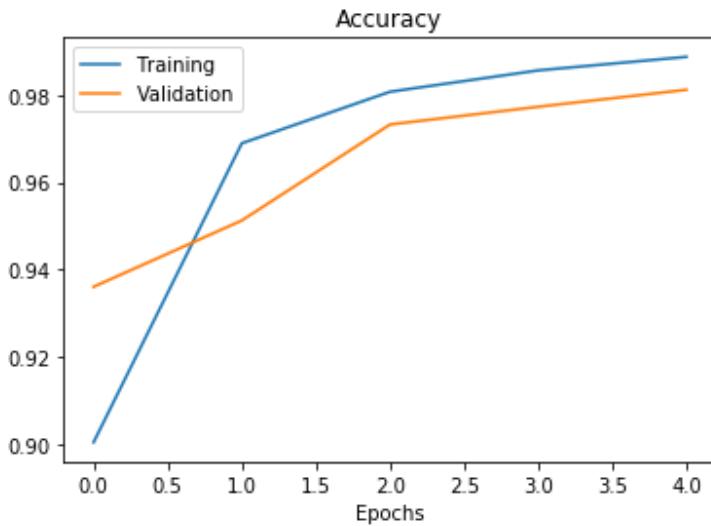
```
h = model.fit(X_train_t, y_train_cat, batch_size=128,
               epochs=5, verbose=1, validation_split=0.3)
```

```
Train on 42000 samples, validate on 18000 samples
Epoch 1/5
42000/42000 [=====] - 2s 48us/step - loss: 0.3374 - acc: 0.9006
Epoch 2/5
42000/42000 [=====] - 2s 41us/step - loss: 0.1090 - acc: 0.9689
Epoch 3/5
42000/42000 [=====] - 2s 41us/step - loss: 0.0668 - acc: 0.9807
Epoch 4/5
42000/42000 [=====] - 2s 42us/step - loss: 0.0495 - acc: 0.9856
...
42000/42000 [=====] - 2s 42us/step - loss: 0.0495 - acc: 0.9856
```

Like before, we can display the training history:

```
plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
```

```
plt.title('Accuracy')
plt.xlabel('Epochs')
plt.show()
```



and compare the accuracy on train and test sets:

```
train_acc = model.evaluate(X_train_t, y_train_cat, verbose=0)[1]
test_acc = model.evaluate(X_test_t, y_test_cat, verbose=0)[1]

print("Train accuracy: {:.4f}, Test accuracy: {:.4f}".format(train_acc, test_acc))
```

```
Train accuracy: 0.9893, Test accuracy: 0.9827
```

The convolutional model achieved a better performance on the MNIST data in less epochs. Overfitting is also reduced, because the model is learning to combine spatial patterns instead of learning the exact values of the pixels.

BEYOND IMAGES

Convolutional networks are great on all data types where the order matters. For example, they can be used on sound files using spectrograms. Spectrograms represent sound as an image where the vertical axis corresponds to the frequency bands, while the horizontal axis indicates the time. We can feed spectrograms to a convolutional layer and treat it like an image. Some of the most famous speech recognition engines use this technique.

Similarly, we can map a sentence of text onto an image where the vertical axis indicates the word index in a vocabulary, and the horizontal axis is for the position in the sentence.

Although they are very powerful, CNNs are not useful at all in some cases. Since they are good at capturing spatial patterns, they are of no use when such local patterns do not exist. This is the case when data is a 2D table coming from a database collecting user data. Each row corresponds to a user and each column to a feature, but there is no special order in either columns or rows.

In other words we can swap the order of the rows or the columns without altering the information contained in the table. In a case like this, a CNN is completely useless and it should not be used.

CONCLUSION

In this chapter we've finally introduced convolutional Neural Networks as a tool to efficiently extract features from images and more generally from spatially correlated data.

Convolutional networks are ubiquitous in object recognition tasks, widely used in robotics, self-driving cars, advertising, and many more fields.

EXERCISE

Exercise 1

You've been hired by a shipping company to overhaul the way they route mail, parcels and packages. They want to build an image recognition system capable of recognizing the digits in the zipcode on a package, so that it can be automatically routed to the correct location.

You are tasked to build the digit recognition system. Luckily, you can rely on the MNIST dataset for the initial training of your model!

Build a deep convolutional Neural Network with at least two convolutional and two pooling layers before the fully connected layer:

- start from the network we have just built
- insert one more `Conv2D` , `MaxPooling2D` and `Activation` pancake, you will have to choose the number of filters in this convolutional layer
- retrain the model
- does performance improve?
- how many parameters does this new model have? More or less than the previous model? Why?
- how long did this second model take to train? Longer or shorter than the previous model? Why?
- did it perform better or worse than the previous model?

Exercise 2

Pleased with your performance with the digits recognition task, your boss decides to challenge you with a harder task. Their online branch allows people to upload images to a website that generates and prints a postcard that is shipped to destination. Your boss would like to know what images people are loading on the site in order to provide targeted advertising on the same page, so he asks you to build an image recognition system capable of recognizing a few objects. Luckily for you, there's a dataset ready made with a collection of labeled images. This is the [Cifar 10 Dataset](#), a very famous dataset that contains images for 10 different categories:

- airplane
- automobile
- bird

- cat
- deer
- dog
- frog
- horse
- ship
- truck

In this exercise we will reach the limit of what you can achieve on your laptop. In later chapters we will learn how to leverage GPUs to speed up training.

Here's what you have to do:

- load the cifar10 dataset using `keras.datasets.cifar10.load_data()`
- display a few images, see how hard/easy it is for you to recognize an object with such low resolution
- check the shape of `x_train`, does it need reshape?
- check the scale of `x_train`, does it need rescaling?
- check the shape of `y_train`, does it need reshape?
- build a model with the following architecture, and choose the parameters and activation functions for each of the layers:
 - conv2d
 - conv2d
 - maxpool
 - conv2d
 - conv2d
 - maxpool
 - flatten
 - dense
 - output
- compile the model and check the number of parameters
- attempt to train the model with the optimizer of your choice. How fast does training proceed?
- If training is too slow, feel free to stop it and read ahead. In the next chapters you'll learn how to use GPUs to

```
from keras.datasets import cifar10
```


Time Series and Recurrent Neural Networks

In this chapter we will learn mainly about **Recurrent Neural Networks** (or RNN, for short). RNNs expand the architectures we have encountered so far by allowing for feedback loops in time. This property makes RNNs particularly suited to work with ordered data, for example time series, sound and text. These networks are able to generate arbitrary sequences of outputs, opening the door on many new types of supervised learning problems.

Machine Learning on Time Series requires a bit more caution than usual, since we need to avoid leaking future information into the training. We will therefore start this chapter talking about Time Series and Sequence problems in general. Then we will introduce RNNs and in particular two famous architectures: **LSTMs** and **GRUs** (for the latter, have a look at the Exercise 2).

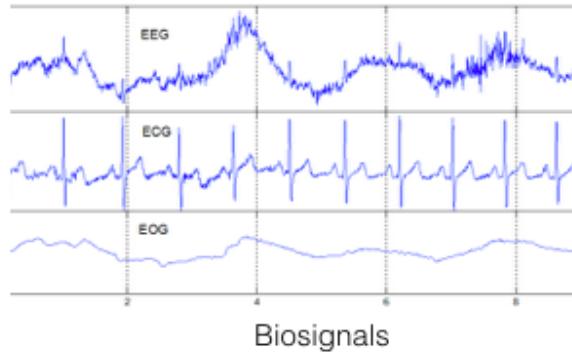
This chapter contains both practical and theoretical parts with some math. Like we did in chapter 5, let us first tell you: **you don't NEED to read the math in this chapter**. This book is meant for the developer and practitioner that is interested in applying neural networks to solve great problems. We provide the math for the curious and we will make sure to highlight which sections can be skipped at a first read.

TIME SERIES

Time series are everywhere. Examples of time series are the values of a stock, music, text, events on your app, video games, which are sequences of actions, and in general any quantity monitored over time that generates a sequence of values.



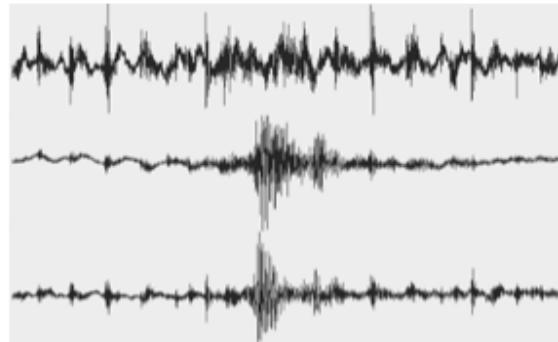
Stock market



Biosignals



Energy consumption



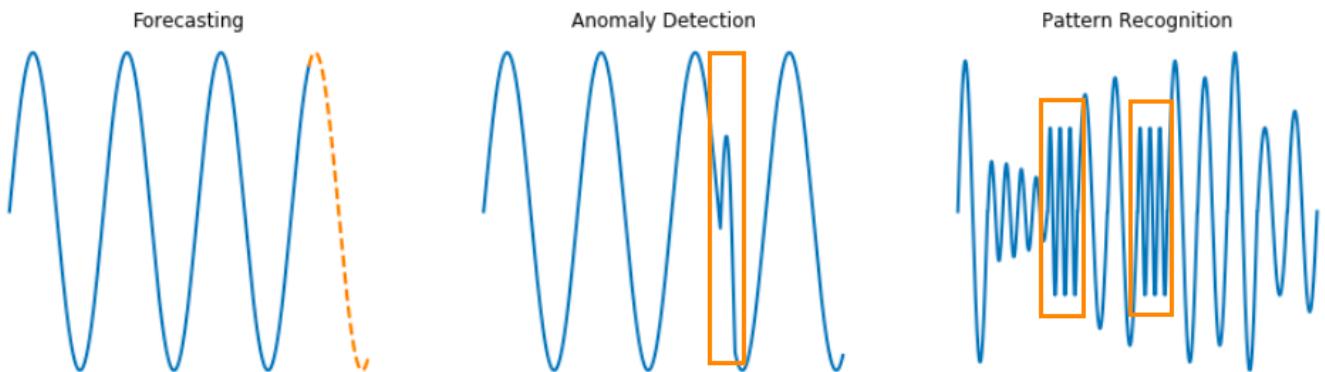
Seismic activity

A time series is an ordered sequence of data points, and it can be univariate or multivariate.

A **univariate** time series is nothing but a sequence of scalars. Example of this are temperature values through the day or the number of times per minute your app was downloaded.

A time series could also take values in a vector space, in which case it is a **multivariate** time series. Examples of vector time series are the speed of a car as a function of time or an audio file recorded in stereo, which has two channels.

Machine Learning can be applied to time series to solve several problems including forecasting, anomaly detection and pattern recognition.

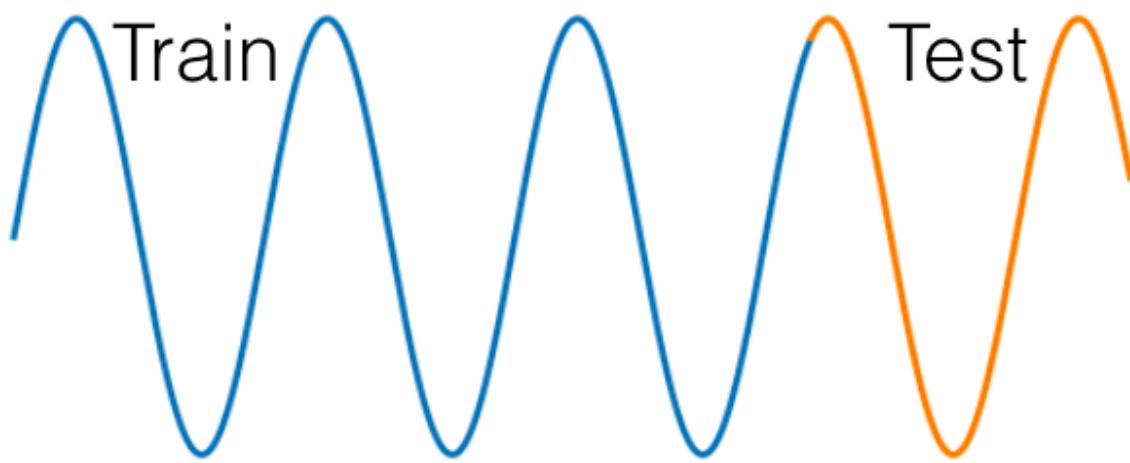


Forecasting refers to predicting future samples in a sequence. In a way, this problem is a regression problem because we are predicting a continuous quantity using features derived from the time series and most likely it is a nonlinear regression.

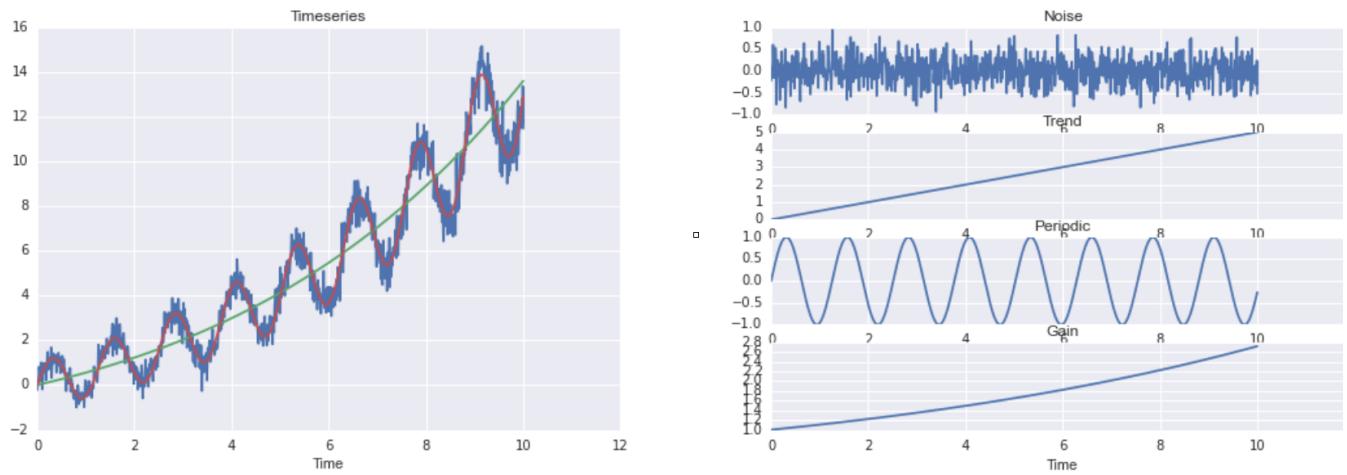
Anomaly detection refers to identifying deviations from a regular pattern. This problem can be approached in two ways: if we know the anomalies we are looking for, we can approach it as a classification problem. If we do not know the anomalies we would just train a model to forecast future values (regression) and then compare the predicted value and the actual signal. In this case, anomalies are located where the model prediction is very different from the actual signal.

Pattern recognition is classification on time series, identifying recurring patterns.

In all these cases we must use particular care because the data is ordered in time and we need to avoid leaking future information in the features used by the model. This is particularly true for model validation. If we split the time series into training and test sets we cannot just pick a random split from the time series. We need to **split the data in time**: all the test data should come after the training data.



Also, sometimes a trend or a periodic pattern is clearly distinguishable.



This is particularly true with any data related to human activity, where **daily, weekly, monthly and yearly periodicities** are found.

Think for example of retail sales. A dataset with hourly sales from a shop, will have regular patterns during the day: with period of higher customer flow and period of lower customer flow, as well as during the week. Depending on the type of goods we may find higher or lower sales during the weekend. Special dates, like black Friday or sales days, will appear as anomalies in these regular patterns and should be easy to catch. In these cases, it is a good idea to either remove these periodicities beforehand or to add the relevant time interval as an input feature.

TIME SERIES CLASSIFICATION

As a warm up exercise let's perform a classification on time series data. Let's load the usual libraries, `pandas`, `numpy`, and `matplotlib`:

```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

The file `sequence_classification.csv.bz2` contains a set of 4000 curves. Let's load it:

```
df = pd.read_csv('../data/sequence_classification.csv.bz2', compression='bz2')
df.head()
```

	anomaly	t_0	t_1	t_2	t_3	t_4	t_5	t
0	False	1.000000	0.974399	0.939818	0.906015	0.878538	0.838718	0.8241
1	True	0.626815	0.665145	0.669603	0.693649	0.697292	0.729030	0.7507
2	False	0.983751	0.944806	0.999909	0.975756	0.972598	1.000000	0.9824
3	True	0.977806	1.000000	0.975431	0.966523	0.941594	0.932092	0.9211
4	False	0.691444	0.710671	0.660787	0.690993	0.706655	0.641938	0.8034

5 rows × 201 columns

TIP: this is the first time that we load a zipped file, i.e. a compressed file convenient to save storage space. Pandas allows to load directly zipped file saved in several formats, for example in this case a `bz2` file. Have a look at the [documentation](#) for further details and discover all the formats supported.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4000 entries, 0 to 3999
Columns: 201 entries, anomaly to t_199
dtypes: bool(1), float64(200)
memory usage: 6.1 MB
```

Each row in the dataset is a curve, the labels for anomalies are given in the first column (in this case, we just have two, `True` and `False`).

```
df['anomaly'].value_counts()
```

```
True    2000
False   2000
Name: anomaly, dtype: int64
...
Name: anomaly, dtype: int64
...
Name: anomaly, dtype: int64
```

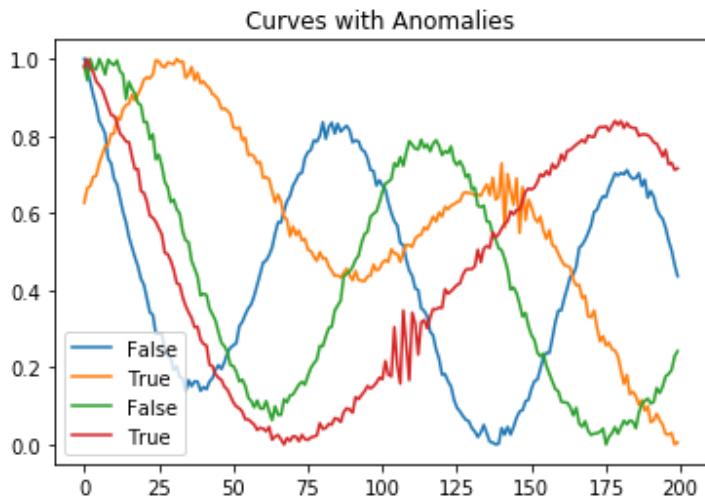
As we can see, 2000 curves present anomalies, while the other 2000 do not. Let's create the `x` and `y` arrays and plot the first 4 curves.

```
X = df.drop("anomaly", axis="columns").values
y = df["anomaly"].values
```

Now let's plot these curves separated by the anomaly values.

```
plt.plot(X[:4].transpose())
plt.legend(y[:4])
plt.title("Curves with Anomalies")
```

```
Text(0.5,1,'Curves with Anomalies')
```



How do we treat this problem with Machine Learning?

We can approach it in various ways.

1. We could use the values of the the curves as features (that is 200 points) and feed them to a fully connected network.
2. We could engineer features from the curves, like statistical quantities, differences and Fourier coefficients and feed those to a Neural Network.
3. We could use a 1D convolutional network to automatically extract patterns from the curves.

Let's quickly try all three.

TIP: if you had to guess, which of the three approaches seems more promising?

First of all, we will perform a train/test split. In this case we do not need to worry about the order in time because the sequences are given to us without any information about their absolute time. For all we know they could be independent measurements of the same phenomenon.

Let's load the `train_test_split` function from `sklearn` first:

```
from sklearn.model_selection import train_test_split
```

Now let's split the data into the training and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
```

Fully connected networks

Let's load our layers from `keras` so we can build our fully connected network:

```
from keras.models import Sequential
from keras.layers import Dense
import keras.backend as K
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: Fu
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Let's also clear the backend of any data it's holding on to with `clear_session()` on the backend:

```
K.clear_session()
```

Finally, let's build our model with our Keras layers, using the 200 points as features. This process should be pretty familiar at this point:

```
model = Sequential()
model.add(Dense(100, input_dim=200, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

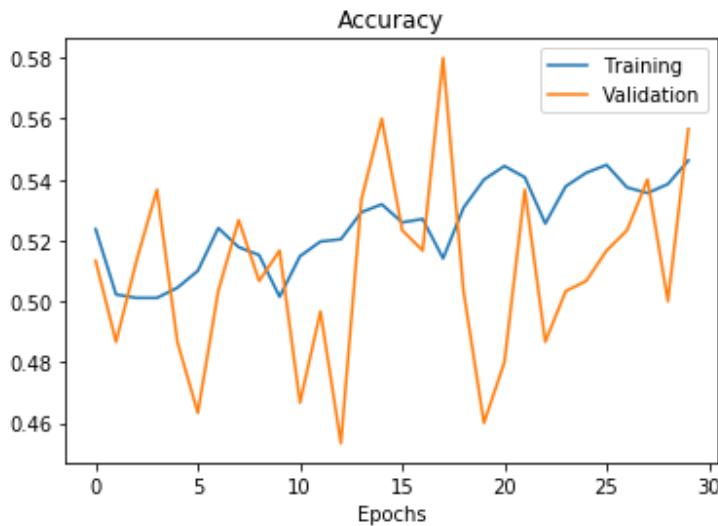
Next, let's train the model to fit against our training set.

```
h = model.fit(X_train, y_train, epochs=30, verbose=0, validation_split=0.1)
```

Now, let's plot the curves of our newly trained model.

```
plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
plt.title('Accuracy')
plt.xlabel('Epochs')
```

```
Text(0.5,0,'Epochs')
```



```
print("Test Accuracy: {:.3f}".format(model.evaluate(X_test, y_test, verbose=0)[1]))
```

```
Test Accuracy: 0.546
```

This model does not seem to perform really well at all (operating around 50% accuracy).

This is easy to understand. In fact, the anomaly can be located anywhere along the curve, making it difficult for a Neural Network to learn about its presence from amplitude values.

Fully connected networks with feature engineering

Let's try to extract some features from the curves. We will limit ourselves to:

- std : standard deviation of the curve values
- std_diff : standard deviation of the first order differences

TIP: feel free to add more features like higher order statistical moments or Fourier coefficients.

First, let's build the new DataFrame `eng_f` containing in the two columns the feature `std` and `std_diff`.

```
eng_f = pd.DataFrame(X.std(axis=1), columns=['std'])
eng_f['std_diff'] = np.diff(X, axis=1).std(axis=1)

eng_f.head()
```

	std	std_diff
0	0.260902	0.023511
1	0.249588	0.030286
2	0.304086	0.023464
3	0.302908	0.030531
4	0.286405	0.066638

We split the data again:

```
eng_f_train, eng_f_test, y_train, y_test = train_test_split(eng_f.values, y, test_size=0.
```

Let's clear out the backend for any memory we've already used:

```
K.clear_session()
```

Next, let's train a fully connected model: as already seen many times, the first layer depends on the number of input features, 2 in this case, and the last layer depends on the output, a binary classification 0/1 in this contest (notice that the last layer is the same of the previous model, since we didn't change our output). The inner layers, only one in this model, depend on the researcher preferences.

```
model = Sequential()
model.add(Dense(30, input_dim=2, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

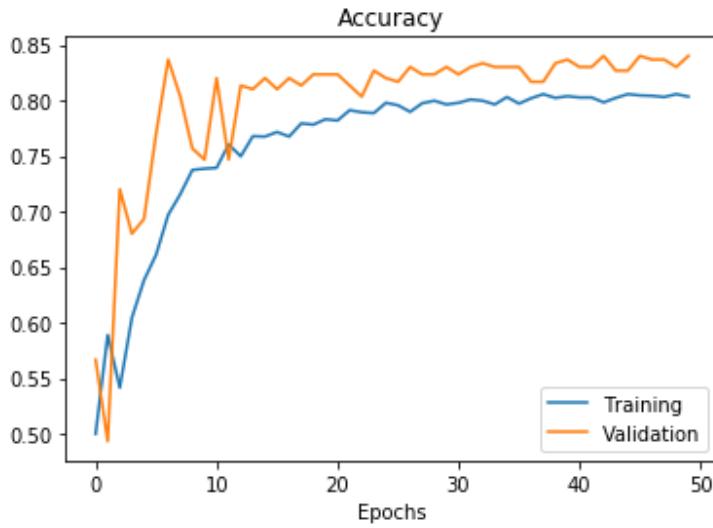
Now let's train our model on our 2 engineered features:

```
h = model.fit(eng_f_train, y_train, epochs=50, verbose=0, validation_split=0.1)
```

Let's plot the output of our model by plotting again:

```
plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
plt.title('Accuracy')
plt.xlabel('Epochs')
```

```
Text(0.5,0, 'Epochs')
```



```
print("Test Accuracy: {:.3f}".format(model.evaluate(eng_f_test, y_test, verbose=0)[1]))
```

```
Test Accuracy: 0.812
```

This model is already much better than the previous one, but can we do better? Let's try with the third approach, i.e. 1D convolutional network to automatically extract patterns from the curves.

Fully connected networks with 1D Convolution

As we know by now, convolutional layers are good for recognizing spatial patterns. In this case we know the anomaly spans across a dozen points along the curve, so we should be able to capture it if we cascade a few `Conv1D` layers with filter size of 3.

TIP: the filter size, 3 in this case, is an arbitrary choice. In the [Appendix](#) we explain how a convolution with a filter size equal to 3 helps identify patterns in the 1D sequence. Cascading multiple layers with small filters allows us to learn longer patterns.

Furthermore, since the anomaly can appear anywhere along the curve, `MaxPooling1D` introduced in [Chapter 6](#) may help to reduce the sensitivity to the exact location.

Finally we will need to include a few non-linear activations, a `Flatten` layer (seen in [Chapter 6](#)), and one or more fully connected layers. Let's do it!

First, let's import the layers from the `keras` package:

```
from keras.layers import Conv1D, MaxPool1D, Flatten, Activation
```

Next, let's clear out the backend memory, just in case:

```
K.clear_session()
```

Next, let's build the model with our layers, considering again the 200 points as input:

```
model = Sequential()
model.add(Conv1D(16, 3, input_shape=(200, 1)))
model.add(Conv1D(16, 3))
model.add(MaxPool1D())
model.add(Activation('relu'))

model.add(Conv1D(16, 3))
model.add(MaxPool1D())
model.add(Activation('relu'))

model.add(Flatten())

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

```
model.add(Dense(1, activation='sigmoid'))  
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Conv1D requires the input data to have shape (N_samples , N_timesteps , N_features) so we need to add a dummy dimension to our data.

```
X_train_t = X_train[:, :, None]  
X_test_t = X_test[:, :, None]
```

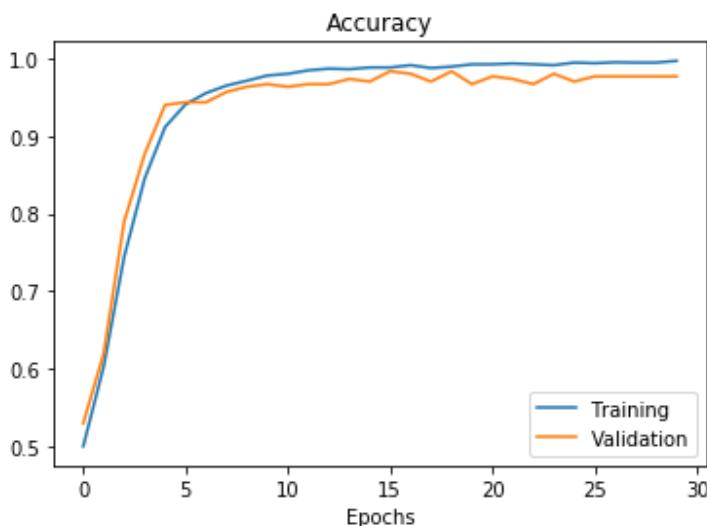
Let's train our model over 30 epochs:

```
h = model.fit(X_train_t, y_train, epochs=30, verbose=0, validation_split=0.1)
```

Now let's plot the accuracy of our model using our 1D convolutional neural network:

```
plt.plot(h.history['acc'])  
plt.plot(h.history['val_acc'])  
plt.legend(['Training', 'Validation'])  
plt.title('Accuracy')  
plt.xlabel('Epochs')
```

```
Text(0.5, 0, 'Epochs')
```



```
print("Test Accuracy: {:.3f}".format(model.evaluate(X_test_t, y_test, verbose=0)[1]))
```

```
Test Accuracy: 0.978
```

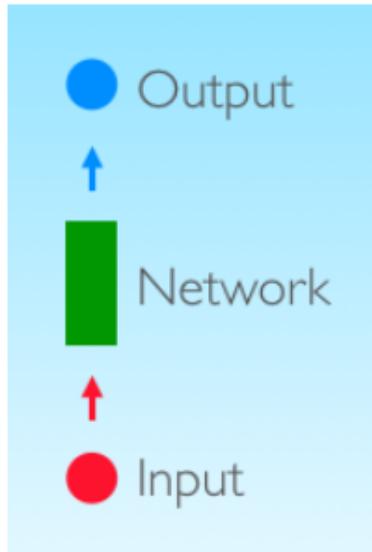
This model is the best so far, and it required no feature engineering. We just reasoned about the type of patterns we were looking for and chose the most appropriate Neural Network architecture to detect them. This is very powerful!

SEQUENCE PROBLEMS

Time series problems can be extended to consider general problems involving sequences. In other words, we can consider a time series as a particular type of sequence, where every element of the sequence is associated with a time. But, in general, we may have sequences of elements not associated with a specific time: for example, a word can be thought as a sequence of characters, or a sentence as a sequence of words. Similarly, the interactions of a user in an app form a sequence of events, and it is a very common use case to try to classify such a sequence or to predict what the next event is going to be.

More generally, we are going to introduce here a few general scenarios involving sequences, that will stretch our application of machine learning to new problems.

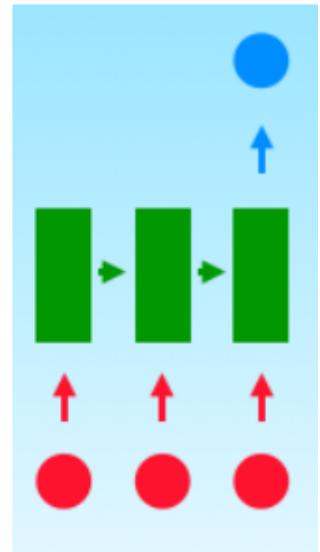
Let's start with 1-to-1 problems



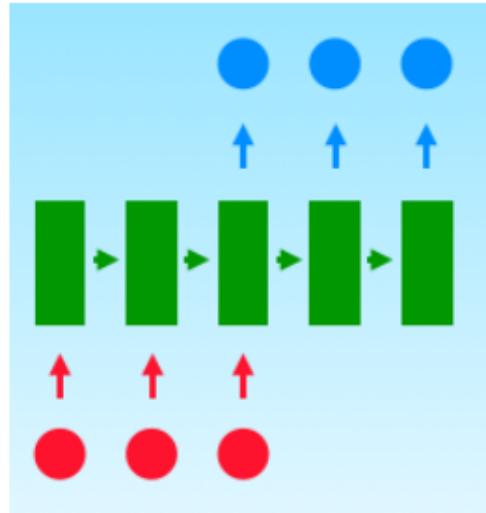
1 to 1



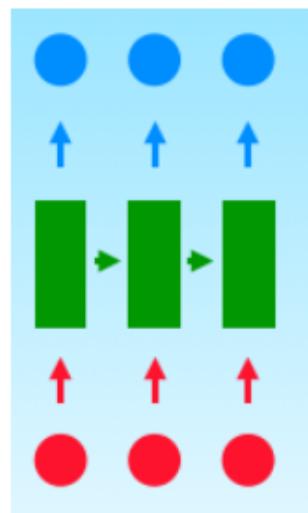
1 to many



Many to 1



many to many
asynchronous



many to many
synchronous

1-to-1

The simplest Machine Learning problem involving a sequence is the **1-to-1 problem**. All the Machine Learning problems we have encountered so far are of this type: linear regression, classification, and convolutional Neural Networks for image or sequence classification. **For each input we have one label**, for each image in MNIST we have a digit label, for each user we have a purchase, for each one banknote we have a label of real or fake.

In all of these cases the Machine Learning model learns a stateless function to connect a given input to a given output.

In the case of sequences, we can expand our framework to allow for the model to make use of past values of the input and of the output. Let's see how.

1-to-many

The **1-to-many** problem starts like the 1-to-1 problem. We have an input and the model generates an output. After the first output is generated, it is fed back to the network as a new input, and the network generates a new output. We can continue like this indefinitely and therefore generate an arbitrary sequence of outputs.

A typical example of this situation is **image captioning**: a single image in input generates as output a text description of arbitrary length.

TIP: a text description can be thought as a sequence either of words or characters. Every single words or characters is indeed an element of the sequence.

many-to-1

The **many-to-1** problem reverses the situation. We feed multiple inputs to the network and at each step we also feed the network output back into the network, until we reach the end of the input sequence. At this point we look at the network output.

Text **sentiment analysis** falls in this category. We associate a single output sentiment label (positive or negative) to a string of text of arbitrary length in input.

asynchronous many-to-many

In the **asynchronous many-to-many** case, we have a sequence in input and a sequence in output. The model first learns to encode an input sequence of arbitrary length into the internal state. Then, when the sequence ends, the model starts to generate a new sequence.

The typical application for this setup is **language translation**, where an input sentence in a language, for example english, is translated to an output sentence in a different language, for example italian. In order to complete the task correctly the model has to "listen" to the whole input sentence first. Once the sentence is finished, the model goes ahead and translates that into the new sentence.

synchronous many-to-many

Finally, there's the **synchronous many-to-many** case, where the network outputs a value at each input, considering both the input and its previous state. **Video frame classification** falls in this category because for each frame we produce a label using the information from the frame but also the information from the state of the network.

RNN allow graphs with cycles

Recurrent Neural Networks can deal with all these sequence problems because their connections form a directed cycle. In other words they are able to retain state from one iteration to the next by using their own output as input for the next step. This is similar to **infinite response filters** in signal processing.

In programming terms, this is like running a fixed program with certain inputs and some internal variables. Viewed this way, RNNs can be thought as networks that learn generic programs.

In fact, RNNs are **Turing-Complete**, which means they can simulate arbitrary programs! We can think of feed-forward Neural Networks as approximating arbitrary functions and recurrent Neural Networks as approximating arbitrary programs. This makes them really really powerful.

TIME SERIES FORECASTING

We have seen how some classification problems involving time series can be solved with the use of convolutional Neural Networks.

The previous dataset however was quite special for a number of reasons. First of all, each sample sequence in the dataset had exactly the same duration, each curve included exactly 200 time steps. Secondly, we had no information about the order of the samples and so we considered them as independent measurements and performed train/test split in the usual way.

Both these conditions are not generally present when dealing with forecasting problems on time series or text data. In fact, a time series can have arbitrary length and it usually comes with a timestamp, indicating the absolute time of each sample.

Let's load a new dataset and let's see how recurrent networks can help in this case.

First of all we are going to load the dataset:

```
df = pd.read_csv('../data/ZonalDemands_2003-2016.csv.bz2',
                 compression='bz2',
                 engine='python')
```

```
df.head(3)
```

	Date	Hour	Total Ontario	Northwest	Northeast	Ottawa	East	Toronto	Essa
0	01-May-03	1	13702	809	1284	965	765	4422	622
1	01-May-03	2	13578	825	1283	923	752	4340	602

	Date	Hour	Total Ontario	Northwest	Northeast	Ottawa	East	Toronto	Essa
2	01-May-03	3	13411	834	1277	910	751	4281	591

```
df.tail(3)
```

	Date	Hour	Total Ontario	Northwest	Northeast	Ottawa	East	Toronto
119853	2016/12/31	22	15195	495	1476	1051	1203	5665
119854	2016/12/31	23	14758	495	1476	1051	1203	5665
119855	2016/12/31	24	14153	495	1476	1051	1203	5665

The dataset contains hourly electricity demands for different parts of Canada and it runs from May 2003 to December 2016. Let's create a `pd.DatetimeIndex` using the `Date` and `Hour` columns.

```
def combine_date_hour(row):
    date = pd.to_datetime(row['Date'])
    hour = pd.Timedelta("%d hours" % row['Hour'])
    return date + hour
```

Let's run this function over our data to generate the `DatetimeIndex` for each column

```
idx = df.apply(combine_date_hour, axis=1)
```

```
idx.head()
```

```
0    2003-05-01 01:00:00
1    2003-05-01 02:00:00
2    2003-05-01 03:00:00
```

```
3    2003-05-01 04:00:00
4    2003-05-01 05:00:00
dtype: datetime64[ns]
...
dtype: datetime64[ns]
...
dtype: datetime64[ns]
```

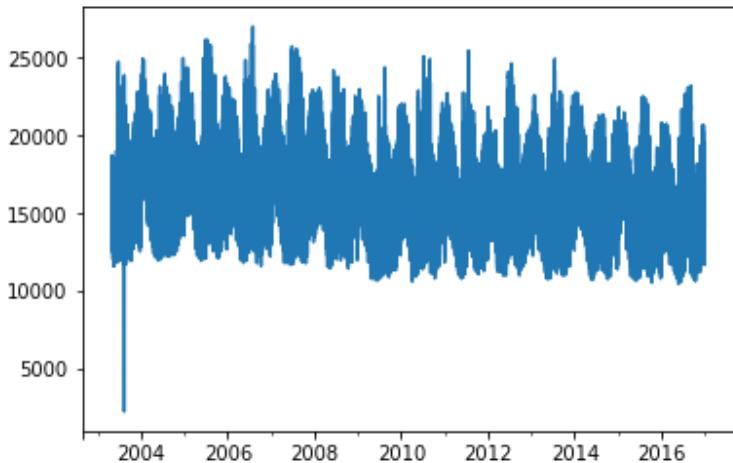
```
df = df.set_index(idx)
```

TIP: the function `set_index()` returns a new DataFrame whose index (row labels) has been set to the values of one or more existing column. Unless you use the `inplace=True` argument this does not alter the DataFrame, it simply returns a different version. That's why we overwrite the original `df` variable.

Now that we have set the index, let's select and plot the `Total Ontario` column:

```
df['Total Ontario'].plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc7162f6240>
```



Great! The time series seems quite regular! This looks promising for forecasting. Let's split the data in time on January 1st, 2014. We will use data before that date as training data and data after that date as test.

```
split_date = pd.Timestamp('01-01-2014')
```

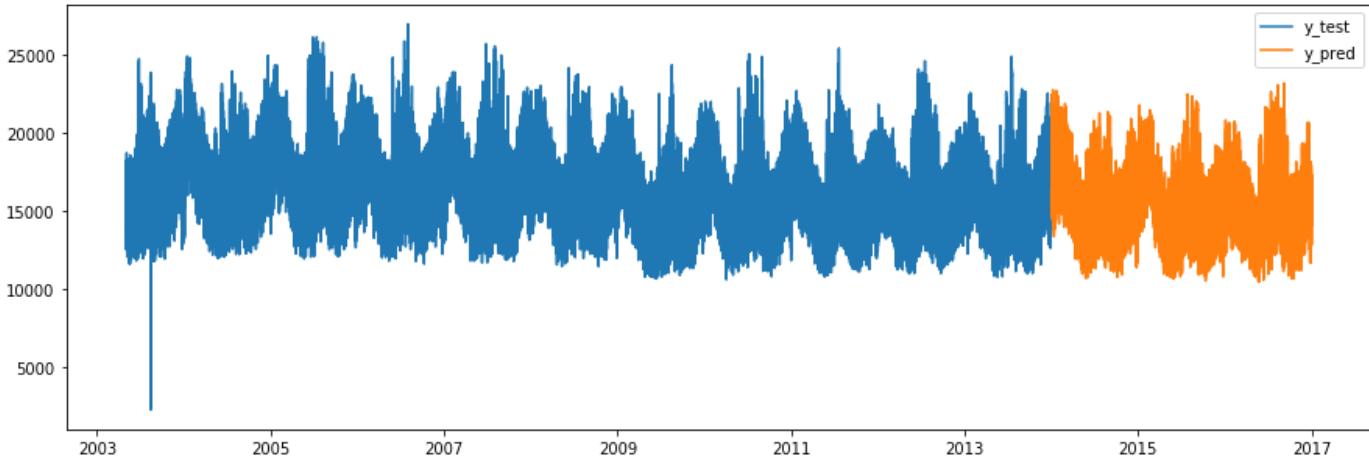
Now we copy the data to a pair of new Pandas data frames that only contain the Total Ontario data up to the split date (train) and after the split date (test).

```
train = df.loc[:split_date, ['Total Ontario']].copy()
test = df.loc[split_date:, ['Total Ontario']].copy()
```

TIP: We use the `.copy()` command here because the `.loc` indexing command may return a view on the data instead of a copy. This could be a problem later on when we do other selections or manipulations of the data.

Let's plot the data. We will use the matplotlib plotting function that is automatically aware of index with dates and times and assign a label to each plot so that we can display them with a legend:

```
plt.figure(figsize=(15,5))
plt.plot(train, label='y_test')
plt.plot(test, label='y_pred')
plt.legend()
plt.show()
```



We've already seen in Chapter 3 that Neural Network models are quite sensitive to the absolute size of the input features. This means that passing in features with very large or very small values will not help our model converge to a solution. Hence, we should rescale the data before anything else.

Notice that there's a huge drop somewhere in 2003. We shouldn't use that as the minimum for our analysis, since it is clearly an outlier.

We will rescale the data in such a way that most of it is close to 1. We can achieve this by subtracting 10000, which shifts everything down and then dividing by 5000.

TIP: feel free to adjust these values as you prefer, or to try out other scaling methods like the `MinMaxScaler` or the `StandardScaler`. The important thing is to get our data close to 1 in size, not exactly between 0 and 1.

```
offset = 10000
scale = 5000

train_sc = (train - offset) / scale
test_sc = (test - offset) / scale
```

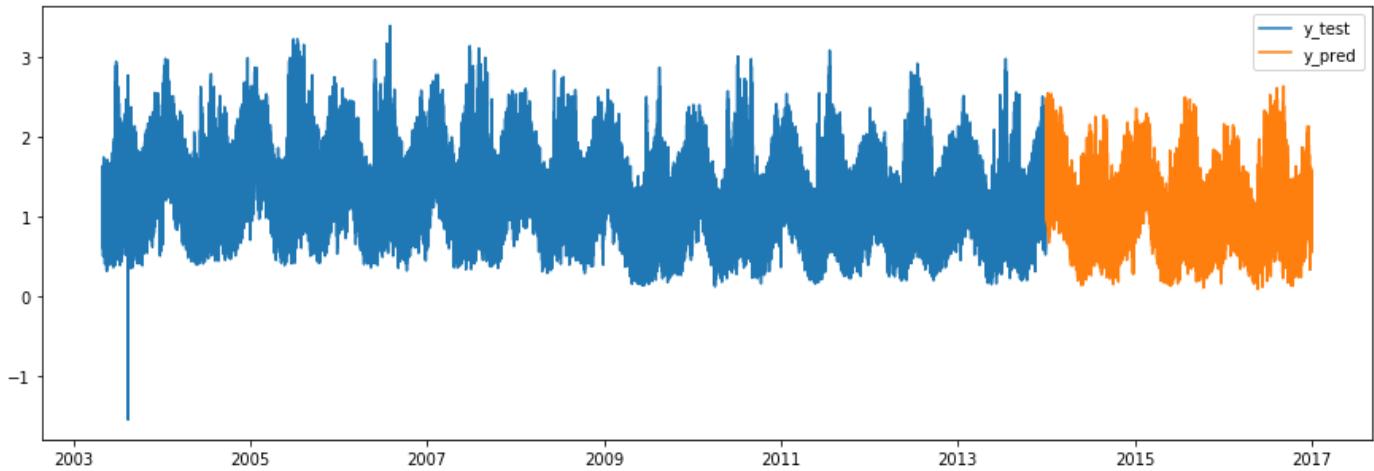
Let's look at the first four dates and demand just to make sure our data is in the expected region of where we think it should be.

```
train_sc[:4]
```

	Total Ontario
2003-05-01 01:00:00	0.7404
2003-05-01 02:00:00	0.7156
2003-05-01 03:00:00	0.6822
2003-05-01 04:00:00	0.7002

Let's plot our entire dataset to confirm it matches our expectation.

```
plt.figure(figsize=(15,5))
plt.plot(train_sc, label='y_test')
plt.plot(test_sc, label='y_pred')
plt.legend()
plt.show()
```



We are finally ready to build a predictive model. Our target is going to be the value of the demand on a certain time, and to start we will use the demand on the previous time as the only feature.

```
X_train = train_sc[:-1].values
y_train = train_sc[1:].values

X_test = test_sc[:-1].values
y_test = test_sc[1:].values
```

Now we have our training data as well as testing data mapped out. Let's move on to model building.

Fully connected network

Let's train a fully connected network to predict and see that it is not able to predict the next value from the previous one.

The network will have single input (the previous hour value) and a single output.

We can see this as a simple *regression problem*, since we want to establish a connection between two continuous variables.

TIP: if you need a refresher on what a regression is and why it makes sense to use it here, have a look at [Chapter 3](#) where we used a Linear regression to predict the weight of individuals given their height.

Since we want to predict a continuous variable, the output of the network does not need an activation function and we will use the `mean_squared_error` as loss function, which is a standard error metric in regression models.

Let's clear the backend of any held memory first, as we have done many times when building a new model:

```
K.clear_session()
```

Next, let's build our model:

```
model = Sequential()
model.add(Dense(24, input_dim=1, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(6, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_1 (Dense)	(None, 24)	48
dense_2 (Dense)	(None, 12)	300
dense_3 (Dense)	(None, 6)	78
<hr/> ... <hr/>		

In this case, before fitting the built Neural Networks, we load the `EarlyStopping` callback, to halt the training if it is not improving.

TIP: a `callback` is a set of functions to be applied at each epoch during the training. We have already encountered them in [Exercise 4 of Chapter 5](#). You can pass a list of callbacks to the `.fit()` method, and in this specific case we use the `EarlyStopping` callback to stop the training if no progress is observed. According to the [documentation](#), `monitor` defines the quantity to be monitored (the `mean_squared_error` in this case) and `patience` defines the number of epochs with no improvement after which the training will be stopped.

In particular we will set the `EarlyStopping` callback to monitor the value of the loss and stop the training loop with a `patience=1` if that does not improve. Without this callback, the training will be stuck on a fixed loss without improving, and the training will not stop by itself (go ahead and try to confirm that!).

```
from keras.callbacks import EarlyStopping  
early_stop = EarlyStopping(monitor='loss', patience=1, verbose=1)
```

Now we can launch the training, using this callback to monitor the progress of the data.

Our dataset has over 100k points so we can choose large batches.

```
model.fit(X_train, y_train, epochs=200,  
          batch_size=512, verbose=1,  
          callbacks=[early_stop])
```

```
Epoch 1/200  
93551/93551 [=====] - 1s 9us/step - loss: 0.4497  
Epoch 2/200  
93551/93551 [=====] - 1s 6us/step - loss: 0.0210  
Epoch 3/200  
93551/93551 [=====] - 1s 6us/step - loss: 0.0178  
Epoch 4/200  
93551/93551 [=====] - 1s 6us/step - loss: 0.0177  
Epoch 5/200  
...  
Epoch 5/200
```

```
<keras.callbacks.History at 0x7fc6b3f676a0>
```

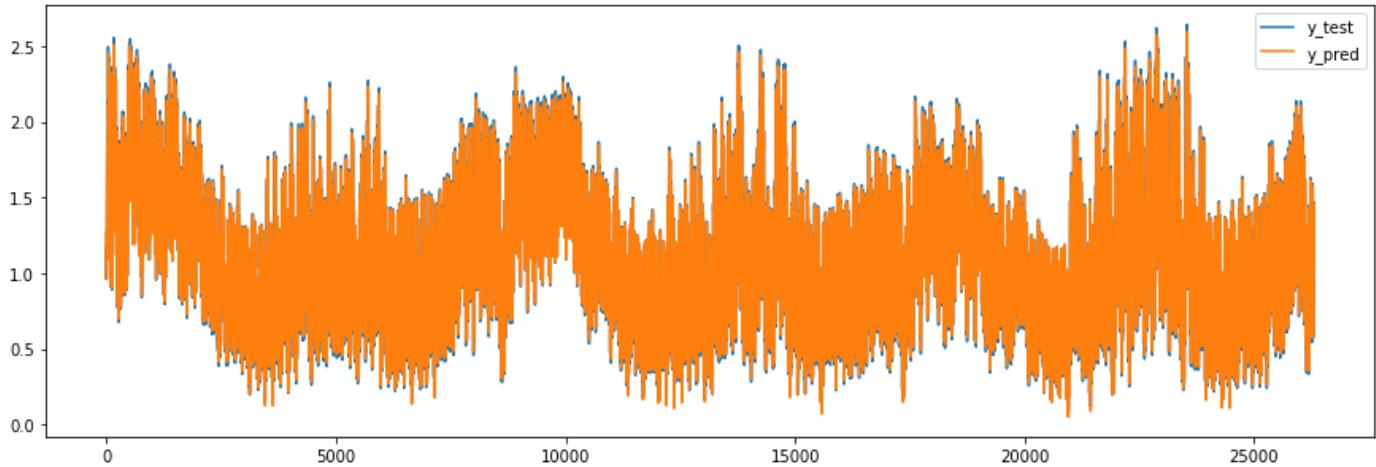
The model stopped improving quite quickly. Feel free to experiment with other architectures and other activation functions. Let's see how our model is doing. We can generate the predictions on the test set by running `model.predict`.

```
y_pred = model.predict(X_test)
```

Let's visually compare test values and predictions:

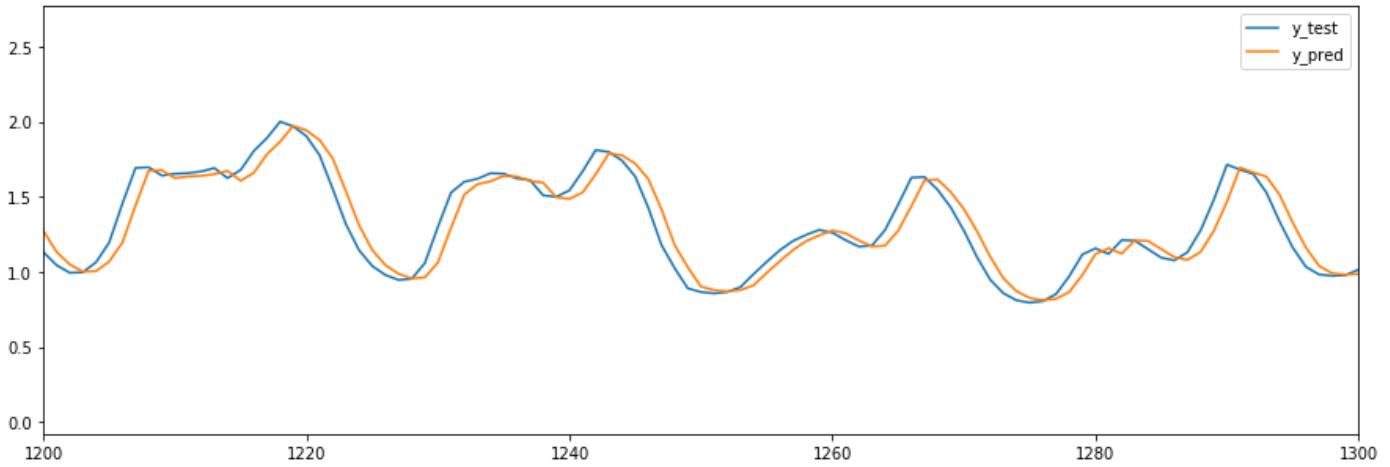
```
plt.figure(figsize=(15,5))  
plt.plot(y_test, label='y_test')  
plt.plot(y_pred, label='y_pred')
```

```
plt.legend()  
plt.show()
```



They seem to overlap pretty well. Is it so? Let's zoom in and watch more closely. We will do this by using the `plt.xlim` function that sets the boundaries of the horizontal axis in a plot. Feel free to choose other values in order to inspect other regions of the plot. Also notice that we have lost the date labels when we created the data, but this is not a problem: we can always bring them back from the original series if we need them.

```
plt.figure(figsize=(15,5))  
plt.plot(y_test, label='y_test')  
plt.plot(y_pred, label='y_pred')  
plt.legend()  
plt.xlim(1200,1300)  
plt.show()
```



Fully connected network evaluation

Is this a good model? At a first glance we may be tempted to say it is.

Let's measure the total *mean squared error* (a.k.a. our total loss) and the R^2 score on the test set. As seen in Chapter 3 here), if the R^2 score is far from 1.0, that is a sign of a bad regression.

TIP: If you need a refresher about Mean Squared Error and R^2 score, how they are defined and used, take a look at Chapter 3 here) and Chapter 3 here

```
from sklearn.metrics import mean_squared_error, r2_score
```

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("MSE: {:.3f}".format(mse))
print("R2: {:.3f}".format(r2))
```

```
MSE: 0.0149
R2: 0.933
```

In this case however the R^2 score is quite high, which would lead us to think the model is quite good.

However, the model is actually not that good! Why?

If you inspect the graph closely, you will realize that the network has just learned to *repeat the same value* it receives in input!

This is not forecasting at all, in other words the model has no real predictive power. It behaves like a parrot that repeats yesterday's value for today. In this particular case, since the curve is varying smoothly, the differences between one day and the next are small and the R^2 score is still pretty close to 1. However, the model is not anticipating any future value and so it would be quite useless for forecasting.

This behavior is not surprising. After all, the only input feature our model knew was the value of the time series in the previous period, so it makes sense that the best it could do was to learn to repeat such value as prediction for what would come next.

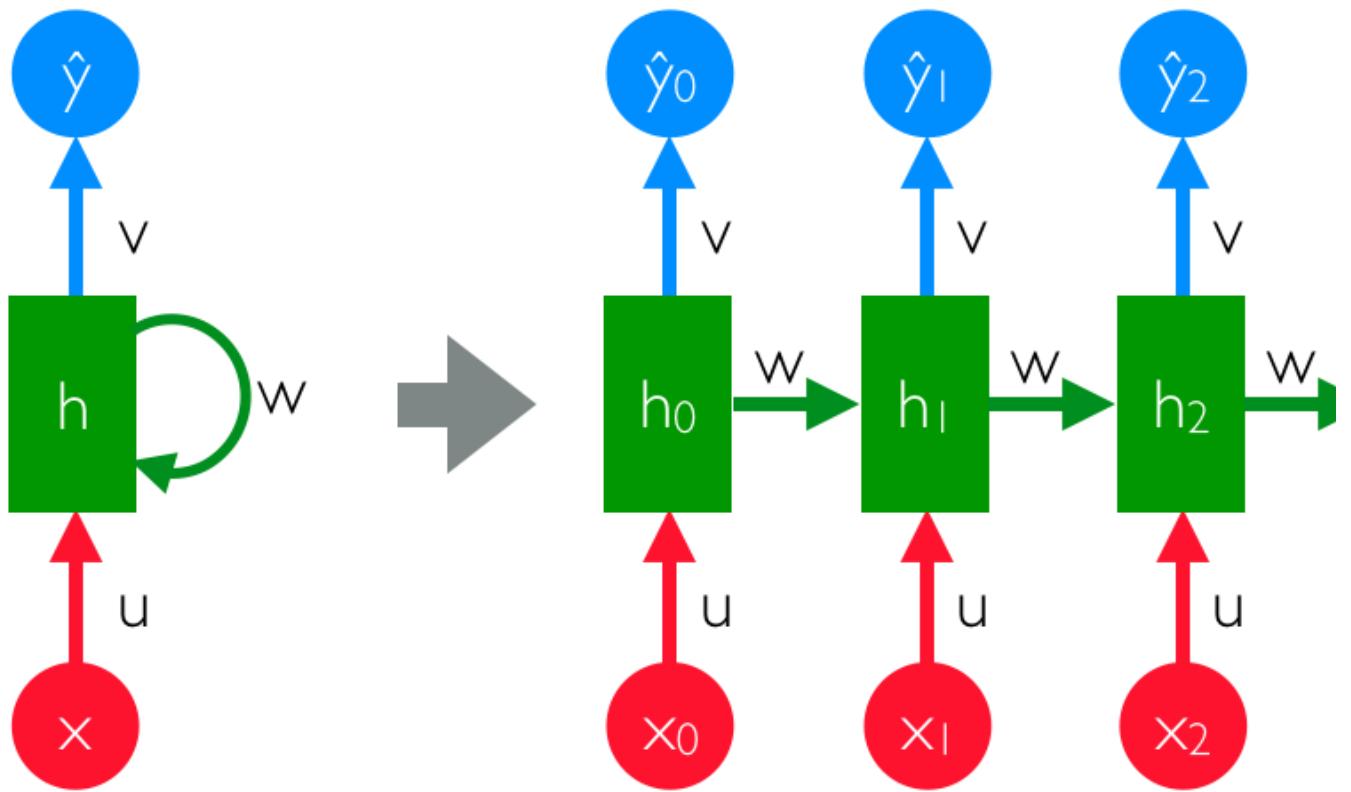
Let's see if a recurrent network improves the situation.

Recurrent Neural Networks

Vanilla RNN

As we introduced, Recurrent Neural Networks are able to maintain an internal state using feedback loops. Let's see how we could build a simple RNN.

The *Vanilla* Recurrent Neural Network can be built as a fully connected Neural Network if we unroll the time axis.



Ignoring the output of the network for the time being, let's focus on the recurrent aspect. The network is recurrent because it's internal state h at time t is obtained by mixing current input x_t with the previous value of the internal state h_{t-1} :

$$h_t = \tanh(w h_{t-1} + u x_t)$$

At each instant of time, the simple RNN is behaving as a fully connected network with two inputs: the current input x_t and the previous output h_{t-1} .

TIP: Notice that for now we are using a network with a single input and a single output, so both x and h are numbers. Later we will extend the notation to networks with multiple input and multiple recurrent units in a layer. As you will see the extension is quite simple.

Notice only two weights are involved: the weight multiplying the previous value of the output w and the weight multiplying the current input u . By the way doesn't this formula remind you of the [Exponentially Weighted Moving Average \(or EWMA\)](#)?

TIP: we have already mentioned EWMA in [Chapter 5](#) and it is explained in the appendix. Just as a reminder, it's a simple smoothing algorithm that follows the formula:

$$y_t = (1 - \alpha) y_{t-1} + \alpha x_t$$

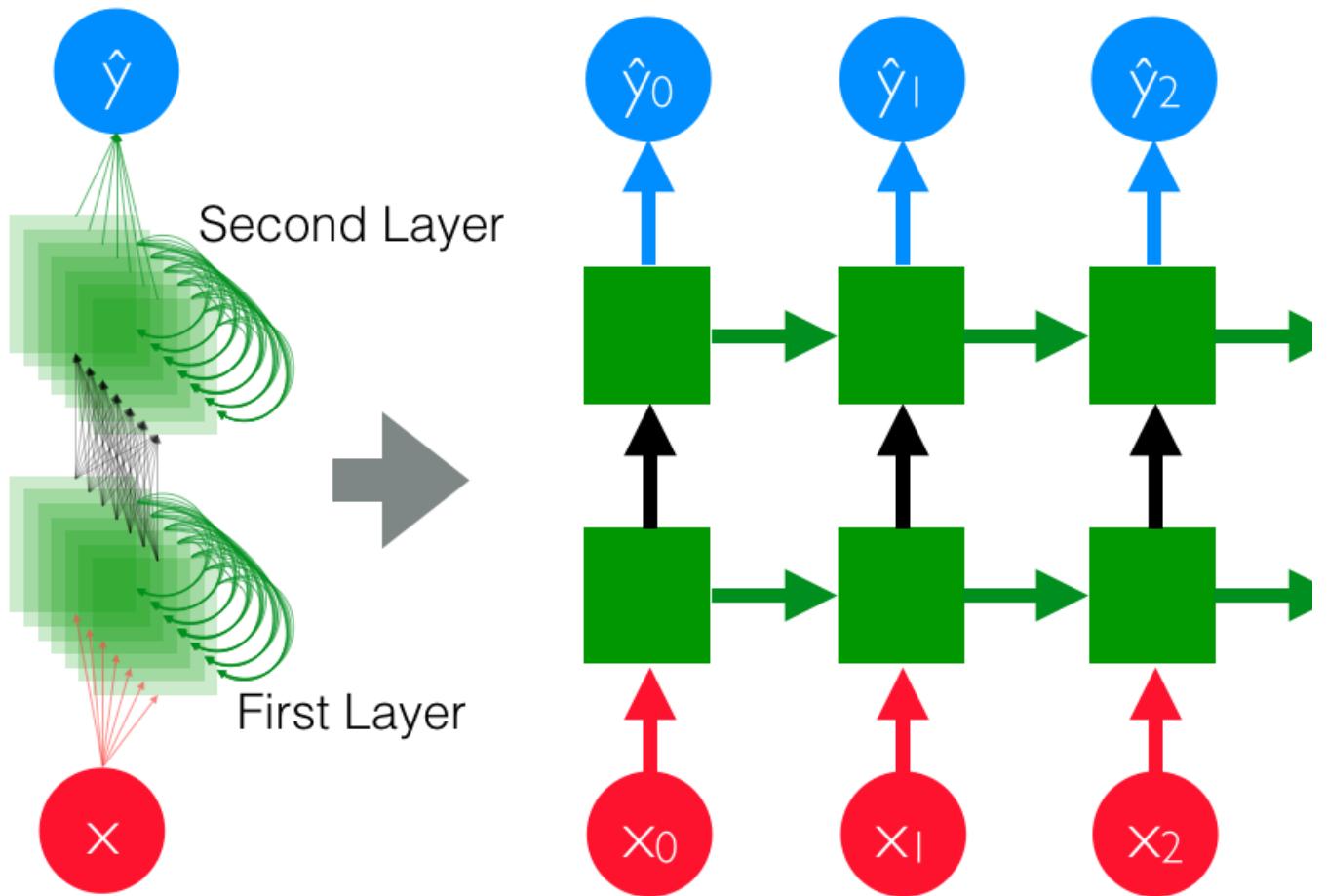
EWMA smooths a signal given by a sequence of data reducing its fluctuations.

It is not exactly the same, because there is a *tanh* and the two weights are independent but it does look similar: it's a linear mixing of the past output with the present input, followed by a non-linear activation function.

Also notice that the **weights do not depend on time**. The network is learning the best values of its two weights which are fixed in time.

Deep Vanilla RNN

We can build deep recurrent Neural Networks by stacking recurrent layers onto one another. We feed the input to a first layer and then feed the output of that layer into a second layer and so on. Also we can add multiple recurrent units in each layer. Each unit is receiving inputs from all the units in the previous layer (or the input) as well as all the units in the same layer at the previous time:



If we have multiple layers we will need to make sure that an earlier layer returns the whole sequence of outputs to the next layer. This is achieved in Keras using the `return_sequences=True` optional argument when defining a layer. We will see an example of this in [Exercise 1](#).

Keras implements Vanilla Recurrent Layers with the `layers.SimpleRNN` class. Let's try it out on our forecasting problem. First of all we import it.

```
from keras.layers import SimpleRNN
```

The [documentation](#) for recurrent layers reads:

Input shape

3D tensor with shape (batch_size, timesteps, input_dim).

but so far we have used only a tensor of order two for our data. Let's stop for a second and think about how to reshape our data, because there's more than one way. Our input data right now has a shape of:

```
x_train.shape
```

```
(93551, 1)
```

So it's like a matrix with a single column. We want to add an additional dimension to the tensor so that the data is a tensor of order three. There are many ways of doing this, but a very simple one is to simply add a `None` axis like this:

```
x_train_t = X_train[:, None]  
X_test_t = X_test[:, None]  
  
y_train_t = y_train[:]  
y_test_t = y_test[:]
```

Let's check the shape of our new variable `x_train_t`:

```
x_train_t.shape
```

```
(93551, 1, 1)
```

Good! We reshaped the data to have one additional axis as requested. Now let's think about the batches. If we randomly sample this data and give the model batches of 1 point in input with the corresponding label in output the model will not leverage the fact that the data is part of a sequence and therefore produce results that are very similar to the ones of the Fully Connected network.

Instead, we'd like to leverage the fact that all the data comes in a sequence. This can be done by feeding the data sequentially to the network. In other words we don't want to randomly sample batches from the sequence, we want to feed the data one by one sequentially, while maintaining the state of the network between one point and the next.

This can be achieved by setting the `stateful=True` argument in the layer, but it requires that the size of our data is exactly a multiple of the batch size.

Since we want to feed the points one by one we will choose a `batch_size=1`. Let's do it!

Now let's create a `SimpleRNN` with one layer with 6 nodes. This means that there are 6 recurrent units in our layer. The principle is the same as above, only each of these units will receive a 6 dimensional vector as recurrent input from the past, together with the single value of the actual input.

TIP: The number of nodes here is arbitrary. We could choose to put many more nodes, but that would result in a bigger model which is slower to train. We have noticed that with 6 nodes results are already acceptable and hence we choose that value.

Notice that since we are using the `stateful=True` flag we will need to pass the `batch_input_shape` to the first layer.

We will use the `Adam` optimizer (which is one of the most efficient and robust optimizer, as seen in [Chapter 5](#), adopting a small value for the learning rate, since the `SimpleRNN` can sometimes be unstable.

```
from keras.optimizers import Adam, RMSprop
```

Let's clear the backend memory again:

```
K.clear_session()
```

Now let's build the model. We will pass a `batch_input_shape=(1, 1, 1)` because we read the data one point at a time. Also we will set the input weights to one. We do this in order to reduce the variability in the results obtained, as you'll see, this model is quite unstable.

TIP: if you get a result that is very different from the one of the book, go ahead and re-initialize the model. It may be just a case of bad luck with the starting point in the minimization.

Let's build the model:

```
model = Sequential()
model.add(SimpleRNN(6,
                    batch_input_shape=(1, 1, 1),
                    kernel_initializer='ones',
                    stateful=True))
model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer=Adam(lr=0.0005))
```

Now we can fit the data. Since we are maintaining states between point, we shall pass the data in order using the `shuffle=False` flag and `batch_size=1`. Also, we run the training for a single epoch. In our experiments this should be sufficient to get decent results:

```
model.fit(X_train_t, y_train_t,  
          epochs=1,  
          batch_size=1,  
          verbose=1,  
          shuffle=False)
```

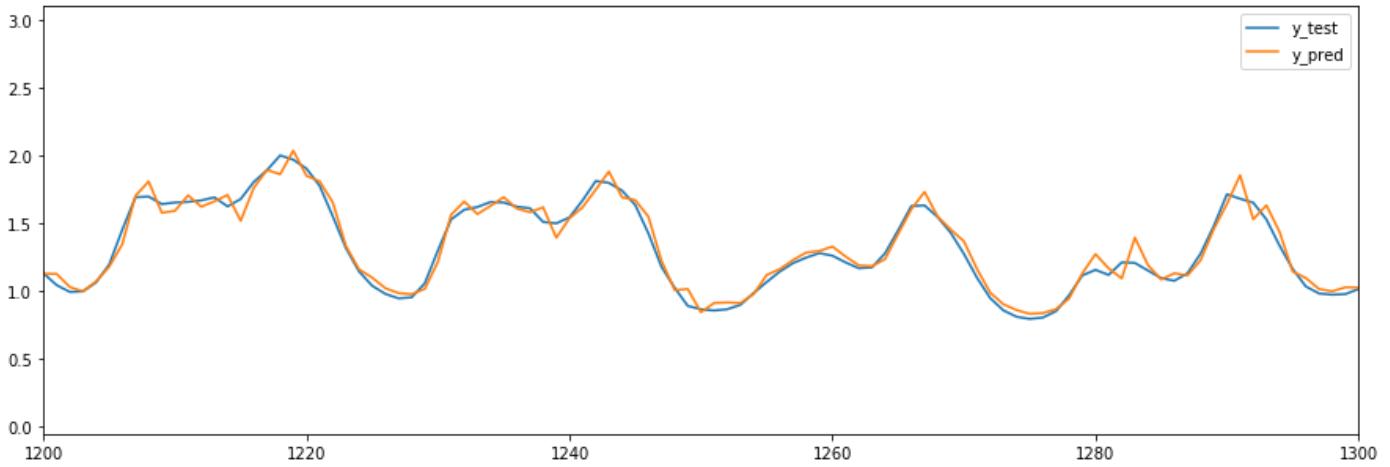
```
Epoch 1/1  
93551/93551 [=====] - 240s 3ms/step - loss: 0.0190
```

```
<keras.callbacks.History at 0x7fc6b3d95518>
```

Let's plot a small part of our predictive model to compare train and test.

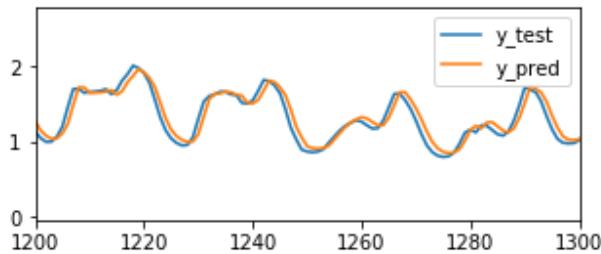
```
y_pred = model.predict(X_test_t, batch_size=1)  
plt.figure(figsize=(15,5))  
plt.plot(y_test_t, label='y_test')  
plt.plot(y_pred, label='y_pred')  
plt.legend()  
plt.xlim(1200, 1300)
```

```
(1200, 1300)
```

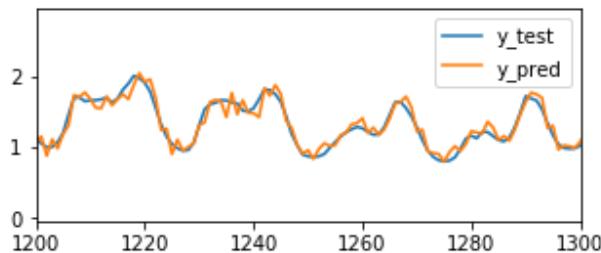


Notice that despite our initialization, the model converges to different solutions at each training run.

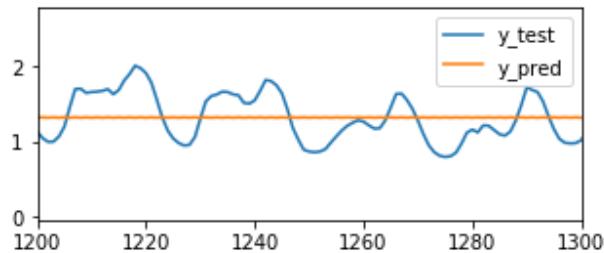
- Sometimes you will get a graph that looks very similar to the Fully Connected result, with no predictivity at all:



- Sometimes you will get a graph that looks noisy while being closer to the actual data in the sharp decays, meaning some forecasting power is actually achieved:



Sometimes the network will get stuck and give nonsense results like this one:



Feel free to change the number of layers, nodes, optimizer and learning rate to see if you can get better results. You will notice that this model is very prone to diverging away from a small value of the loss, which is not ideal at all.

Let's also check *MSE* and the R^2 score:

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

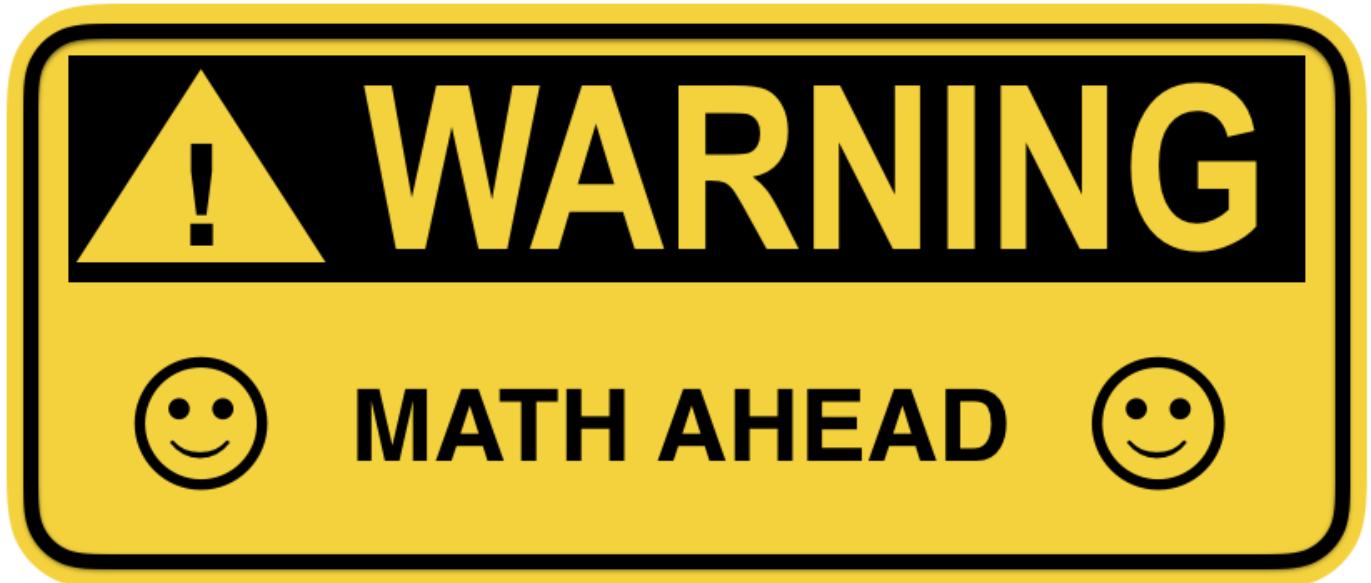
print("MSE: {:.3f}".format(mse))
print("R2: {:.3f}".format(r2))
```

```
MSE: 0.00598
R2: 0.973
```

All in all this model does not seem to be much better than the Fully Connected one and it is also quite unstable. The problem lies with the fact that the `SimpleRNN` actually has a short memory and cannot learn really long-term patterns.

Let's see why this happens and how we can fix it.

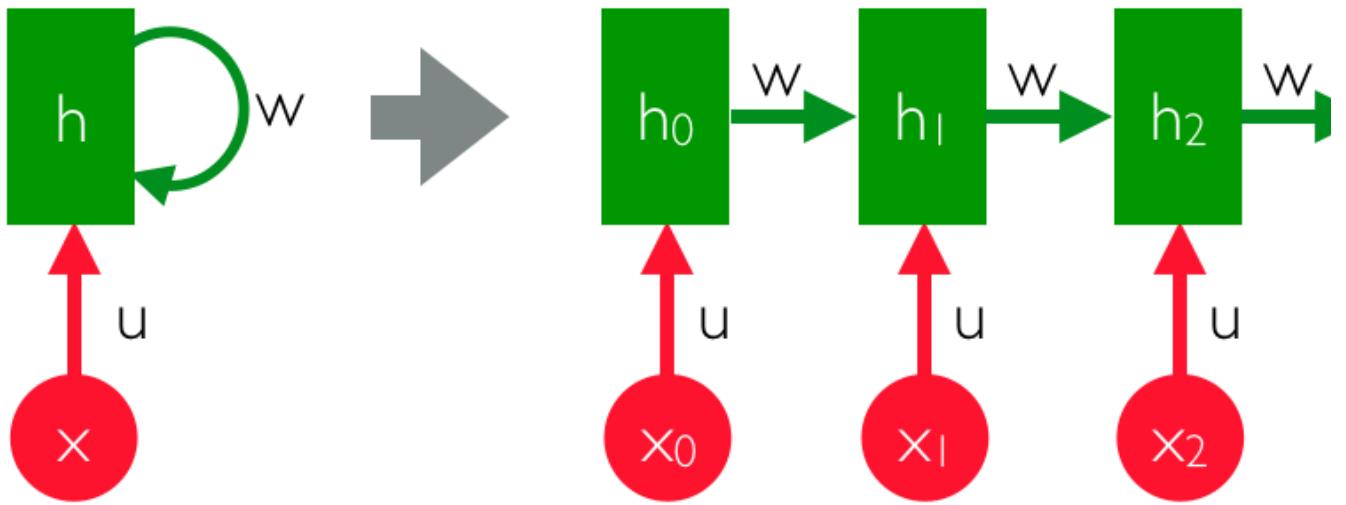
Recurrent Neural Network Maths



In order to fully understand how recurrent networks work and why our simple implementation fails we will need a little bit of maths. Like we suggested in [Chapter 5](#) you can feel free to skip this section entirely if you just want to get to the working model. You can always come back to it later on, if you are curious about how a recurrent network works

Vanishing Gradients

Let's start from the equation of backpropagation through time, and let's ignore the output of the network for now and let's focus on the recurrent part. This is also called an **encoder network**, since it the output is discarded.



This network is encountered in many cases, for example when solving many-to-1 problems like sentiment analysis or asynchronous many-to-many problems like machine translation.

Let's rewrite the recurrent relations, that in this case are:

$$\begin{aligned} z_t &= w h_{t-1} + u x_t \\ h_t &= \phi(z_t) \end{aligned}$$

where we substituted the tanh activation function to a generic activation ϕ .

We can now use the *overline* notation introduced in Chapter 5 to study the backpropagation through time.

If we assume to have already backpropagated from the output all the way back to the error signal \bar{h}_T , we can write the backpropagation relations as:

$$\begin{aligned} \bar{h}_t &= \bar{z}_{t+1} w \\ \bar{z}_t &= \bar{h}_t \phi'(z_t) \end{aligned}$$

Let's focus our attention on \bar{h}_t , and let's propagate back all the way to \bar{h}_0 :

$$\begin{aligned} \bar{h}_0 &= w \bar{z}_1 \\ &= w \bar{h}_1 \phi'(z_1) \\ &= w^2 \bar{h}_2 \phi'(z_1) \phi'(z_2) \\ \dots &= w^T \bar{h}_T \phi'(z_1) \phi'(z_2) \dots \phi'(z_T) \end{aligned}$$

Now remembering the definition of $h = \frac{\partial J}{\partial h}$ we can write:

$$\overline{h_0} = \overline{h_T} \frac{\partial h_T}{\partial h_0}$$

which implies:

$$\frac{\partial h_T}{\partial h_0} = w^T \phi'(z_1) \phi'(z_2) \dots \phi'(z_T)$$

Now let's stop for a second and focus on $\phi'(z)$. For most activation functions (*sigmoid*, *tanh*, *relu*) this quantity is **bounded**. This is easily seen looking at the graph of the derivative of these functions:

First, let's define the `sigmoid` and `relu` functions:

```
x = np.linspace(-10, 10, 1000)

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def relu(x):
    cond = x > 0
    return cond * x
```

Let's plots for the `sigmoid`, the `Tanh`, and the `relu` activation functions along with their derivatives.

```
plt.figure(figsize=(12,8))
plt.subplot(321)
plt.plot(x, sigmoid(x))
plt.title('Sigmoid')

plt.subplot(322)
plt.plot(x[1:], np.diff(sigmoid(x))/np.diff(x))
plt.title('Derivative of Sigmoid')

plt.subplot(323)
plt.plot(x, np.tanh(x))
plt.title('Tanh')

plt.subplot(324)
plt.plot(x[1:], np.diff(np.tanh(x))/np.diff(x))
plt.title('Derivative of Tanh')

plt.subplot(325)
```

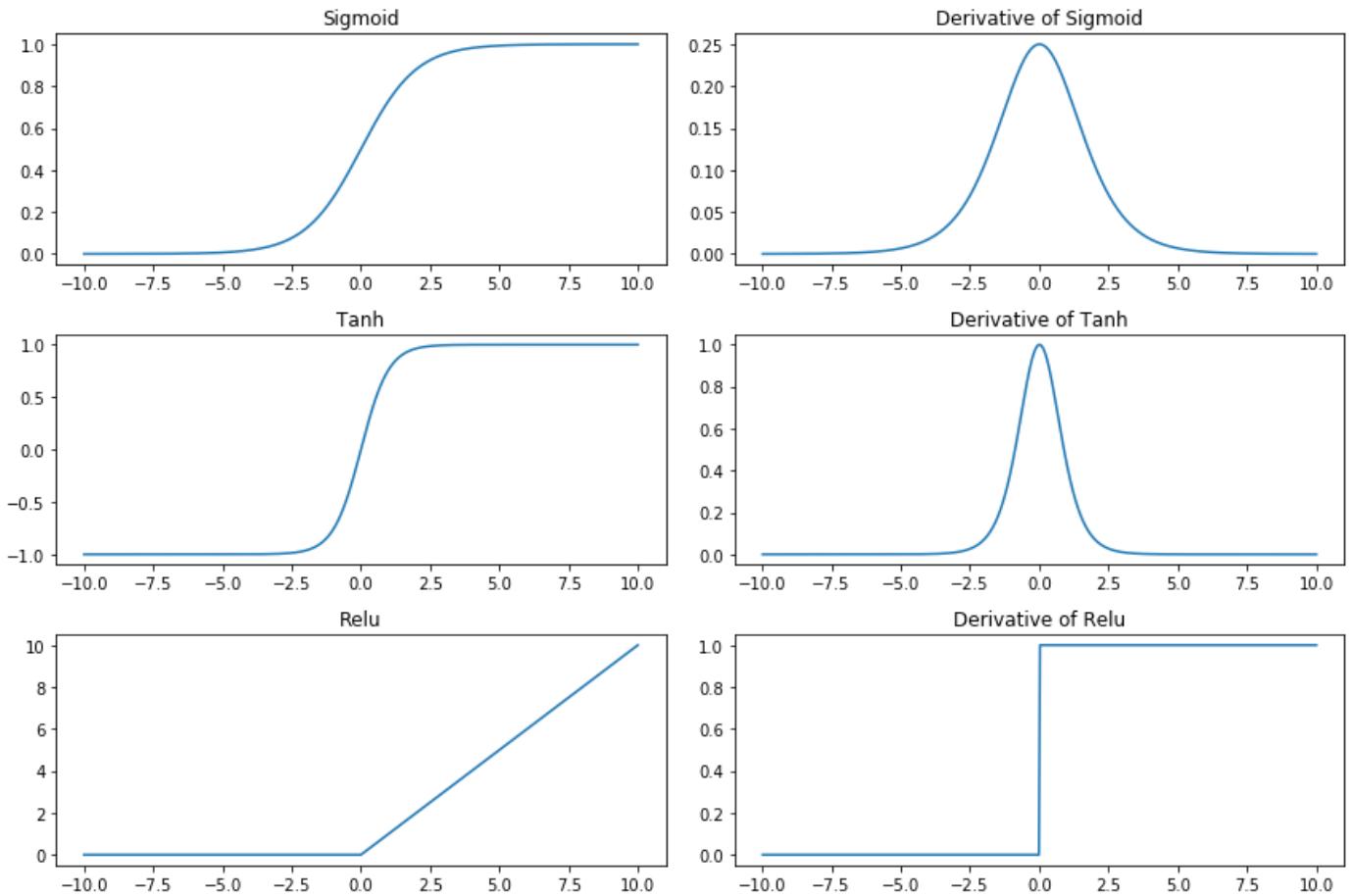
```

plt.plot(x, relu(x))
plt.title('Relu')

plt.subplot(326)
plt.plot(x[1:], np.diff(relu(x))/np.diff(x))
plt.title('Derivative of Relu')

plt.tight_layout()

```



All the derivatives take values between 0 and 1, i.e. they are **bounded**. We can use this fact to rewrite the last equation as:

$$\frac{\partial h_T}{\partial h_0} = w^T \phi'(z_1) \phi'(z_2) \dots \phi'(z_T) \leq w^T$$

Which means that derivative of the last output with respect to the first output is less than or equal to w^T .

At this point the vanishing gradient problem should be evident. If $w < 1$ the propagation through time is suppressed at each additional time step. This means that means the influence of an input point that is 3 steps back in time, will contribute to the gradient with a term smaller than w^3 . If, for example, $w = 0.1$, the previous point will contribute with less than 10%, the one before with less than 1% and the one before with 0.1% and so on. You can see that their contributions quickly disappear.

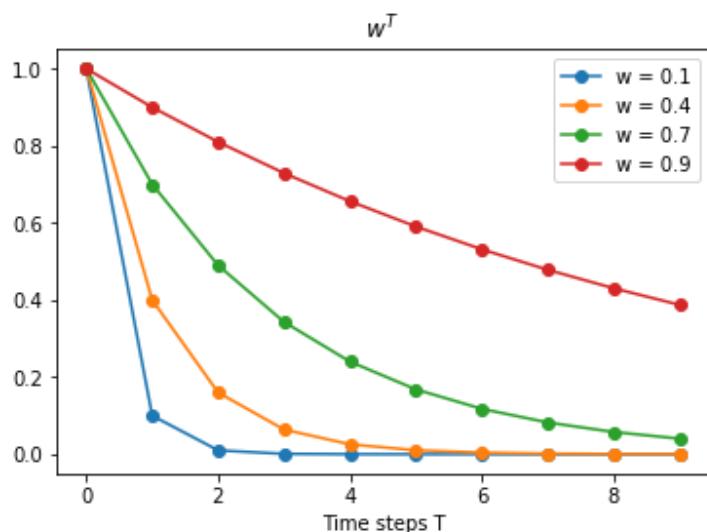
Let's take a peek at what this looks like visually. First, let's create a decay function that we'll use to create our plots.

```
def decay(w, T):
    t = np.arange(T)
    b = w**t
    return t, b
```

Now let's plot the quantity w^T as a function of T for several values of w :

```
ws = [0.1, 0.4, 0.7, 0.9]
for w in ws:
    t, b = decay(w, 10)
    plt.plot(t, b, 'o-')

plt.title("$w^T$")
plt.xlabel("Time steps T")
plt.legend(['w = {}'.format(w) for w in ws])
plt.show()
```



The error signal quickly goes to zero if the recurrent weight is smaller than 1. This suggests that the recurrent model is only able to capture short time dependencies, but longer dependencies are rendered useless.

Similarly, it can be shown that when w is greater than a certain threshold, the gradient will exponentially explode over time (notice that in the gradient we have w^T), rendering the backpropagation unstable.

It would appear as if we are stuck with a model that either does not converge at all or it quickly forgets about the past. How can we solve this?

Long Short-Term Memory Networks (LSTM)

LSTMs were designed to overcome the problems of simple Recurrent networks by allowing the network to store data in a sort of memory that can be accessed at later times. LSTM units are a bit more complicated than the nodes we have seen so far, so let's take our time to understand what this means and how they work.

Again, feel free to skip this section at first and come back to it later on.

We will start from an intuitive description of how LSTM works and we will gradually approach the mathematical formulas. We will do this because the formulas for the LSTM can be daunting at first, so it is good to break them down and learn them gradually.

At the core of the LSTM is the **internal state c_t** . This can be thought of as an internal conveyor belt that carries information from one time step to the next. In the general case, this is a vector, with as many entries as the number of units in the LSTM layer. The LSTM unit will store information in this vector and this information will be available for retrieval later on.



At time t the LSTM block receives 2 inputs:

- the new data at time t , which we indicate as \mathbf{x}_t
- the previous output at time $t - 1$ which we indicate as \mathbf{h}_{t-1} .

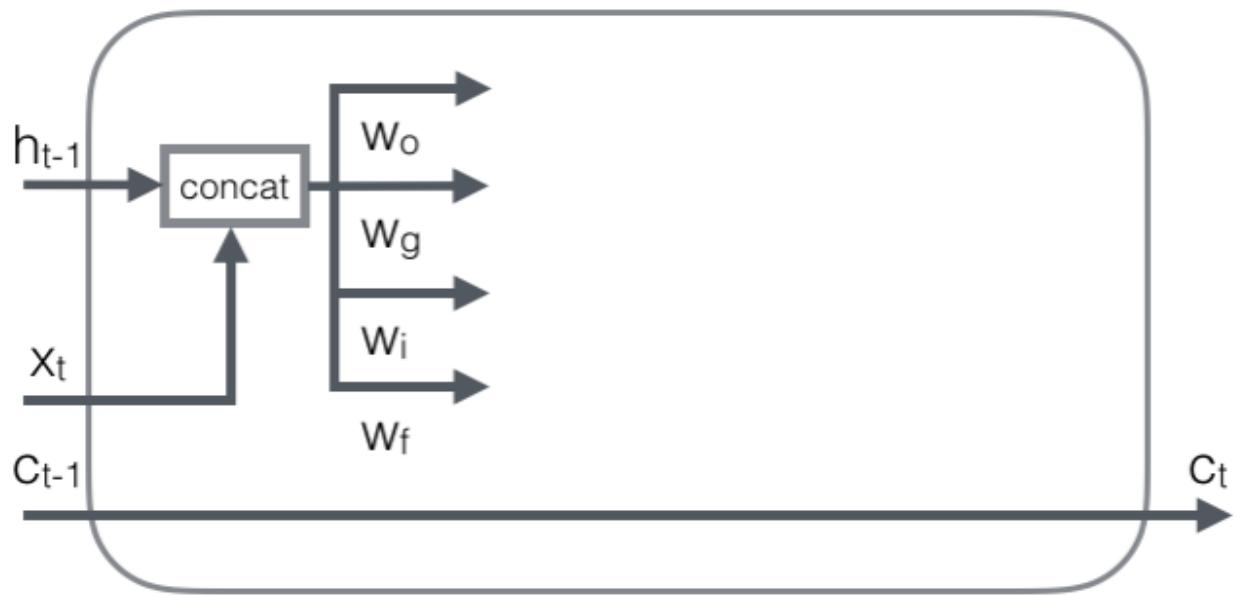
These two inputs are concatenated, in order to create a unique set of input feature.

For example, if the input vector has length 3 (i.e. there are 3 input features) and the output vector has length 2 (i.e. there are 2 output features), the concatenated vector has now 5 features, 3 coming from the input vector and 2 coming from the output vector.



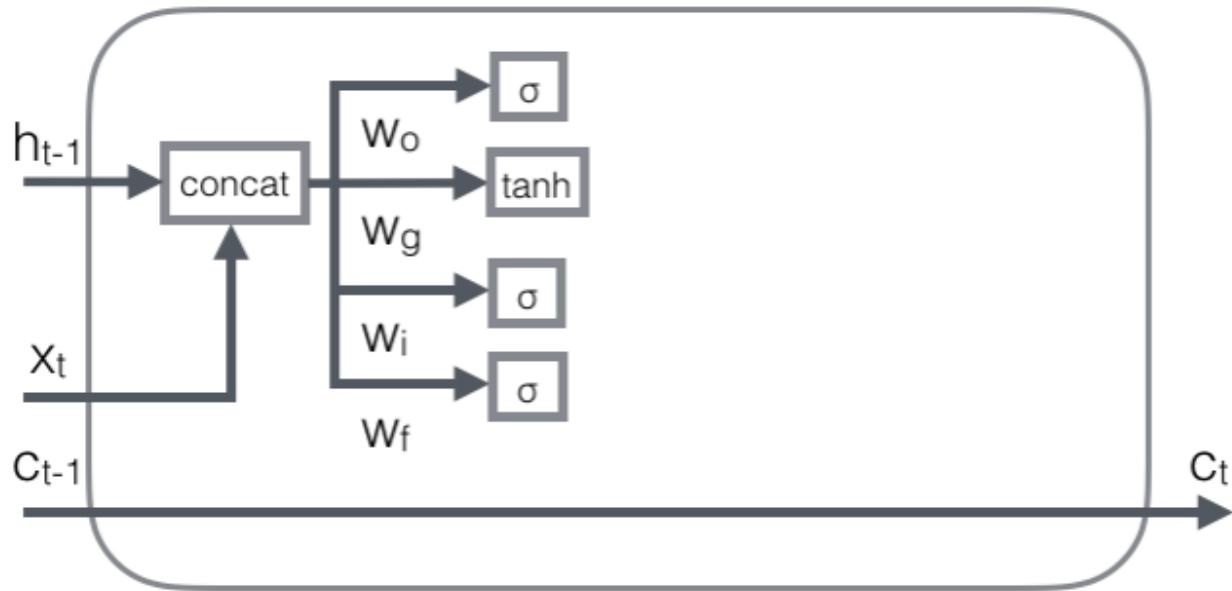
The next step is to apply 4 different simple Neural Network layers to these concatenated features along 4 parallel branches. Each branch takes a copy of the features and multiplies them by an independent set of weights and a different activation function.

TIP: You may be wondering why 4 and not 3 or 5. The reason is simple: one branch is the one that will process the data similarly to the `Vanilla RNN`, i.e. it will take the past and the present, weight them and send them through a `tanh` activation function. The other three branches will control operations that we call **gates**. As you will see, these gates control how past and present information are recorded in the internal state. Other kinds of recurrent units, like GRU, use a different number of gates, so 4 is specific to the LSTM architecture.



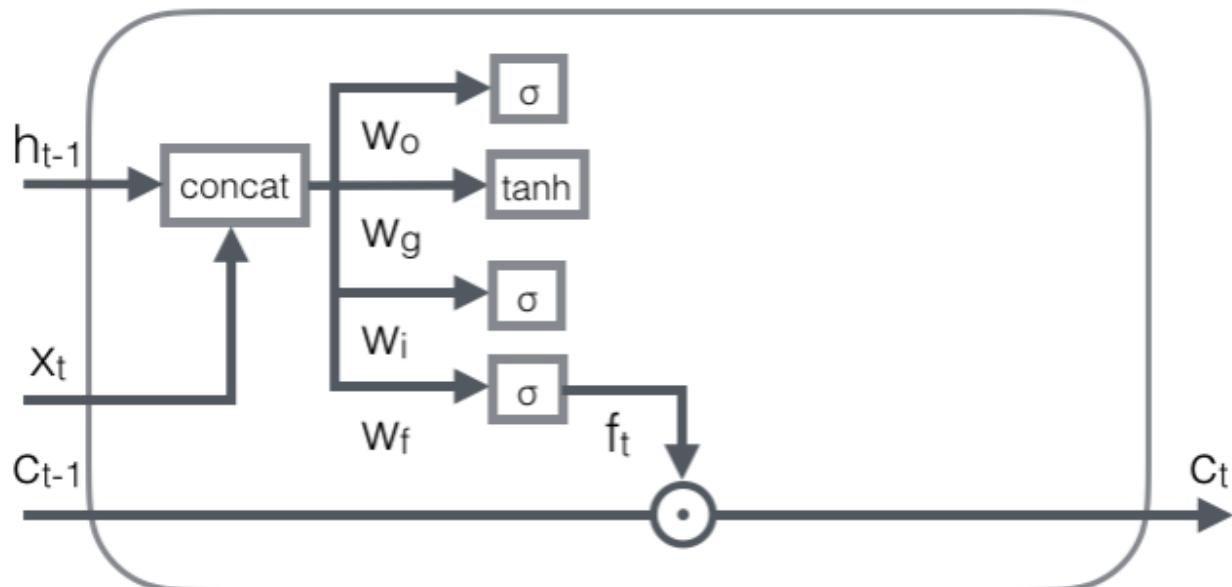
Notice that the weights here are actually weight matrices. The number of rows in the weight matrix corresponds to the number of features, while the number of columns corresponds to the number of output features, i.e. the number of nodes in the LSTM layer.

After the matrix multiplication with the weights, the results are passed through 4 independent nonlinear activation functions.



Three of these are sigmoids, yielding the output vectors with values between 0 and 1. These 3 outputs take the name of **gates**, because they control the flow of information. The last one is not a sigmoid, it is a tanh.

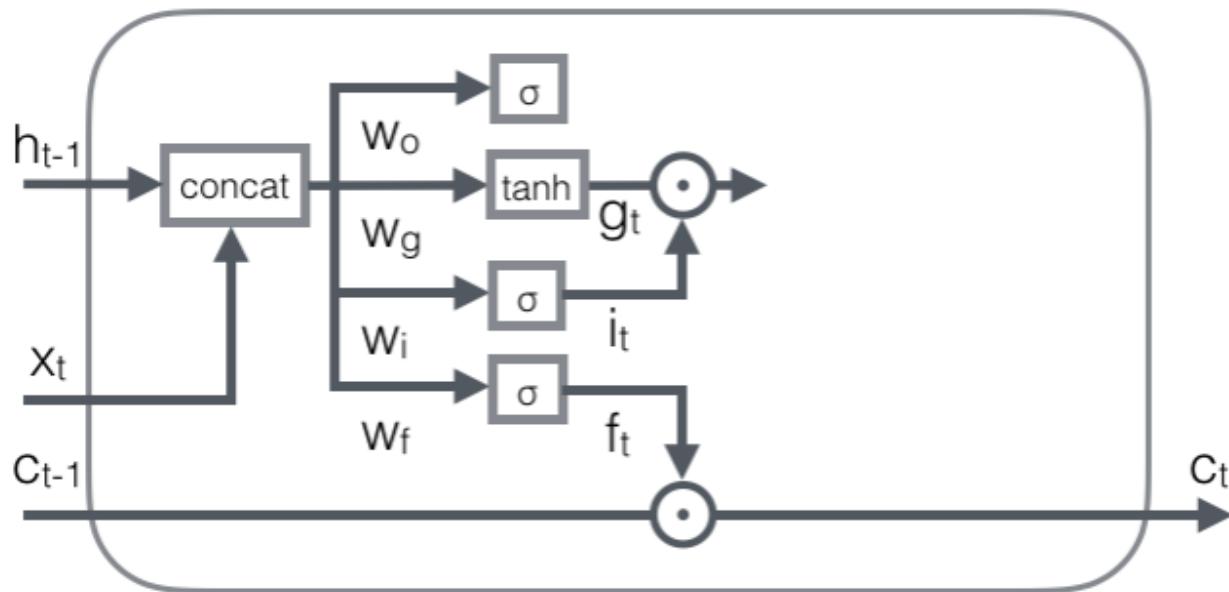
Let's now look at the role of each of these nonlinear outputs. We start from the bottom one. This is called the **forget gate**.



The role of the forget gate is to mediate how much of the internal state vector will be kept and passed through to the future times. Since the value of this gate comes from a dense layer followed by a sigmoid, the LSTM node is learning which fraction of the past data to retain and which fraction to forget.

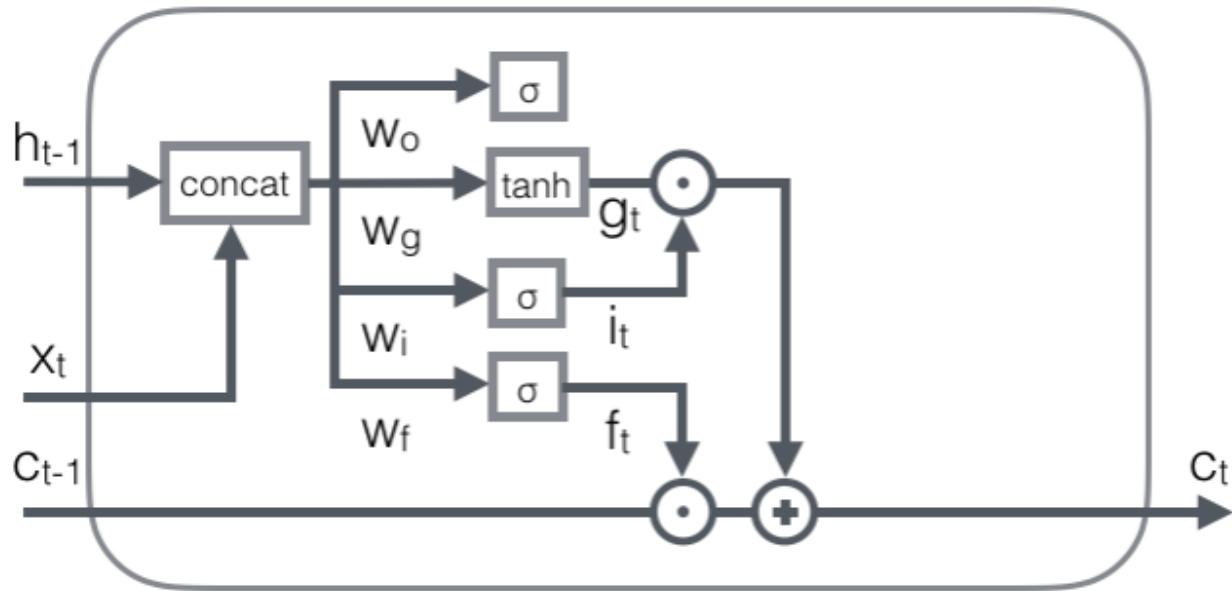
Notice that the \odot operator implies we are multiplying f_t and c_t elementwise. This fact also means they are vectors of the same length.

Let's look at the gate mediated by w_i . This gate is the **input gate** and it mediates how much of the input to keep. However, it's not the plain input concatenated vector, it's a vector that went through the \tanh layer. We call it g_t .



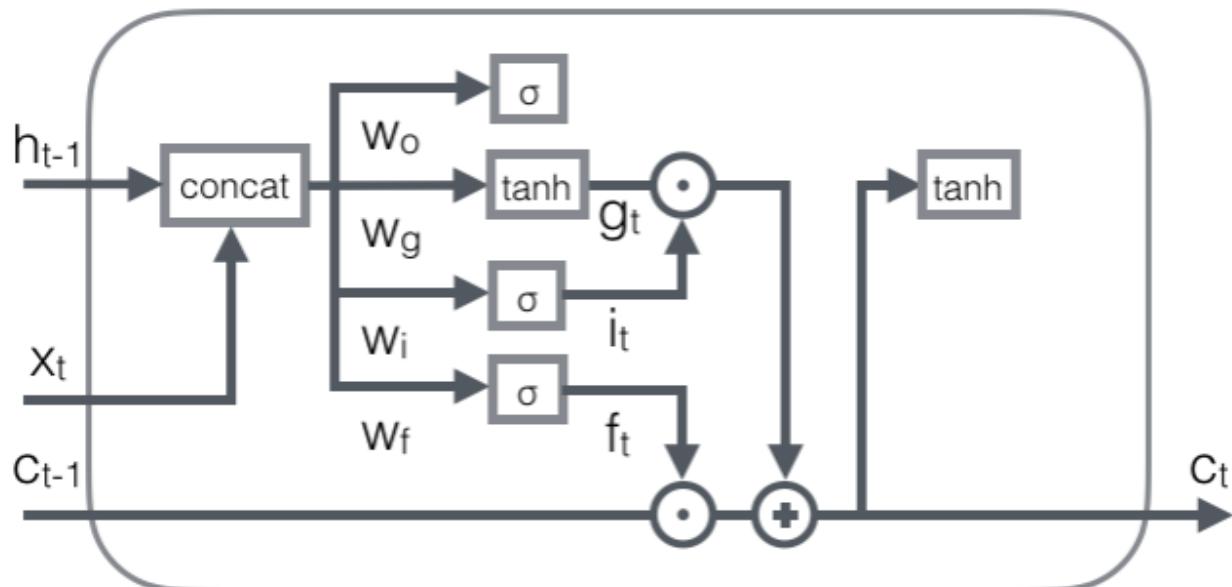
This gating operation is also performed elementwise on g_t .

The resulting vector $i_t \odot g_t$ is added to the fraction of internal state that had been retained through the forget gate.

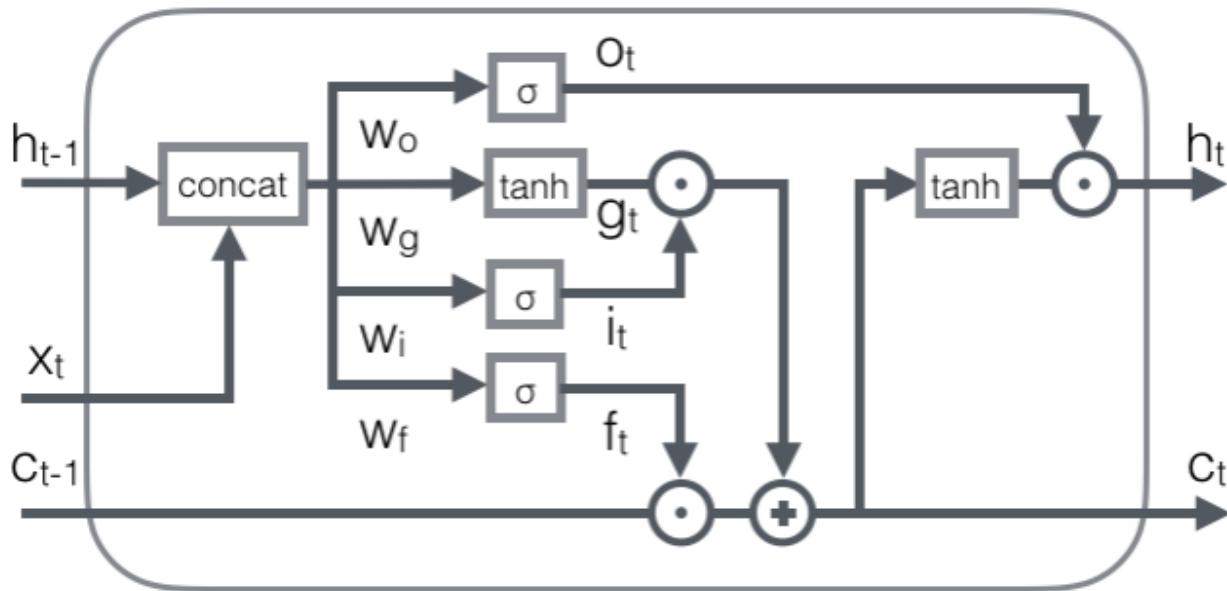


The new internal state c_t is the result of these two operations: forgetting a bit of the past state and adding some new elements coming from the input and the past output.

Now that we have the update rules for the internal state, let's see how the output state is calculated from the internal state. One last tanh operation



Now let's look at the **output gate** \mathbf{o}_t . This gate mediates the output of the tanh only allowing part of it to escape to the output.

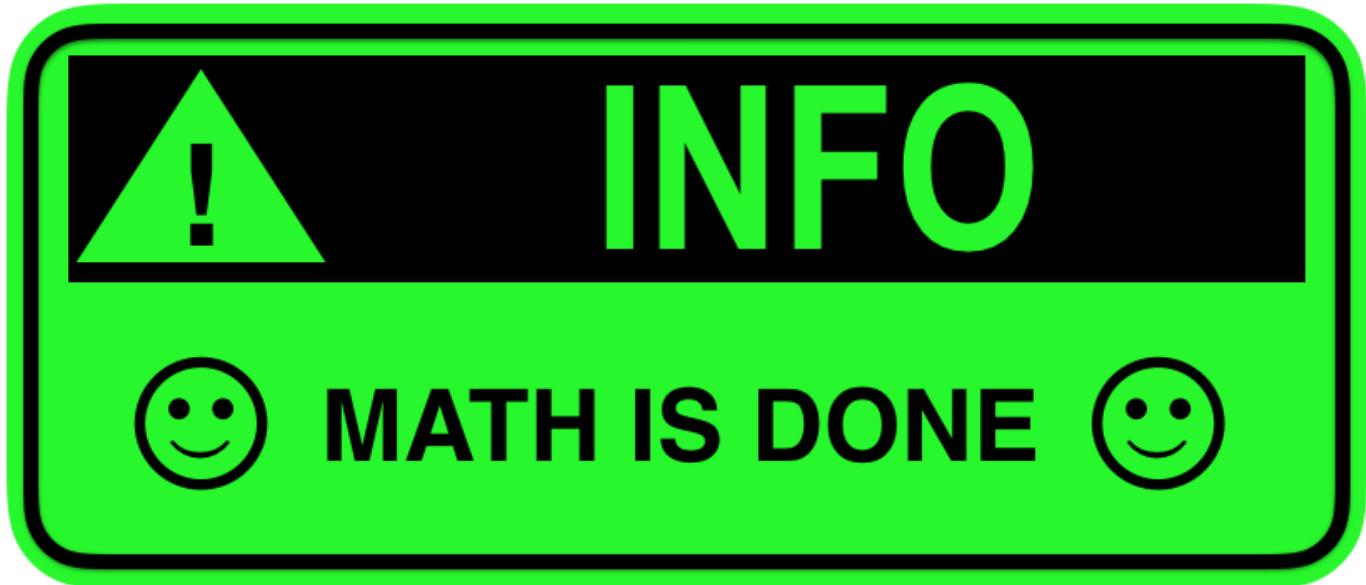


There we go! It looks complex, but that's because it's one of the most complicated units in Neural Networks.

We've just dissected the LSTM block that has revolutionised our ability to tackle problems with long term dependencies. For example, LSTM blocks have been successfully used to learn the structure of language, to produce code from text descriptions, to translate between language pairs and so on.

For the sake of completeness we will write here the equations of the LSTM, though it's not so important that you learn them: Keras has them implemented in a conveniently available `LSTM` layer!

$$\begin{aligned}
 \mathbf{n}_t &= [\mathbf{x}_t, \mathbf{h}_{t-1}] \\
 \mathbf{i}_t &= \sigma(\mathbf{n}_t \cdot \mathbf{W}_i) \\
 \mathbf{f}_t &= \sigma(\mathbf{n}_t \cdot \mathbf{W}_f) \\
 \mathbf{o}_t &= \sigma(\mathbf{n}_t \cdot \mathbf{W}_o) \\
 \mathbf{g}_t &= \tanh(\mathbf{n}_t \cdot \mathbf{W}_g) \\
 \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\
 \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
 \end{aligned}$$



LSTM forecasting

Enough with math and theory! Let's try to use an LSTM and see if we get a better result on our forecasting problem.

Let's import the `LSTM` layer from `keras`:

```
from keras.layers import LSTM
```

Let's clear the backend memory again to build our model:

```
K.clear_session()
```

Now let's build our model using the `LSTM` layer now.

TIP: according to the [documentation](#), the `LSTM` layer may have many arguments. In this case we create a layer with 6 recurrent nodes, like we had 6 units in our fully connected layer and we will set `batch_input_shape=(1, 1, 1)` and `stateful=True`, i.e. the last state for each data point will be used as initial state next data point.

- 12 ??
- `batch_input_shape`

It's also a good idea to *always* describe the arguments for layers we've already discussed before.

Like we did above, we will use the `Adam` optimizer with a small learning rate and initialize the input weights to one:

```
model = Sequential()
model.add(LSTM(6,
              batch_input_shape=(1, 1, 1),
              kernel_initializer='ones',
              stateful=True))
model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer=Adam(lr=0.0005) )
```

Now let's train our model. In doing so, we will use the `X_train_t` and `y_train_t` set for the training, the already specified `batch_size=1`, because we feed the data one point at a time, and `shuffle=False` to pass the data in order. We train the model for 2 epochs.

```
model.fit(X_train_t, y_train_t,
           epochs=2,
           batch_size=1,
           verbose=1,
           shuffle=False)
```

```
Epoch 1/2
93551/93551 [=====] - 336s 4ms/step - loss: 0.0200
Epoch 2/2
85723/93551 [=====>...] - ETA: 27s - loss: 0.0144
...
85723/93551 [=====>...] - ETA: 27s - loss: 0.0144
...
85723/93551 [=====>...] - ETA: 27s - loss: 0.0144
...
85723/93551 [=====>...] - ETA: 27s - loss: 0.0144
```

Notice that the `LSTM` takes much longer to train than `SimpleRNN`. This is because it has many more weights to adjust. In a future chapter we will learn how to use GPUs in order to speed up the training.

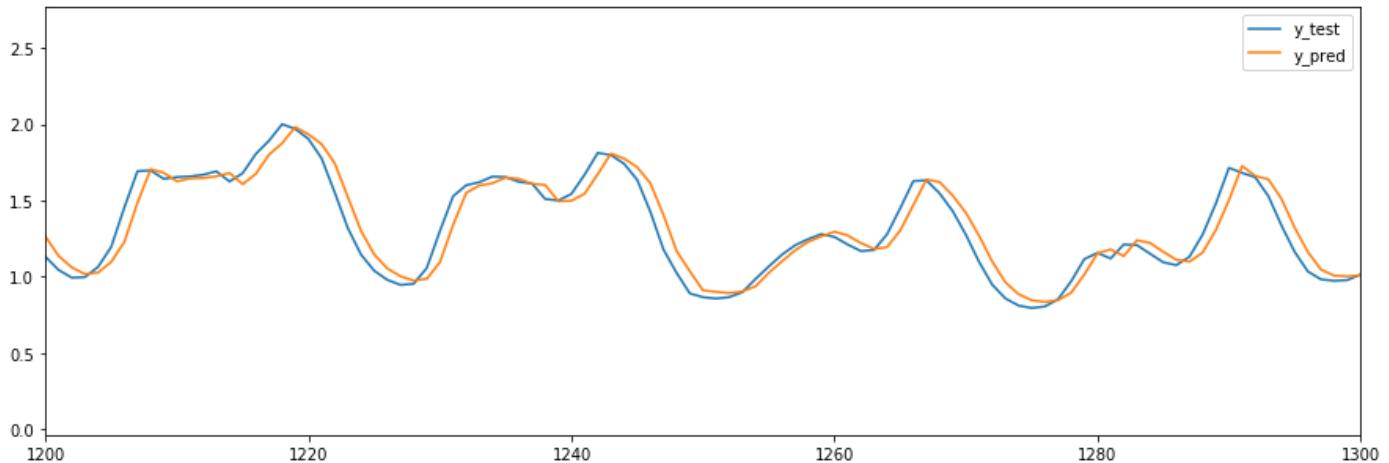
To examine the effectiveness of our model, and like we did before, we can plot a small part of the time series and compare our predictions with the true values:

```

y_pred = model.predict(X_test_t, batch_size=1, )
plt.figure(figsize=(15,5))
plt.plot(y_test_t, label='y_test')
plt.plot(y_pred, label='y_pred')
plt.legend()
plt.xlim(1200,1300)

```

(1200, 1300)



This should look better than what we have obtained previously, but even in this case we see that the ability of the network to forecast is limited. As done for the other models, let's also check the *Mean Squared Error* and the R^2 , for an objective evaluation of the error:

```

mse = mean_squared_error(y_test_t, y_pred)
r2 = r2_score(y_test_t, y_pred)

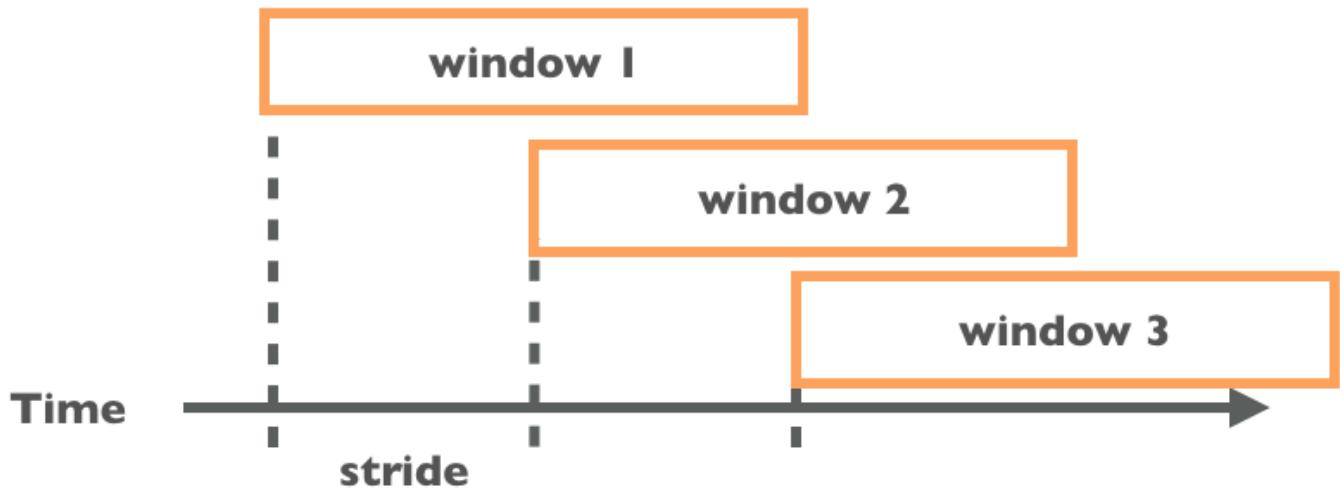
print("MSE: {:.3f}".format(mse))
print("R2: {:.3f}".format(r2))

```

MSE: 0.0128
R2: 0.943

IMPROVING FORECASTING

In all the models used so far we fed the data sequentially to our recurrent unit, one point at a time. This is not the only way in which we can train a recurrent layer. We can also use the [rolling windows approach](#).



Instead of taking a single previous point as input, we can use a set of points, going back in time for a window. This will allow us to feed data to the network in larger batches, speeding up training and hopefully improving convergence.

We will reformat our input tensor `x` to have the following shape: `(N_windows, window_len, 1)`. By doing this we treat the time series as if it was composed by many independent windows of fixed length and we can treat each window as an individual data point. This has the advantage of allowing to randomize the windows in our train and test data.

Let's start by defining the window size. We'll take a window of 24 periods, i.e. the data from the previous day. You can always adjust this later on if you wish:

```
window_len = 24
```

Next we'll use the `.shift` method of a pandas DataFrame to create lagged copies of our original time series. Note that we will start from the `train_sc` and `test_sc` vectors we've defined earlier. Let's double check that they still contain what we need:

```
train_sc.head()
```

	Total Ontario
2003-05-01 01:00:00	0.7404
2003-05-01 02:00:00	0.7156
2003-05-01 03:00:00	0.6822
2003-05-01 04:00:00	0.7002
2003-05-01 05:00:00	0.8020

To create the lagged data we define a helper function `create_lagged_Xy_win` that creates an input matrix `x` with lags going from `start_lag` to `start_lag + window_len` and an output vector `y` with the unaltered values.

So for example if we call: `create_lagged_Xy_win(train_sc, start_lag=24, window_len=168)` this will return a dataset `x` where periods run from 24 hours before to 8 days before the corresponding value in `y`.

Let's do it:

```
def create_lagged_Xy_win(data, start_lag=1, window_len=1):
    X = data.shift(start_lag).copy()
    X.columns = ['T_{}'.format(start_lag)]

    if window_len > 1:
        for s in range(1, window_len):
            X['T_{}'.format(start_lag + s)] = data.shift(start_lag + s)

    X = X.dropna()
    idx = X.index
    y = data.loc[idx]
    return X, y
```

Now we use the function on the train and test data. We will use `start_lag=1` so that we can compare the results with our previous results:

```

start_lag=1
window_len=24

X_train, y_train = create_lagged_Xy_win(train_sc, start_lag, window_len)
X_test, y_test = create_lagged_Xy_win(test_sc, start_lag, window_len)

```

Let's take a look at our data:

```
x_train.head()
```

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
2003-05-02 01:00:00	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096	1.6290
2003-05-02 02:00:00	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096
2003-05-02 03:00:00	0.7218	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486
2003-05-02 04:00:00	0.6914	0.7218	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228
2003-05-02 05:00:00	0.7018	0.6914	0.7218	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008

5 rows × 24 columns

```
y_train.head()
```

	Total Ontario
--	---------------

	Total Ontario
2003-05-02 01:00:00	0.7742
2003-05-02 02:00:00	0.7218
2003-05-02 03:00:00	0.6914
2003-05-02 04:00:00	0.7018
2003-05-02 05:00:00	0.7904

As you can see, to predict the value 0.7806 that appears in `y` at 2003-05-08 05:00:00 , in `x` we have the previous values, going back in time from 0.6950 (previous hour) to 0.6734 (two hours before) and so on.

In order to feed this data to a recurrent model we need to reshape as a tensor of order with the shape (`batch_size, timesteps, input_dim`) . We are still dealing with a univariate time series, so `input_dim=1` , while `timesteps` is going to be 168, the number of timesteps in the window. Easy to do using the `.reshape` method from numpy.

We will get numpy arrays using the `.values` attribute. We have already checked that the data is shifted correctly, so it's not a problem to throw away the index and the column names:

```
x_train_t = X_train.values.reshape(-1, window_len, 1)
X_test_t = X_test.values.reshape(-1, window_len, 1)

y_train_t = y_train.values
y_test_t = y_test.values
```

Let's check the shape of our tensor is correct:

```
X_train_t.shape
```

```
(93528, 24, 1)
```

Yes! We have correctly reshaped the tensor. Note here that if we had multiple time series, we could have bundled them together in an input vector along the last axis.

Let's build a new recurrent model. This time we will not need to use the `stateful=True` directive because some history is already included in the input data. For the same reason we will use `input_shape` instead of `batch_input_shape`.

Also, since we will use batches of more than one point, and each point contains a lot of history, the model convergence will be a lot more stable. Therefore we can increase the learning rate a lot without risking that the model becomes unstable.

```
K.clear_session()
model = Sequential()
model.add(LSTM(6, input_shape=(window_len, 1),
              kernel_initializer='ones'))
model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer=Adam(lr=0.05) )
```

Let's go ahead and train our model using a batch of size 256 for 5 epochs. This may take some time. Later in the book we will learn how to speed it up using GPUs. For now take advantage of this time with a little break. You deserve it!

```
model.fit(X_train_t, y_train_t,
           epochs=5,
           batch_size=256,
           verbose=1)
```

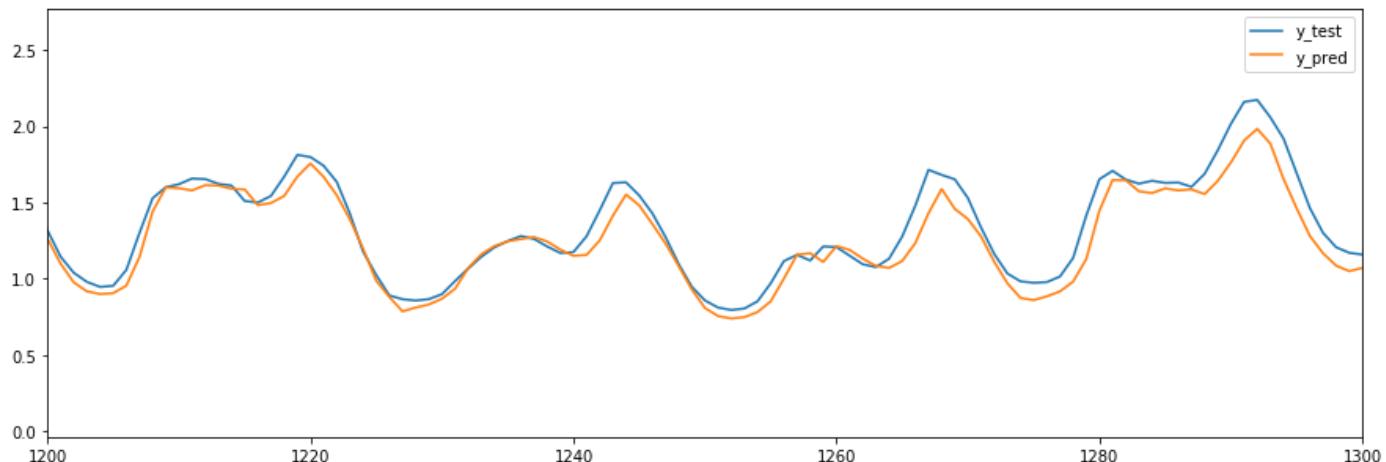
```
Epoch 1/5
93528/93528 [=====] - 10s 106us/step - loss: 0.1867
Epoch 2/5
93528/93528 [=====] - 9s 100us/step - loss: 0.0587
Epoch 3/5
93528/93528 [=====] - 9s 100us/step - loss: 0.0585
Epoch 4/5
93528/93528 [=====] - 9s 100us/step - loss: 0.0583
Epoch 5/5
...
Epoch 5/5
```

```
<keras.callbacks.History at 0x7fc6b3df5748>
```

Let's generate the predictions and compare them with the actual values:

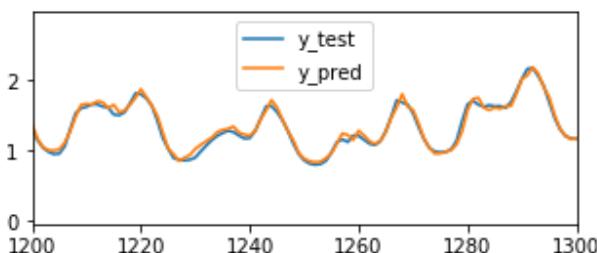
```
y_pred = model.predict(X_test_t, batch_size=256)
plt.figure(figsize=(15,5))
plt.plot(y_test_t, label='y_test')
plt.plot(y_pred, label='y_pred')
plt.legend()
plt.xlim(1200,1300)
```

(1200, 1300)



This model trained considerably faster than the previous ones and its predictions should look much better than the previous models. First of all the model seems to have learned the temporal pattern much better than the other models: it's not simply repeating the input like a parrot, it's genuinely trying to predict the future. Also, the curves look quite close to one another, which is a great sign!

TIP: Try to re-initialize and re-train the model if the loss of your model does not reach 0.05 and the above figure does not look like this:



One problem with recurrent models is that they tend to get stuck in local minima and be sensitive to

initialization. Also, keep in mind that we chose only 6 units in this network, which is probably small for this problem.

Conclusion

Well done! You have completed the chapter on Time Series and Recurrent Neural Networks. Let's recap what we have learned.

1. We learned how to classify time series of a fixed length using both fully connected and convolutional neural networks
- We learned about recurrent neural networks and about how they allow us to approach new problems with sequences, including generating a sequence of arbitrary length and learning from sequences of arbitrary length
- We trained a fully connected network to forecast future values in a sequence
- We performed a deep dive in recurrent neural networks, in particular in the Long Short-Term Memory network to see what advantage they bring
- Finally we trained an LSTM model to forecast values using both a single point as well as a window of past data

Wow, this is a lot for a single chapter!

In the exercises we will explore a couple of extensions of what we have done and we will try to predict the price of Bitcoin from its historical value!

EXERCISES

Exercise 1

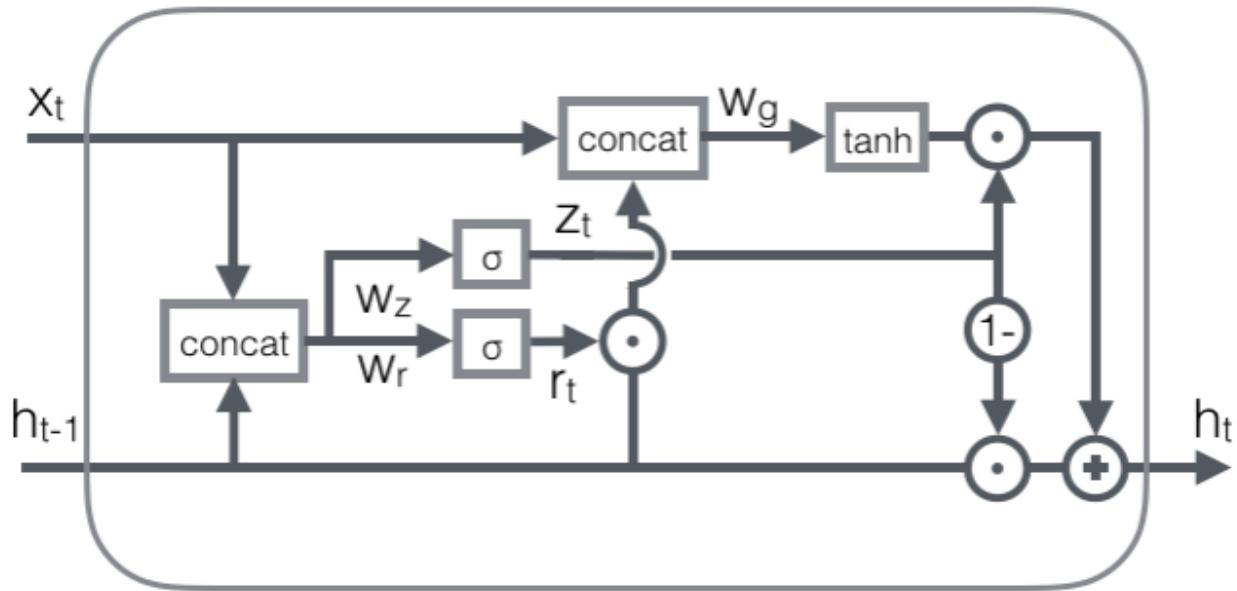
Your manager at the power company is quite satisfied with the work you've done predicting the electric load of the next hour and would like to push it further. He is curious to know if your model can predict the load on the next day or even on the next week instead of the next hour.

- Go ahead and use the helper function `create_lagged_Xy_win` we created above to generate new `x` and `y` pairs where the `start_lag` is 36 hours or even further. You may want to extend the window size to a little longer than a day.
- Train your best model on this data. You may have to use more than one layer. In which case, remember to use the `return_sequences=True` argument in all layers except for the last one so that they pass sequences to one another.
- Check the goodness of your model by comparing it with test data as well as looking at the R^2 score.

Exercise 2

Gate Recurrent Unit (GRU) are more modern and simpler implementation of a cell that retains longer term memory.

Their flow diagram is as follows:



Keras makes them available in `keras.layers.GRU`. Try swapping the LSTM layer with a GRU layer and re-train the model. Does its performance improve on the 36 hours lag task?

Exercise 3

Does a fully connected model work well using windows? Let's find out! Try to train a fully connected model on the lagged data with windows, which will probably train much faster:

- reshape the input data back to an Order-2 tensor, i.e. eliminate the 3rd axis
- build a fully connected model with one or more layers
- train the fully connected model on the windowed data. Does it work well? Is it faster to train?

Exercise 4

Predicting the price of Bitcoin from historical data.

Disclaimer: past performance is no guarantee of future results. This is not investment advice.

You have heard a lot of talk about Bitcoin and how it is growing that you decide to put your newly acquired deep learning skills to test in trying to beat the market. The idea is simple: if we could predict what Bitcoin is going to do in the future, we can trade and profit using that knowledge.

The simplest formulation of this forecasting problem is to try to predict if the price of Bitcoin is going to go up or down in the future, i.e. we can frame the problem as a binary classification that answers the question: is Bitcoin going up.

Here are the steps to complete this exercise:

1. Load the data from `../data/poloniex_usdt_btc.json.gz` into a Pandas DataFrame. This data was obtained through the public API of the Poloniex cryptocurrency exchange. Here's an example of a query, in case you wanted to download more of such data: https://poloniex.com/public?command=returnChartData¤cyPair=USDT_BTC&start=1230768000&end=1518640274&period=1800.
- Check out the data using `df.head()`. Notice that the dataset contains the close, high, low, open for 30 minutes intervals, which means: the first, highest, lowest and last amounts of US Dollars people were willing to exchange Bitcoin for during those 30 minutes. The dataset also contains Volume values, that we shall ignore, and a weighted average value, which is what we will use to build the labels.
- Convert the date column to a datetime object using `pd.to_datetime` and set it as index of the DataFrame.
- Plot the value of `df['close']` to inspect the data. You will notice that it's not periodic at all and it has an overall enormous upward trend, so we will need to transform the data into a more stationary timeseries. We will use percentage changes, i.e. we will look at relative movements in the price instead of absolute values.
- Create a new dataset `df_percent` with percent changes using the formula:

$$v_t = 100 \times \frac{x_t - x_{t-1}}{x_{t-1}}$$

this is what we will use next.

- Inspect `df_percent` and notice that it contains both infinity and nan values. Drop the null values and replace the infinity values with zero.
- Split the data at January 1st 2017, using the data before then as training and the data after that as test.
- Use the window method to create an input training tensor `x_train_t` with the shape `(n_windows, window_len, n_features)`. This is the main part of the exercise, since you'll have to make a few choices and be careful not to leak information from the future. In particular you will have to:
 - decide the `window_len` you want to use
 - decide which features you'd like to use as input (don't use `weightedAverage`, since we'll need it for the output).

- decide what lag you want to introduce between the last timestep in your input window and the timestep of the output.
- You can start from the `create_lagged_Xy_win` function we defined in Chapter 7, but you will have to modify it to work with numpy arrays because Pandas DataFrames are only good with 1 feature.
- Create a binary outcome variable that is 1 when `train[weightedAverage] >= 0` and 0 otherwise. This is going to be our label.
- Repeat the same operations on the test data
- Create a model to work with this data. Make sure the input layer has the right `input_shape` and the output layer has 1 node with a Sigmoid activation function. Also make sure to use the `binary_crossentropy` loss and to track the accuracy of the model.
- Train the model on the training data
- Test the model on the test data. Is the accuracy better than a baseline guess? Are you going to be rich?

Again disclaimer: past performance is no guarantee of future results. This is not investment advice.

Chapter 8: Natural Language Processing and Text Data

In this chapter we will learn a few techniques to approach problems involving text. This a very important topic since text data is very common.

We will start by introducing text data and some use cases of Machine Learning and Deep Learning applied to text prediction. Then we will explore the traditional approach to text problems: the **Bag of Words** (BOW) approach.

This topic will take us to explore how to extract features from text. We will introduce new techniques to do this as well as a couple of new Python packages specifically designed to deal with text data.

We will explore the limitations of the BOW approach and see how Neural Networks can help to overcome them. In particular we will look at embeddings to encode text and at how they can be used in Keras. Finally we will discuss **large scale language modeling**, **skip-grams** and **Word2Vec**. Let's get started!

USE CASES

As noted in the introduction text data is encountered in many applications. Let's take a look at a few of them.

Spam Detection is probably the one we are all familiar with. It is a text classification problem where we try to distinguish legitimate documents from extraneous documents.

Spam detection can be applied to email spam, sms spam, im spam and in general any corpus of messages. The problem is usually presented as a binary classification where one has two sets of documents: the spam messages and the "ham" messages, i.e. the legitimate messages that we would like to keep.

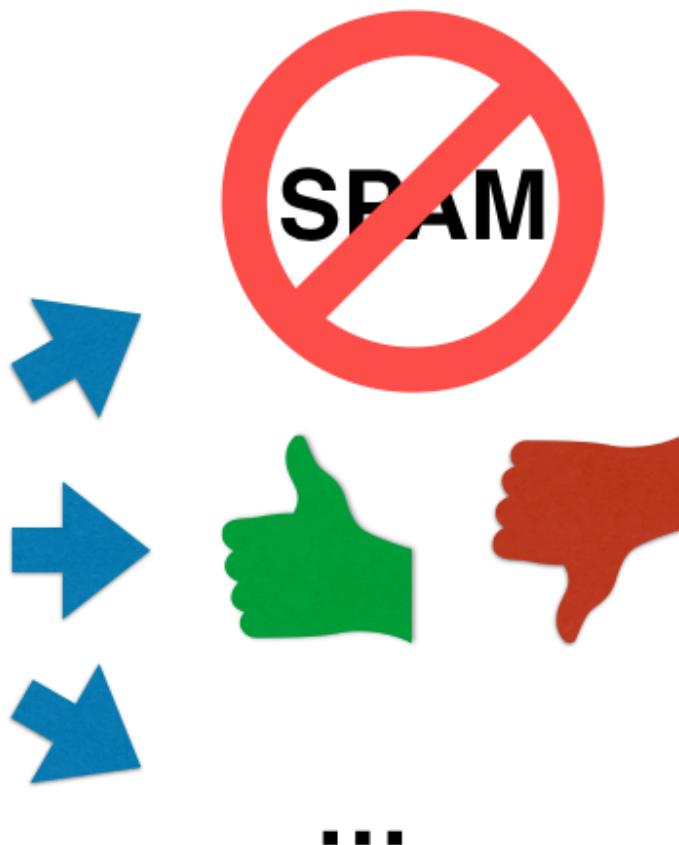
A similar binary classification problem involving text is that of **Sentiment Analysis**.

Imagine you are a rock star tweeting about your latest album. Millions of people will reply to your tweet and it will be impossible for you to read all of the messages from your fans. You would like to capture the overall sentiment of your fan base and see if they are happy about what you tweeted.

Sentiment analysis does that by classifying a piece of text as *positive* or *negative* in regard to the overall sentiment. If you know the sentiment for each tweet it's easy to draw conclusions like: 74% of your fans responded positively to your tweet".

Sentiment Analysis is widely applied to many fields including stock trading, e-commerce reviews, customer service and in general any website or application where users are allowed to submit free-form text comments.

er. The license will always be a rectangle. A good way to handle the orientation of the license plate in the image would be to generate a fake license plate. We could do this with a small set of real plates, then rotate them in 3D space. It we do not need to store the training examples for the model. These examples are lost. In this way we could generate them anywhere. Also, since we are not bound by passing batches of training data to training from streaming data. Data is collected sequentially. After that, data is discarded.



Extending beyond classification problems, we can consider regression problems involving text, for example extracting a score, a price, or any other metric starting from a text document. An example of this would be estimating the number of followers your tweet will generate based on its text content or predicting the number of downloads your application will do based on the content of a blog article.

All the above problems are traditional Machine Learning problems where text is the input to the problem. Text can also be the output of a Machine Learning problem. For example, **Machine Translation** involves converting text from a language to another. It is a supervised, many-to-many, sequence learning problem, where pairs of sentences in two languages are fed to a model that learns to generate the output sequence (for example a sentence in English), given a certain input sequence (the corresponding sentence in Italian).

Machine translation is an example of a whole category of Machine Learning problems involving text: problems involving **automatic text generation**. Another famous example in this category is that of **Language Modeling**.

In Language Modeling a *corpus of documents* (see next section for a proper definition) is fed sequentially to a model. The model will learn the probability distribution of a certain word to appear after a sentence. The

model is then sampled randomly and is capable of producing sentences that resemble the properties of the corpus. Using this approach people had models produce new sonnets from Shakespeare, new chapters of Harry Potter, new episodes of popular novels and so on.

Since Language Modeling works on sequences, we can also build character level models that learn the syntax or our input corpus. In this way we can produce syntactically accurate markup languages like HTML, Wiki, Latex and even C! See the [wonderful article by Andrej Karpathy](#) for a few examples of this.

It is clear that text is involved in many useful application. So let's see how to prepare text documents for Machine Learning.

TEXT DATA

Loading text data

Text data is usually a collection of articles or documents. Linguists call this collection a **corpus** to indicate that it's coherent and organized. For example we could be dealing with the corpus of patents from our company or with a corpus of articles from a news platform.

The first thing we are going to learn is how to load text data using Scikit-Learn. We will build a simple Spam detector to separate sms containing spam from legitimate sms messages. The data comes from the [UCI SMS Spam collection](#), but it has been re-organized and re-compressed.

The file `data/sms.zip` is a compressed archive of a folder with the structure:

```
sms
| -- ham
|   | -- msg_000.txt
|   | -- msg_001.txt
|   | -- msg_003.txt
|   +-- ...
|
+-- spam
    | -- msg_002.txt
    | -- msg_005.txt
    | -- msg_008.txt
    +-- ...
```

Let's extract all the data into the data folder:

First, let's import the `zipfile` package from python so that we can extract the data into folders:

```
import zipfile
```

`zipfile` allows to operate directly zipped folder into our workspace. Have a look at the documentation for further details. Here we use it to extract the data for later loading it:

```
with zipfile.ZipFile('../data/sms.zip', 'r') as fin:
    fin.extractall('../data/')
```

This last operation created a folder called `sms` inside the `data` folder. Let's look at its content. The `os` module contains a lot of functions to interact with the host system. Let's import it:

```
import os
```

And let's use the command `os.listdir` to look at the content of the folder:

```
os.listdir('../data/sms')
```

```
['spam', 'ham']
```

As expected there are two subfolders: `ham` and `spam`. We can count how many files they contain with the help of the following little function that lists the content of `path` and uses a filter to only count files.

```
def count_files(path):
    files_list = [name for name in os.listdir(path)
                  if os.path.isfile(os.path.join(path, name))]
    return len(files_list)
```

Let's use this function to count the number of files in the folders:

```
ham_count = count_files('../data/sms/ham/')
ham_count
```

```
4825
```

```
spam_count = count_files('../data/sms/spam/')
spam_count
```

```
747
```

We have 4825 ham files and 747 spam files. We can use these numbers to establish a benchmark for our classification efforts:

```
benchmark_acc = ham_count / (ham_count + spam_count)
print("Benchmark accuracy: {:.3f}".format(benchmark_acc))
```

```
Benchmark accuracy: 0.866
```

If we always predicted the large class, i.e. we never predicted spam, we would be correct 86.6% of the time. Our model needs to score higher than that to be of any help.

Let's also look at a couple of examples of our messages for each class:

```
def read_file(path):
    with open(path) as fin:
        msg = fin.read()
    return msg
```

```
read_file('../data/sms/ham/msg_000.txt')
```

```
'Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine
```

```
read_file('../data/sms/ham/msg_001.txt')
```

```
'Ok lar... Joking wif u oni...'
```

```
read_file('../data/sms/spam/msg_002.txt')
```

```
"Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to ..."
```

```
read_file('../data/sms/spam/msg_005.txt')
```

```
"FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun you
```

As expected, spam messages look quite different from ham messages. In order to start building a spam detection model, let's first load all the data into a dataset.

Scikit Learn offers a function to load text data from folders for classification purposes. Let's use the `load_files` function from `sklearn.datasets` package:

```
from sklearn.datasets import load_files
```

```
data = load_files('../data/sms/', encoding='utf-8')
```

`data` is an object of type:

```
type(data)
```

```
sklearn.utils.Bunch
```

The documentation for a `Bunch` reads:

```
vocabulary-like object, the interesting attributes are: either  
data, the raw text data to learn, or 'filenames', the files  
holding it, 'target', the classification labels (integer index),  
'target_names', the meaning of the labels, and 'DESCR', the full  
description of the dataset.
```

so let's look at the available keys:

```
data.keys()
```

```
dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR'])
```

Following the documentation, let's assign the `data.data` and `data.target` to two variables `docs` and `y`.

```
docs = data.data
```

The first five text examples in our `docs` variable are:

```
docs[:5]
```

```
['Hi Princess! Thank you for the pics. You are very pretty. How are you?',
 "Hello my little party animal! I just thought I'd buzz you as you were with your friends",
 'And miss vday the parachute and double coins?? U must not know me very well...',
 'Maybe you should find something else to do instead??',
 'What year. And how many miles.']
...
'What year. And how many miles.'
...
'What year. And how many miles.'
```

```
y = data.target
```

The first five entries in our `y` variable:

```
y[:5]
```

```
array([0, 0, 0, 0, 0])
```

Before we do anything else, let's save the data we have loaded as a DataFrame, just in case we need to reload it later. As usual we import our workhorses `numpy`, `pandas` and `pyplot`:

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

```
%matplotlib inline  
import matplotlib.pyplot as plt
```

and then create a Dataframe with all the documents

```
import pandas as pd  
df = pd.DataFrame(docs, columns=['message'])  
df['spam'] = y  
df.head()
```

	message	spam
0	Hi Princess! Thank you for the pics. You are v...	0
1	Hello my little party animal! I just thought I...	0
2	And miss vday the parachute and double coins??...	0
3	Maybe you should find something else to do ins...	0
4	What year. And how many miles.	0

Pandas allows to save a dataframe to a variety of different formats, including Excel, CSV and SAS. We will export to CSV:

```
df.to_csv('../data/sms_spam.csv', index=False, encoding='utf8')
```

Feature extraction from text

A machine learning algorithm is not able to deal with text as it is. Instead, we need to extract features from the text!

I image. We could do this with a small
se plates, then rotates them in 3D spac
it we do not need to store the training i
examples for the model. These examp
hen lost. In this way we could generat
where. Also, since we are not bound b
ly passing batches of training data to t
ning from streaming data. Data is colle
nformation. After that data is disc



X ₁	X ₂	X ₃	X ₄	...
0.1	0.4	0.3	0.0	...
0.5	0.1	0.0	0.9	...
...

Let's begin with a naive solution and gradually build up to a more complex one. The simplest way to build features from text is to use the counts of certain words that we assume to carry information about the problem.

For example, spam messages often offer something for free or give a link to some service. Since these are sms message, this link will likely be a number. With these two ideas in mind, let's build a very simple classifier that uses only two features:

- The count of the occurrence of the word "free".
- The count of numerical characters.

Notice that our text contains uppercase and lowercase words, so as a preprocessing step let's convert everything to lowercase so we don't include meaningless features.

```
docs_lower = [d.lower() for d in docs]
```

The first five entries in our `docs_lower` variable are:

```
docs_lower[:5]
```

```
['hi princess! thank you for the pics. you are very pretty. how are you?',  
 "hello my little party animal! i just thought i'd buzz you as you were with your friends  
 'and miss vday the parachute and double coins??? u must not know me very well...',  
 'maybe you should find something else to do instead???' ,  
 'what year. and how many miles.' ]  
 ...  
 'what year. and how many miles.]
```

```
...
'what year. and how many miles.]
```

We can define a simple helper function that counts the occurrences of a particular word in a sentence:

```
def count_word(word, sentence):
    tokens = sentence.split()
    return len([w for w in tokens if w == word])
```

and apply it to each document:

```
df = pd.DataFrame([count_word('free', d) for d in docs_lower], columns=['free'])
```

```
df.head()
```

	free
0	0
1	0
2	0
3	0
4	0

Similarly let's build a helper function that counts the numerical character in a sentence using the `re` package:

```
import re
```

```
def count_numbers(sentence):
    return len(re.findall('[0-9]', sentence))
```

```
df['num_char'] = [count_numbers(d) for d in docs_lower]
```

```
df.head()
```

	free	num_char
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0

Spam classification

Notice that most messages don't contain our special features, so we don't expect any model to work super well in this case, but let's try to build one anyways. First, let's import the `train_test_split` function from `sklearn` as well as the the usual `Sequential` model and the `Dense` layer:

```
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: Fu
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Now let's define the helper function that follows the usual process that we repeated several times in the previous chapters:

- Train/test split
- Model definition

- Model training
- Model evaluation on test set

We will use a simple [Logistic Regression model](#) to start, to make things simple and quick:

```
def split_fit_eval(X, y, model=None, epochs=10, random_state=0):
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=random_state)

    if not model:
        model = Sequential()
        model.add(Dense(1, input_dim=X.shape[1], activation='sigmoid'))

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

    h = model.fit(X_train, y_train, epochs=epochs, verbose=1)

    loss, acc = model.evaluate(X_test, y_test)

    return loss, acc, model, h
```

Executing the function with our values, we'll capture the result in a variable we'll call `res` :

```
res = split_fit_eval(df.values, y)
```

```
Epoch 1/10
4179/4179 [=====] - 2s 379us/step - loss: 2.2069 - acc: 0.8629
Epoch 2/10
4179/4179 [=====] - 0s 76us/step - loss: 2.1131 - acc: 0.8629
Epoch 3/10
4179/4179 [=====] - 0s 76us/step - loss: 1.9875 - acc: 0.8629
Epoch 4/10
4179/4179 [=====] - 0s 76us/step - loss: 1.7851 - acc: 0.8626
Epoch 5/10
...
Epoch 5/10
```

Let's check the accuracy of our model:

```
print("Accuracy of the simple model is: {:.3f}".format(res[1]))
```

```
Accuracy of the simple model is: 0.963
```

Despite our initial skepticism, this dataset is easy to separate! In fact, it is so easy that two very simple features (the counts of the word `free` and the count of numerical characters) already achieve a much better accuracy score than the benchmark, that was 0.866.

Bag of Words features

We can extend the simple approach of the previous model in a few ways:

- We could build a vocabulary with more than just one word, and build a feature for each of them which counts how many times that word appears.
- We could filter out common English words.

Scikit Learn has a transformer that allows to do exactly these two tasks, it's called `CountVectorizer`. Let's import it from `sklearn.feature_extraction.text`:

```
from sklearn.feature_extraction.text import CountVectorizer
```

Let's plan on using the top 3000 most common words in the corpus, this is going to be our **vocabulary size**:

```
vocab_size = 3000
```

Then we can initialize the vectorizer. Here we have to use the additional argument `stop_words='english'` that tells the vectorizer to **ignore common English stop words**. We do this because we are ranking features starting from the most common word. If we didn't ignore common words, we would end up with word features like "if", "and", "of" etc. at the top of our list, since these words are just very common in the English language. However, these words do not carry much meaning about spam and by ignoring them we get word features that are more specific to our corpus.

We are also going to ignore decoding errors using the `decode_error='ignore'` argument:

```
vect = CountVectorizer(decode_error='ignore',
                      stop_words='english',
```

```
lowercase=True,  
max_features=vocab_size)
```

Notice that it also allows for automatic lowercase conversion. You can check what are the stop words using the `.get_stop_words()` method:

```
vect.get_stop_words()
```

```
frozenset({'a',  
           'about',  
           'above',  
           'across',  
           'after',  
           'afterwards',  
           'again',  
           'against',  
           'all',  
           ...  
           'yourselves'})
```

Now that we have created the vectorizer, let's apply it to our corpus:

```
X = vect.fit_transform(docs)  
X
```

```
<5572x3000 sparse matrix of type '<class 'numpy.int64'>'  
with 37142 stored elements in Compressed Sparse Row format>  
...  
with 37142 stored elements in Compressed Sparse Row format>  
...  
with 37142 stored elements in Compressed Sparse Row format>
```

`X` is a **sparse matrix** i.e. a matrix in which most of the elements are 0. This makes sense since most messages are short and they will only contain a few of the 3000 words in our feature list. The `X` matrix has 5572 rows (i.e. the total number of sms) and 3000 columns (i.e. the total number of selected words) but only 37142 non-zero entries (less than 1%).

In order to use it for machine learning we will convert it to a dense matrix, which we can do by calling `todense()` on the object:

```
Xd = X.todense()
```

TIP: be careful with converting sparse matrices to dense. If you are dealing with large datasets you will quickly run out of memory with all those zeros. In those cases we do on-the-fly conversion to dense of each batch during Stochastic Gradient Descent.

Let's also have a look at the features found by the vectorizer:

```
vocab = vect.get_feature_names()
```

The features are listed in alphabetical order:

```
vocab[:10]
```

```
['00',
 '000',
 '02',
 '0207',
 '02073162414',
 '03',
 '04',
 '05',
 '06',
 ...
 '07123456789']
```

```
vocab[-10:]
```

```
['yogasana', 'yor', 'yr', 'yrs', 'yummy', 'yun', 'yunny', 'yuo', 'yup', 'zed']
```

Let's use the helper function we've defined above to train a model on the new features:

```
res = split_fit_eval(Xd, y)
```

```
Epoch 1/10  
4179/4179 [=====] - 1s 140us/step - loss: 0.6171 - acc: 0.8787  
Epoch 2/10  
4179/4179 [=====] - 0s 85us/step - loss: 0.4888 - acc: 0.9665  
Epoch 3/10  
4179/4179 [=====] - 0s 84us/step - loss: 0.4008 - acc: 0.9694  
Epoch 4/10  
4179/4179 [=====] - 0s 84us/step - loss: 0.3377 - acc: 0.9734  
Epoch 5/10  
...  
Epoch 5/10
```

```
print("The Accuracy score on the Test set is:\t{:0.3f}".format(res[1]))
```

```
The Accuracy score on the Test set is: 0.973
```

The accuracy on the test set is not much higher than our simple model, however, we can use this model to look for features importances, i.e. to identify words whose weight is high when predicting spam or not. Let's recover the trained model from the `res` object returned by our custom function:

```
model = res[2]
```

Then let's put the weights in a Pandas Series, indexed by the vocabulary:

```
vocab_weights = pd.Series(model.get_weights()[0].ravel(), index=vocab)
```

Let's look at the top 20 words with positive weights:

```
vocab_weights.sort_values(ascending=False).head(20)
```

txt	0.617662
www	0.562382

```
mobile      0.545765
claim       0.522931
uk          0.518820
150p        0.506219
reply        0.498766
free         0.491721
service      0.443694
...
dtype: float32
```

Not surprisingly we find here words like `www` , `claim` , `prize` , `cash` etc. Similarly we can look at the bottom 20 words:

```
vocab_weights.sort_values(ascending=False).tail(20)
```

```
yeah      -0.453327
lor       -0.464423
wat       -0.472508
later     -0.473824
got       -0.474368
oh        -0.481702
sorry     -0.487159
did       -0.490643
da        -0.495354
...
dtype: float32
```

and see they are pretty common legitimate words like `sorry` , `ok` , `lol` , etc... If we were spammer we could take advantage of this information and craft messages that attempt to fool these simple features by using a lot of words like `sorry` or `ok` . This is a typical **Adversarial Machine Learning** scenario, where the target is constantly trying to beat the model.

In any case, it's pretty clear that this dataset is an easy one. So let's load a new dataset and learn a few more tricks!

Word frequencies

In the previous spam classification problem we used a `CountVectorizer` transformer from Scikit Learn to produce a sparse matrix with term counts of the top 3000 words. Using the absolute counts was ok because the corpus was formed by sms messages, whose length is capped at 160 characters. In the general case,

using absolute counts may be a problem if we deal with documents of uneven length. Think of a brief email versus a long article, both about the topic of AI. The word AI will appear in both documents, but it will likely be repeated more times in the long article. Using the counts would lead us to think that the article is more about AI than the short email, while it's simply a longer text.

We can account for that using the **term frequency** instead of the count, i.e. by dividing the counts by the length of the document. Using term frequencies is already an improvement, but we can do even better.

In fact, there will be some words that are common in every document. These could be **common english stop words** (like: *a, and, if, on*, etc.), but they could also be words that are common across the specific corpus. For example, if we are trying to sort a corpus of patents by topics, it is clear that words like: *patent, application, grant* and similar legal terms will be common across the whole corpus and not really indicative of the particular topic of each of the documents in the corpus.

We want to normalize our term frequencies with a term inversely proportional to the fraction of documents containing that term, i.e. we want to use an **inverse document frequency**.

These features go by the name of **TF-IDF** i.e. **term frequency-inverse document frequency**, which is also available as a vectorizer in Scikit Learn.

In other words, TF-IDF features look like:

```
TFIDF(word, document) = counts_in_document(word, document) / counts_of_documents_with_wo
```

or using maths:

$$\text{tf-idf}(w, d) = \frac{\text{tf}(w, d)}{\text{df}(w, d)}$$

where w is a word, d is a document, tf stands for "term frequency" and df for "document frequency".

As you can read in the [Wikipedia](#) article, there are several ways to improve the above of the **TF-IDF** formula, using different regularization schemes. [Scikit Learn](#) implements it as follows:

$$\text{tf-idf}(w, d) = \text{tf}(w, d) \times \log \left(\frac{1+n_d}{1+\text{df}(w, d)} \right) + 1$$

where n_d is the total number of documents and the regularized logarithm takes care of words that are extremely rare or extremely common.

Sentiment classification

Let's load a new dataset, containing reviews from the popular website Rotten Tomatoes:

```
df = pd.read_csv('../data/movie_reviews.csv')
df.head()
```

	title	review	vote
0	Toy story	So ingenious in concept, design and execution ...	fresh
1	Toy story	The year's most inventive comedy.	fresh
2	Toy story	A winning animated feature that has something ...	fresh
3	Toy story	The film sports a provocative and appealing st...	fresh
4	Toy story	An entertaining computer-generated, hyperreali...	fresh

Let's take a peek into the data we loaded, see how many reviews we have and a few other pieces of information.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14072 entries, 0 to 14071
Data columns (total 3 columns):
title      14072 non-null object
review     14072 non-null object
vote       14072 non-null object
dtypes: object(3)
memory usage: 329.9+ KB
```

Let's look at the division between the `fresh` votes, the `rotten`, and the `none` votes.

```
df['vote'].value_counts() / len(df)
```

```
fresh      0.612067
rotten     0.386299
none       0.001634
Name: vote, dtype: float64
...
Name: vote, dtype: float64
...
Name: vote, dtype: float64
```

As you can see, the dataset contains reviews about famous movies and a judgment of `rotten VS fresh`, which is the class we will try to predict.

First of all, we notice that a small number of reviews do not have a class, so let's eliminate those few rows from the dataset. We'll do this by selecting all the `votes` that are *not* `none`:

```
df = df[df.vote != 'none'].copy()
```

```
df['vote'].value_counts() / len(df)
```

```
fresh      0.613069
rotten     0.386931
Name: vote, dtype: float64
...
Name: vote, dtype: float64
...
Name: vote, dtype: float64
```

Our reference accuracy is 61.3%, the fraction of the larger class.

Label encoding

Notice that the labels are strings, and we need to convert them to 0 and 1 in order to use them for classification. We could do this in many ways, one way is to use the `LabelEncoder` from Scikit Learn. It is a transformer that will look at the unique values present in our label column and encode them to numbers from 0 to $N - 1$, where N is the number of classes, in our case 2.

Let's first import it from `sklearn.preprocessing`:

```
from sklearn.preprocessing import LabelEncoder
```

Let's instantiate the `LabelEncoder` :

```
le = LabelEncoder()
```

Finally, let's create a vector of 0s and 1s that represent the features:

```
y = le.fit_transform(df['vote'])
```

`y` is now a vector of 0s and 1s. Let's look at the first 10 entries:

```
y[:10]
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Bag of words: TF-IDF features

Let's import the `TfidfVectorizer` vector from Scikit Learn:

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

Let's initialize it to look at the top 10000 words in the corpus, excluding English stop words:

```
vocab_size = 10000

vect = TfidfVectorizer(decode_error='ignore',
                      stop_words='english',
                      max_features=vocab_size)
```

We can use the vectorizer to transform our reviews:

```
X = vect.fit_transform(df['review'])
```

```
X
```

```
<14049x10000 sparse matrix of type '<class 'numpy.float64'>'  
with 130025 stored elements in Compressed Sparse Row format>  
...  
    with 130025 stored elements in Compressed Sparse Row format>  
...  
    with 130025 stored elements in Compressed Sparse Row format>
```

This generates a sparse matrix with 14049 rows and 10000 columns. This is still small enough to be converted to dense and passed to our model evaluation function. Let's call `todense()` on the object to convert it to a dense matrix:

```
Xd = X.todense()
```

Let's train our model. We will use a higher number of epochs in this case, to ensure convergence with the larger dataset:

We'll use our function again:

```
res = split_fit_eval(Xd, y, epochs=30)
```

```
Epoch 1/30  
10536/10536 [=====] - 1s 118us/step - loss: 0.6744 - acc: 0.6148  
Epoch 2/30  
10536/10536 [=====] - 1s 100us/step - loss: 0.6463 - acc: 0.6159  
Epoch 3/30  
10536/10536 [=====] - 1s 99us/step - loss: 0.6279 - acc: 0.6159  
Epoch 4/30  
10536/10536 [=====] - 1s 100us/step - loss: 0.6123 - acc: 0.6169  
Epoch 5/30  
...  
Epoch 5/30
```

```
print("The Accuracy score on the Test set is:\t{:0.3f}".format(res[1]))
```

```
The Accuracy score on the Test set is: 0.749
```

The accuracy on the test set is much lower than the last value of accuracy obtained on the training set (last line printed during training), therefore the model is overfitting. This is not unexpected given the large number of features. Despite the overfitting, the test score is still higher than the 61.3% accuracy obtained by always predicting the larger class.

Text as a sequence

The bag of words approach is very crude. It does not take into account context, i.e. each word is treated as independent feature, regardless of its position in the sentence. This is particularly bad for tasks like sentiment analysis where negations could be present ("This movie was not good") and the overall sentiment could not be carried by any particular word.

In order to go beyond the bag of words approach we need to treat text as a sequence instead of just looking at frequencies. In order to do this, we will proceed to:

1. create a vocabulary, indexed starting from the most frequent word and then continuing in decreasing order.
2. convert the sentences to sequences of integer indices using the vocabulary
3. feed the sequences to a neural network in order to perform the sentiment classification

Keras has a preprocessing `Tokenizer` that allows us to create a vocabulary and convert the sentences using it. Let's load it:

```
from keras.preprocessing.text import Tokenizer
```

Let's initialize the `Tokenizer`. We will use the same vocabulary size of 10000 used in the previous task:

```
vocab_size
```

```
10000
```

```
tokenizer = Tokenizer(num_words=vocab_size)
```

We can fit the tokenizer on our reviews using the function `.fit_on_texts`. We will pass the column of the dataframe `df` that contains the reviews:

```
tokenizer.fit_on_texts(df['review'])
```

Great! The tokenizer has finished its job, so let's give a look at some of its attributes.

The `.document_count` gives us the number of documents used to build the vocabulary:

```
tokenizer.document_count
```

```
14049
```

These are the 14049 reviews left in the dataset after we removed the ones without a vote. The `.num_words` attribute gives us the number of features in the vocabulary. These should be 10000:

```
tokenizer.num_words
```

```
10000
```

Finally, we can retrieve the word index by calling `.word_index`, which returns a vocabulary. Let's look at the first 10 items in it:

```
list(tokenizer.word_index)[:10]
```

```
['the', 'a', 'and', 'of', 'to', 'is', 'in', 'it', 'that', 'as']
```

As you can see this is not sorted alphabetically, but in decreasing order of frequency starting from the most common word. Let's use the tokenizer to convert our reviews to sequences. For instance, "The movie is great" translates to the sequence 1539 :

The movie is great ...

↓ ↓ ↓ ↓
1 5 3 9 ...

```
sequences = tokenizer.texts_to_sequences(df['review'])
```

sequences is a list of lists. Each of the inner lists is one of the reviews:

```
sequences[:3]
```

```
[[36,  
 1764,  
 7,  
 1058,  
 800,  
 3,  
 1765,  
 9,  
 27,  
 ...  
 [2, 1012, 347, 225, 9, 24, 107, 14, 564, 21, 1, 354, 7122]]
```

Let's just double check that the conversion is correct by converting the first list back to text. We will need to use the reverse index -> word map:

```
idx_to_word = {i:w for w, i in tokenizer.word_index.items()}
```

The first review is:

```
df.loc[0, 'review']
```

'So ingenious in concept, design and execution that you could watch it on a postage stamp

The first sequence is:

```
' '.join([idx_to_word[i] for i in sequences[0]])
```

'so ingenious in concept design and execution that you could watch it on a postage stamp :

The two sentences are almost identical, however notice a couple of things:

1. punctuation has been stripped away in the tokenization.
- all words are lowercased.
- some really rare word (e.g. "engulfed") are out of our top 3000 words and are therefore ignored.

Now that we have sequences of numbers, we can organize them into a matrix with one review per row and one word per column. Since not all sequences have the same length, we will need to pad the short ones with zeros.

Let's calculate the longest sequence length:

```
maxlen = max([len(seq) for seq in sequences])
maxlen
```

49

The longest review contains 49 words. Let's pad every other review to 49 using the `pad_sequences` function from Keras:

```
from keras.preprocessing.sequence import pad_sequences
```

As you can read in the documentation:

Signature: `pad_sequences(sequences, maxlen=None, dtype='int32', padding='pre', truncating='post')`
Docstring:
Pads each sequence to the same length (length of the longest sequence).

If maxlen is provided, any sequence longer than maxlen is truncated to maxlen.

pad_sequences operates on the sequences by padding and truncating them. Let's set the maxlen parameter to the value we already found:

```
X = pad_sequences(sequences, maxlen=maxlen)
```

```
X.shape
```

```
(14049, 49)
```

x has 14049 rows (i.e. the number of samples, review in this case) and 49 columns (i.e. the words of the longest review). Let's print out the first few reviews:

```
X[:4]
```

```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0, 36, 1764,  7, 1058,  800,  3, 1765,
       27, 151, 268,  8, 21,  2, 9088, 3879, 5881, 115,
      101, 20, 22, 17, 360],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
       ...,
       1678,    4,   13,   742,   724]], dtype=int32)
```

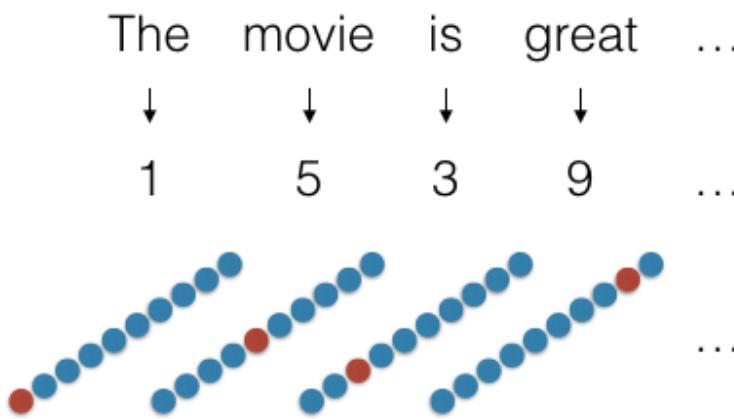
Can we feed this matrix to a Machine Learning model? Let's think about it for a second. If we treat this matrix as tabular data, it would mean each column represents a feature. But what feature? Columns in the matrix correspond to the position of the word in a sentence and so there's absolutely no reason why two words appearing at the same position would carry coherent information about sentiment.

Also, the numbers here are the indices of our words in a vocabulary, so their actual value is not a quantity, it's their rank in order of frequency in the vocabulary. In other words, word number 347 is not 347 times as

large as the word at index 1, it's just the word that appears at index 347 in the vocabulary.

In fact, the correct way to think of this data is to recognize that each number in `x` really represents an index in a vector with length `vocab_size`, i.e. a vector with 10000 entries. These are the actual features, i.e. all the words in our vocabulary.

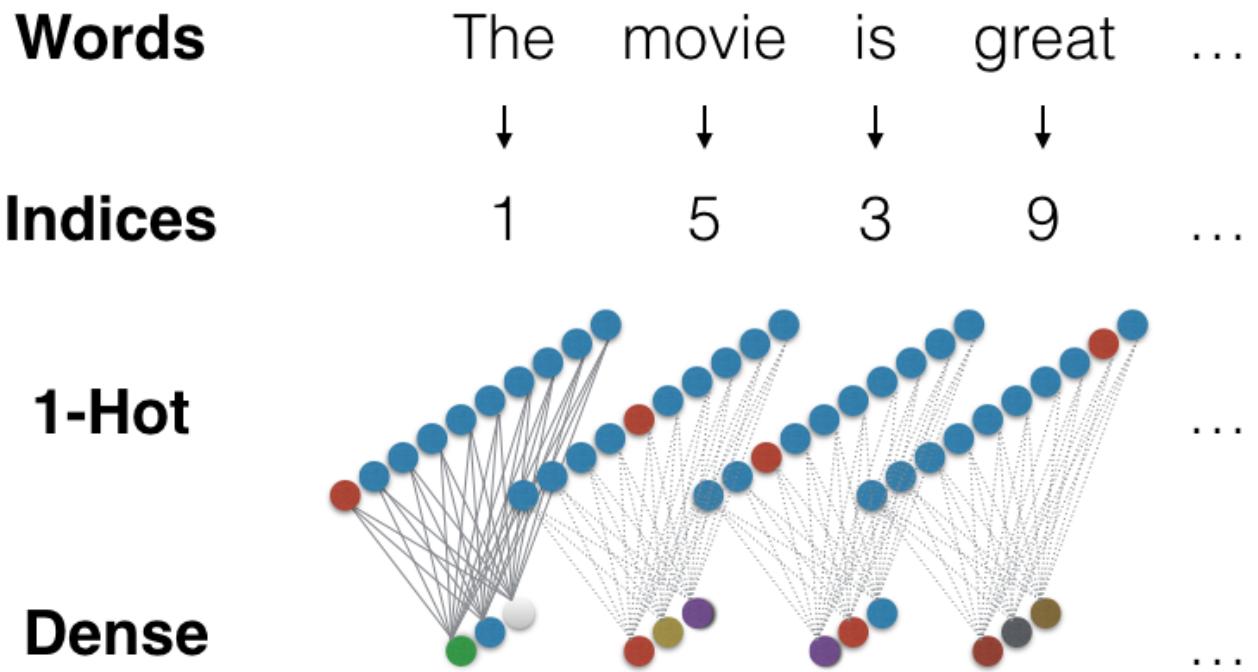
So, this matrix is really a shorthand for an order-3 sparse tensor whose three axes are (sentence, position along sentence, word feature index). The first axis would locate the sentence in the dataset and it corresponds to the row axis of our `x` matrix. The second axis would locate the word position in the sentence and it corresponds to the column axis of our `x` matrix. The third axis would locate the word in the vocabulary and it corresponds to actual value in the entry of the matrix.



It looks like one way to feed this data to a neural network would be to expand the `x` matrix to a 1-hot encoded order-3 tensor with 0s and 1s, and then feed this tensor to our network, for example to a [Recurrent layer](#) with `input_shape=(49, 10000)`. This would be the equivalent of feeding a dataset of 10000 time series, whose elements are all mostly zeros, except for 1 at each time, which is not zero when that word occurs in that particular sentence.

While this encoding works, it is not really memory efficient. Besides that, representing each word along a different orthogonal axis in a 10000-dimensional vector space doesn't capture any information on how that word is used in its context. How can we improve this situation?

One idea would be to insert a fully connected layer to compress our input space from the very large sparse space of 10000 words in our vocabulary to a much smaller dense space, for example with just 32 axes. In this new space each word is represented by a dense vector, whose entries are floating point numbers instead of all 0s and a single 1.



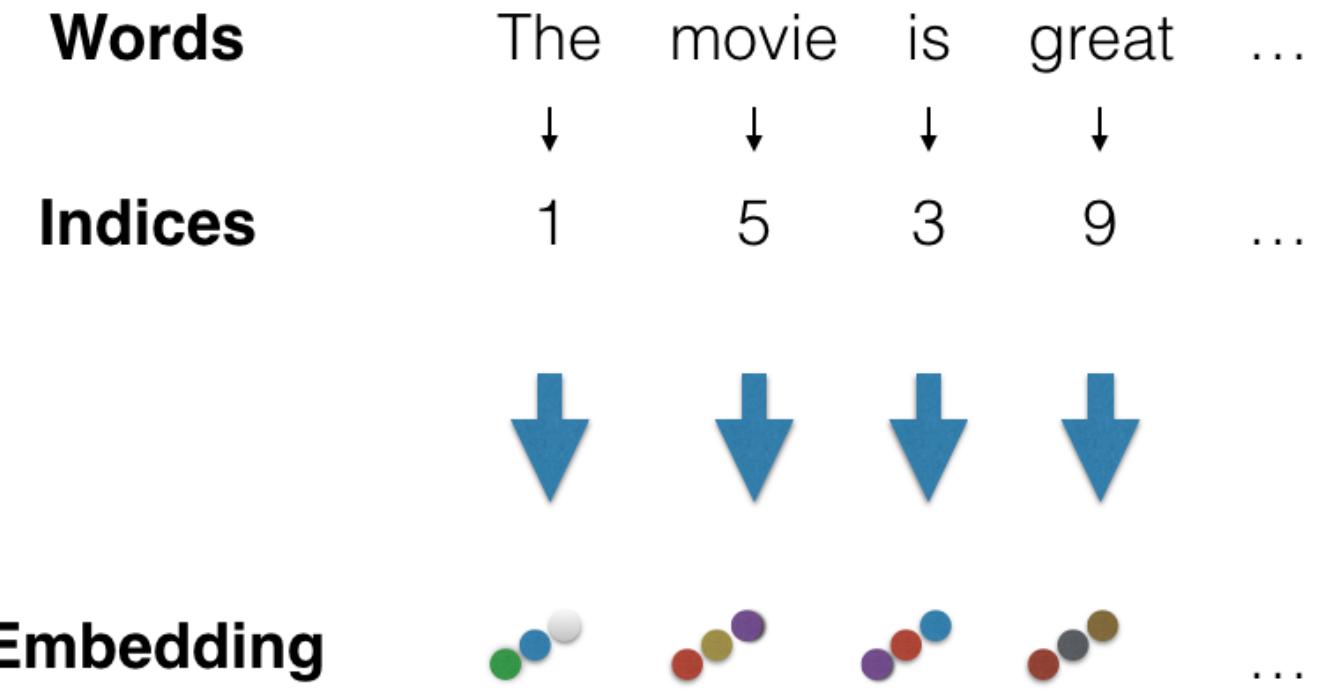
This is really cool, because now we can feed our sequences of much smaller dense vectors to a recurrent network in order to complete the sentiment classification task, i.e. we are treating the sentiment classification problem as a [Sequence Classification problem](#) like the ones encountered in Chapter 7.

Furthermore, since the dense vector is obtained through a fully connected layer, we can jointly train the fully connected layer and the recurrent layer allowing the fully connected layer to find the best representation for the words in order to help the recurrent layer achieve its task.

Embeddings

In practice we never actually go through the burden of converting the word indices to 1-hot vectors and then back to dense vectors. We use an **Embedding layer**. In this layer we specify the output dimension, i.e. the length of the dense vector and it has an independent set of as many weights for each of the words in the vocabulary. So, for example, if the vocabulary is 10000 words and we specify an output dim of 100, the embedding layer will carry 1000000 weights, 100 for each of the words in the vocabulary.

The numbers in input will be understood as the index that selects the set of 100 weights, i.e. they will be interpreted as indices in a phantom sparse space, saving us from converting the data to 1-hot and then converting it back to dense.



Let's see how to include this in our own network. Let's load the `Embedding` layers from Keras:

```
from keras.layers import Embedding
```

Let's see how it works by creating a network with a single such layer that maps a feature space of 100 words to an output dense space of only 2 dimensions:

```
model = Sequential()
model.add(Embedding(input_dim=100, output_dim=2))
model.compile(optimizer='sgd', loss='categorical_crossentropy')
```

The network above assumes the input will be made of numbers between 0 and 99. These are interpreted as the indices of the single non-zero entry in a 100-dimensional 1-hot vector. Sequences of such indices will be interpreted as sequences of such vectors and will be transformed to sequences 2-dimensional dense vectors, since 2 is the dimension of the output space.

Let's feed a single sequence of a few indices and perform a forward pass:

```
model.predict(np.array([[ 0, 81, 1, 0, 79]]))
```

```

array([[[ -0.01848672,   0.04419189],
       [  0.04020962,   0.02369903],
       [  0.01303122,   0.04238135],
       [-0.01848672,   0.04419189],
       [-0.0146019 ,   0.01175102]]], dtype=float32)
...
[[-0.0146019 ,   0.01175102]]], dtype=float32)
...
[[-0.0146019 ,   0.01175102]]], dtype=float32)

```

The embedding layer turned the sequence of five numbers into a sequence of five 2-dimensional vectors. Since we have not trained our Embedding Layer yet, these are just the weight vectors corresponding to each word, so for example, words `o` corresponds to the weights `[-0.03977597, -0.01466479]`. Notice how these appear both on the first row and on the fourth row, exactly as one would expect since the words `o` appears at the first and fourth positions in our five words sentence.

Similarly if we feed a batch of few sequences of indices, we will obtain a batch of few sequences of vectors i.e. a tensor of order three, with axes (sentence, position in sentence, embedding) :

```

model.predict(np.array([[ 0,  81,   1,  96,  79],
                      [ 4,  17,  47,  69,  50],
                      [15,  49,   3,  12,  88]]))

```

```

array([[[ -0.01848672,   0.04419189],
       [  0.04020962,   0.02369903],
       [  0.01303122,   0.04238135],
       [-0.01654088,   0.00373893],
       [-0.0146019 ,   0.01175102]],

      [[ 0.02288331,   0.01345874],
       [-0.04597616,   0.00985124],
       [-0.00511587,   0.02233421],
       ...,
       [ 0.04102742,   0.04139148]]], dtype=float32)

```

Great! Now we know what to do to build our sentiment classifier. We will:

- Split as usual our `x` matrix of indices into train and test
- Build a network with:

- Embedding
- Recurrent
- Dense
- Classify the sentiment of our reviews

Let's start from the train/test split. As done several times in the book we set `random_state=0` so that we all get the same train/test split.

TIP: Setting the random state is useful when you want to have repeatable random splits.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
X.shape
```

(14049, 49)

Recurrent model

Let's build our model as we did in the previous chapter. First, let's import the `LSTM` layer from Keras:

```
from keras.layers import LSTM
```

Next, let's build up our model. We'll create our `Embedding` layer followed by our `LSTM` layer and the regular `Dense` and `Activation` layers after that:

```
model = Sequential()
model.add(Embedding(input_dim=vocab_size,
                    output_dim=16,
                    input_length=maxlen))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

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

Let's train our model using the `fit()` function. We will train the model on batches of 128 reviews for 8 epochs with a 20% validation split:

```
h = model.fit(X_train, y_train, batch_size=128, epochs=8, validation_split=0.2)
```

```
Train on 8428 samples, validate on 2108 samples
Epoch 1/8
8428/8428 [=====] - 5s 539us/step - loss: 0.6673 - acc: 0.6153 -
Epoch 2/8
8428/8428 [=====] - 4s 448us/step - loss: 0.5848 - acc: 0.6679 -
Epoch 3/8
8428/8428 [=====] - 4s 445us/step - loss: 0.3902 - acc: 0.8392 -
Epoch 4/8
8428/8428 [=====] - 4s 448us/step - loss: 0.2635 - acc: 0.9021 -
...
8428/8428 [=====] - 4s 448us/step - loss: 0.2635 - acc: 0.9021 -
```

The model seems to be doing much better on the training set than any of the previous models based on Bag of Words, since it achieves an accuracy greater than 95% in only 10 epochs. On the other hand, the validation accuracy sees to be consistently lower, which indicates probable overfitting. Let's evaluate the model on the test set in order to verify the ability of our model to generalize:

```
loss, acc = model.evaluate(X_test, y_test, batch_size=32)
acc
```

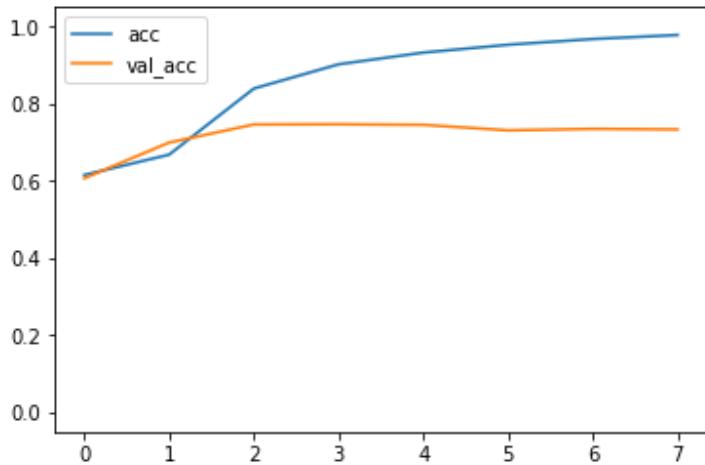
```
3513/3513 [=====] - 1s 383us/step
```

0.7389695416343812

Ouch! The test score is not much better than the score obtained by our BOW model. This means the model is overfitting. Let's plot the training history:

```
dfhistory = pd.DataFrame(h.history)
dfhistory[['acc', 'val_acc']].plot(ylim=(-0.05, 1.05))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe98cc6e2e8>
```

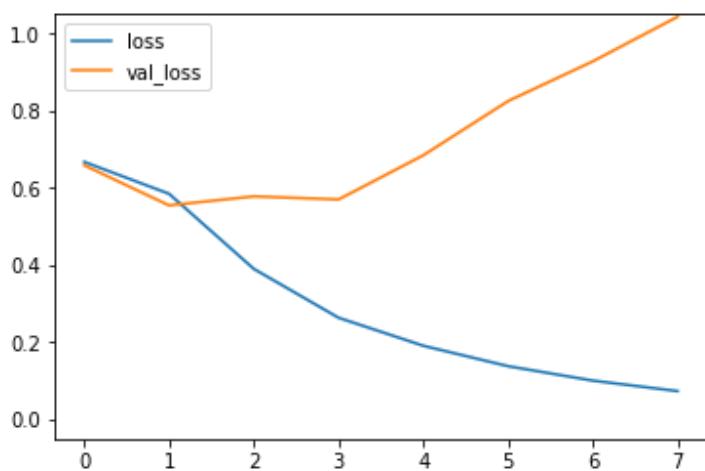


As you can see, after a few of epochs the validation accuracy stops improving while the training accuracy keeps improving.

We can also look at the loss and notice that the validation loss does not decrease after a certain point, while the training loss does.

```
dfhistory[['loss', 'val_loss']].plot(ylim=(-0.05, 1.05))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe98c191940>
```



How many weights are there in our model? Is it too big?

```
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
embedding_2 (Embedding)	(None, 49, 16)	160000
lstm_1 (LSTM)	(None, 32)	6272
dense_4 (Dense)	(None, 1)	33
<hr/>		
<hr/>		

The model is quite big compared to the size of the dataset. We have over 160 thousand parameters to classify less than 15 thousand short reviews. This is not a good situation and overfitting is expected. In the exercises we will repeat the sentiment prediction on a larger corpus of reviews and see if we can get better results.

We will also learn another way to reduce overfitting later in the book, when we discuss about pre-trained models.

SEQUENCE GENERATION AND LANGUAGE MODELING

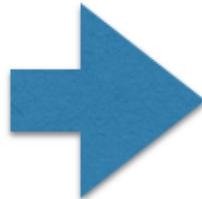
As mentioned at the beginning of this chapter and in the previous one, Neural Networks are not only suited to deal with textual input data but also to generate text output. In fact, text can be viewed as a **sequence of words** or as a **sequence of characters** and we can use a model to predict the next character or words in a sequence. This is called **Language Modeling** and it has been successfully used to generate "Shakespeare-sounding" poems, new pages of Wikipedia and so on (see [this wonderful article by A. Karpathy](#) for a few examples).

The basic idea is to apply to text the same approach we used to [improve forecasting](#) in the time series prediction of the last chapter.

We will start from a corpus of text, split it into short, fixed-size windows, i.e. sub-sentences with a few characters, and then train a model to predict the next character after the sequence.

What are "windows" here? What do they refer to?

... this is a long sentence that we are trying to model using a character-based recurrent Neural Network model as explained in ...



Text	Char
this is a	I
his is a l	o
is is a lo	n
s is a lon	g
is a long	
is a long	s
...	...

Let's give an example by designing an RNN to generate names of babies. We will use this corpus as training data, which contains thousands of names.

We start by loading all the names from `../data/names.txt`. We also add a `\n` character to allow the model to learn to predict the end of a name and convert the names to lowercase.

```
with open('../data/names.txt') as f:  
    names = f.readlines()  
    names = [name.lower().strip() + '\n' for name in names]  
  
print('Loaded %d names' % len(names))
```

```
Loaded 7939 names
```

Let's have a look at the first three of them:

```
names[:3]
```

```
['aamir\n', 'aaron\n', 'abbey\n']
```

We need to count all of the characters in our "vocabulary" and build a vocabulary that translates between the character and its assigned index (and vice versa). We could do this using the `Tokenizer` from Keras, but it is so simple that we can do it by hand using a Python `set`:

```
chars = set()  
  
for name in names:  
    chars.update(name)  
  
vocab_size = len(chars)
```

Let's look at the number of chars we've saved:

```
vocab_size
```

```
chars
```

```
{'\n',
'-',
'a',
'b',
'c',
'd',
'e',
'f',
'g',
...
'z'}
```

Now let's create two dictionaries, one to go from characters to indices and the other to go back from indices to characters. We'll use these two dictionaries a bit later.

```
char_to_inds = dict((c, i) for i, c in enumerate(chars))
inds_to_char = dict((i, c) for i, c in enumerate(chars))
```

Character sequences

We can use the vocabulary created above to translate each name in `names` to its number format in `int_names`. We will achieve this using a nested [list comprehension](#) where we iterate on names and for each name we iterate on characters:

```
int_names = [[char_to_inds[c] for c in name] for name in names]
```

Now each name has been converted to a sequence of integers, for example, the first name:

```
names[0]
```

```
'aamir\n'
```

Was converted to:

```
int_names[0]
```

```
[4, 4, 8, 22, 14, 12]
```

Great! Now we want to create short sequences of few characters and try to predict the next. We will do this by cutting up names into input sequence of length `maxlen` and using the following character as training labels. Let's start with `maxlen = 3` :

```
maxlen = 3

name_parts = []
next_chars = []

for name in int_names:
    for i in range(0, len(name) - maxlen):
        name_parts.append(name[i: i + maxlen])
        next_chars.append(name[i + maxlen])
```

`name_parts` is a list with short fractions of names (three characters). Let's take a look at the first elements:

```
name_parts[:4]
```

```
[[4, 4, 8], [4, 8, 22], [8, 22, 14], [4, 4, 14]]
```

`next_chars` is a list with single entries, each representing the next character:

```
next_chars[:4]
```

```
[22, 14, 12, 21]
```

As a last step we convert the nested list of `name_parts` to an array. We can do this, using the same `pad_sequences` function used earlier in this chapter. This takes the nested list and converts it to an array trimming the longer sequences and padding the shorter sequences:

```
X = pad_sequences(name_parts, maxlen=maxlen)
```

The final shape of our input is:

```
X.shape
```

```
(32016, 3)
```

i.e. we have 32016 name parts, each with 3 consecutive characters.

Now let's deal with the labels. We can use the `to_categorical` function to 1-hot encode the targets. Let's import it from `keras.utils`:

```
from keras.utils import to_categorical
```

Now let's create our categories from the `next_chars` using this function. Notice that we let Keras know how many characters are in the vocabulary by setting `num_classes=vocab_size` in the second argument of the function:

```
y = to_categorical(next_chars, vocab_size)
```

The shape of our labels is:

```
y.shape
```

```
(32016, 28)
```

i.e. we have 32016 characters, each represented by a 1-hot encoded vector of `vocab_size` length.

Recurrent Model

At this point we are ready to design and train our model.

We will need to set up an embedding layer for the input, one or more recurrent layers and a final dense layer with softmax activation to predict the next character. We can design the model using the Sequential API as usual or we can start to practice with the [Functional API](#), which we will use more often later on. This API is much more powerful than the [Sequential API](#) we used so far, because it allows us to build models that can have more than one processing branch. It is good to start approaching it on a simple case, so that we will be more familiar with it when we use it on larger and more complex models.

Let's import the `Model` class from `keras` and the `Input` layer:

```
from keras.models import Model  
from keras.layers import Input
```

Using the Functional API, each layer is treated as a function, which receives the output of the previous layer and it returns an output to the next. When we specify a model in this way, we need to start from an `Input` layer with the correct shape.

Since we have padded our name subsequences to a length of 3, we'll create an `Input` layer with shape `(3,)`:

TIP: remember that the trailing comma is needed in Python to distinguish a tuple with one element from a simple number within parentheses.

```
inputs = Input(shape=(3, ))
```

Let's look at the `inputs` variable we have just defined:

```
inputs
```

```
<tf.Tensor 'input_1:0' shape=(?, 3) dtype=float32>
```

It's a Tensorflow tensor with `shape=(?, 3)`, i.e. it will accept batches of data with 3 features, exactly as we want. Next we create the `Embedding` layer, with input dimension equals to the vocabulary size (i.e. 28) and

output dimension equal to 5.

```
emb = Embedding(input_dim=vocab_size, output_dim=5)
```

Next we will use this layer as a function, i.e. we well pass the `inputs` tensor to it and save the output tensor to a temporary variable called `h` (for hidden).

```
h = emb(inputs)
```

Note that we could have achieve the previous two operations in a single line by writing:

```
h = Embedding(input_dim=vocab_size, output_dim=5)(inputs)
```

Following this style we define the net layer to be an `LSTM` layer with 8 units and we reuse the `h` variable for its output:

```
h = LSTM(8)(h)
```

Finally we create the output layer, a `Dense` layer with as many nodes as `vocab_size` and with a Softmax activation function:

```
outputs = Dense(vocab_size, activation='softmax')(h)
```

Now that we have created all the layers we need and connected their inputs and outputs, let's create a model. This is done using the `Model` class that needs to know what the inputs and outputs of the model are:

```
model = Model(inputs=inputs, outputs=outputs)
```

From here onwards we proceed in an identical way to what we've been doing with the Sequential API. We compile the model for a classification problem:

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

and now we are ready to train it. We will let the training run for at least 10 epochs. While the model trains, let us reflect on a couple of questions:

- Will this model reach 99% accuracy?
- Will any model ever reach 99% accuracy on this task?
- Would this change if we had access to a corpus of millions of names?
- What accuracy would you expect from randomly guessing the next character?

Let's train our model by using the `fit()` function. We will run the training for 20 epochs:

```
model.fit(X, y, epochs=20)
```

```
Epoch 1/20
32016/32016 [=====] - 7s 224us/step - loss: 2.6391 - acc: 0.2458
Epoch 2/20
32016/32016 [=====] - 7s 203us/step - loss: 2.4707 - acc: 0.2563
Epoch 3/20
32016/32016 [=====] - 7s 204us/step - loss: 2.3767 - acc: 0.2690
Epoch 4/20
32016/32016 [=====] - 7s 204us/step - loss: 2.3381 - acc: 0.2756
Epoch 5/20
...
Epoch 5/20
```

```
<keras.callbacks.History at 0x7fe98cd8a0f0>
```

Great! The model has finished training. Getting above 30% accuracy is a good result in this case. The reason is, we are trying to predict the next character after a sequence of three characters, but there is no unique solution to this prediction problem.

Think for example of the 3 characters `and`. How many names are there in the dataset that start with `and` ?

- `anders` -> next char is `e`
- `andie` -> next char is `i`
- `andonis` -> next char is `o`
- `andre` -> next char is `r`

- andrea -> next char is r
- andreas -> next char is r
- andrej -> next char is r
- andres -> next char is r
- andrew -> next char is r
- andrey -> next char is r
- andri -> next char is r
- andros -> next char is r
- andrus -> next char is r
- andrzej -> next char is r
- andy -> next char is y

From this example we see that while r is the most frequent answer, it's not the only one. Other letters could come after the letters and in our training set.

By training the model on the truncated sequences we are effectively teaching our model a probability distribution over our vocabulary. Using the example above, given the sequence of characters ['a', 'n', 'd'] the model is learning that the character r appears 11/15 times, i.e. it has a probability of 0.733, while the characters e, i, o, y each appear 1/15 times, i.e. each has a probability of 0.066.

TIP: For the math inclined reader, the model is learning to predict the probability $p(c_t | c_{t-3}c_{t-2}c_{t-1})$ where the index t indicates the position of a character in the name. $p(A|B)$ is the **conditional probability** of A given B. This is the probability that A will happen when B has already happened.

Since the vocabulary size is 28, if the next character would have been predicted using a random uniform distribution over the vocabulary, on average we would predict correctly only 1 time every 28 trials, which would give an accuracy of about 3.6%. We get to an accuracy of about 30%, which is 10x higher than random.

Sampling from the model

Now that the model is trained, we can use it to produce new names, that should at least sound like English names. We can sample the model by feeding in a few letters and using the model's prediction for the next letter. Then we feed the model's prediction back in to get the next letter, etc.

First of all, let's define a helper function called `sample`. This function has to take an array of probabilities $\mathbf{p} = [p_i]_{i \in \text{vocab}}$ for the characters in the vocabulary and return the index of a character,

with probabilities according to **p**. This means that if a character has a high probability, its index will be returned more often than a character with a low probability.

The **multinomial distribution** is a generalization of the binomial distribution that can help us in this case. It is implemented in Numpy and its [documentation](#) reads:

```
The multinomial distribution is a multivariate generalization of the binomial distribution. Take an experiment with one of ``p`` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents `n` such experiments. Its values, ``x_i = [x_0, x_1, ..., x_p]``, represent the number of times the outcome was ``i``.
```

This description says that if our experiment has three possible outcomes with probabilities $[0.25, 0.7, 0.05]$, a single multinomial experiment will return an array of length three, where all the entries will be zero except one, that will be a 1, corresponding to the randomly chosen outcome for that experiment. If we were to repeat the experiments multiple times, the frequencies of each outcome would tend towards the assigned probabilities.

Therefore we can implement the sample function as:

```
sample(p) := argmax(multinomial(1, p, 1))
```

In fact, we are going to generalize this a bit more, introducing a parameter called **diversity** that rescales the probabilities. For high values of the diversity, the probability vectors will be squished to zero and we will be approaching the random uniform distribution. When the diversity is low, the most likely characters will be selected even more often, approaching a deterministic character generator.

Let's create the `sample` function that accepts an input list with a `diversity` argument that allows us to rescale the probabilities as an argument:

```
def sample(p, diversity=1.0):
    p1 = np.asarray(p).astype('float64')
    p1 = np.log(p1) / diversity
    e_p1 = np.exp(p1)
    s = np.sum(e_p1)
    p1 = e_p1 / s
    return np.argmax(np.random.multinomial(1, p1, 1))
```

Let's make sure we understand how this function works with an example. Let's define the probabilities of 3 outcomes (you may think of these as win-lose-draw) where the first one is happens 1/4th of the time, the second one 65% of the time and the last one only 10% of the time.

```
probs = [0.25, 0.65, 0.1]
```

Drawing samples from this probability distribution we would expect to pull out 1 about 55% of the time and so on.

Let's sample 100 times:

```
draws = [sample(probs) for i in range(100)]
```

and let's use a `Counter` to count how many of each we drew:

```
from collections import Counter
```

```
Counter(draws)
```

```
Counter({0: 29, 1: 60, 2: 11})
```

As you can see our results reflect the actual probabilities, with some statistical fluctuations.

Great! Now that we can sample from the vocabulary, let's generate a few names. We will start from an input seed of three letters and then iterate in a loop the following steps:

- Use the seed to predict the probability distribution for next characters.
- Sample the distribution using the `sample` function.
- Append the next character to the seed.
- Shift the input window by one to include the last character appended.
- Repeat.

The loop ends either when a termination character is reached or when a pre-defined length is reached.

Let's go ahead and build this function step by step. Let's set up the `seed` of our name to be something like `ali`.

```
seed = 'ali'  
out = seed
```

In order to build the name, let's create an output list (we'll call `x`) to store our output, setting the length to that of the maximum length of the name we want to generate:

```
x = np.zeros((1, maxlen), dtype=int)
```

Let's use a variable we'll call `stop` to stop the loop if our network predicts the '`\n`' character as the next character and set it to `False`:

```
stop = False
```

Finally let's loop until we have to stop:

```
while not stop:  
    for i, c in enumerate(out[-maxlen:]):  
        x[0, i] = char_to_inds[c]  
  
    preds = model.predict(x, verbose=0)[0]  
  
    c = inds_to_char[sample(preds)]  
    out += c  
  
    if c == '\n':  
        stop = True  
out
```

```
'alin\n'
```

The network produced a few characters and then stopped. Now let's wrap these steps in a function that encapsulates this entire process in a single method. Let's call our function `complete_name`. This function will take an input `seed` of three letters and run through the previous steps to predict the next character.

```
def complete_name(seed, maxlen=3, max_name_len=None, diversity=1.0):
    """
    Completes a name until a termination character is predicted
    or max_name_len is reached.

    Parameters
    -----
    seed : string
        The start of the name to sample
    maxlen : int, default 3
        The size of the model's input
    max_name_len : int, default None
        The maximum name length; if None then samples
        are generated until the model generates a '.' character
    diversity : float, default 1.0
        Parameter to increase or decrease the randomness of the
        samples; higher = more random, lower = more deterministic

    Returns
    -----
    out : string
    """

    out = seed

    x = np.zeros((1, maxlen), dtype=int)

    stop = False

    while not stop:
        for i, c in enumerate(out[-maxlen:]):
            x[0, i] = char_to_inds[c]

        preds = model.predict(x, verbose=0)[0]

        c = inds_to_char[sample(preds, diversity)]
        out += c

        if c == '\n':
            stop = True
        else:
            if max_name_len is not None:
                if len(out) > max_name_len - 1:
                    stop = True

    return out
```

Nice! Now that we have a function to complete names, let's predict a few names that start as `jen`:

```
for i in range(10):
    print(complete_name('jen'))
```

jendonsamo

jenl

jenric

jene

jeni

...

jeni

Not bad! Let's play with the *diversity* parameter to understand what it does. If we set the diversity to be high, we get random sequences of characters:

```
for i in range(10):
    print(complete_name('jen', diversity=10, max_name_len=20))
```

jenrtypoy

jenoddrzuncuowbgfliy

jenclmefliwmyiflmzza

jeneanwzuawnuvi

jencarxtzy

jens-cukihnst

...

jens-cukihnst

If we set it to a small value, the function becomes deterministic.

TIP: since the `sample` function involves logarithms and exponential, it accumulates numerical errors very quickly. It would be better to build a model that predicts logits instead of probabilities, but keras does

not allow to do that.

```
for i in range(10):
    print(complete_name('jen', diversity=0.01, max_name_len=20))
```

```
jen
jen
jen
jen
jen
...
jen
```

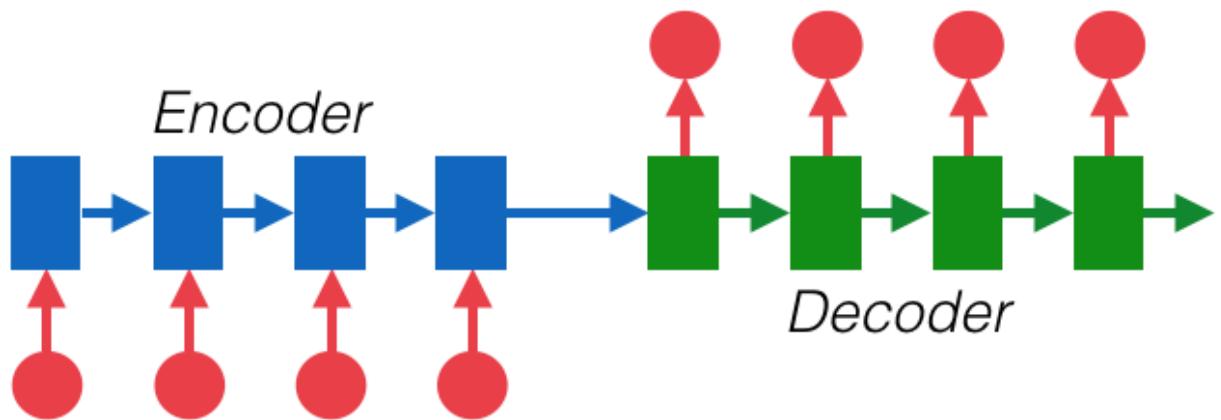
Awesome! We now know how to build a language model! Go ahead and unleash your powers on your author of choice and start producing new poems or stories. The model we built has a memory of 3 characters, so it won't exactly be "Shakespeare" when it tries to produce sentences. In order to have a model producing correct sentences in English we would need to train it with a much larger corpus and with longer windows of text. For example, a memory of 20-25 characters is long enough to generate English-looking text.

In the next section we will extend our skills to build a language translation model.

Sequence to sequence models and language translation

Sequence-to-sequence (Seq2Seq) models take a sentence in input and return a new sequence in output. They are very common in language translation, where the input sequence is a sentence in the first language and the output sequence is the translation in the second language.

Questo libro è fantastico



This book is great!

There is a [great article](#) by Francois Chollet on the Keras Blog on how to build them in keras. We strongly encourage you to read it!

In this chapter we have approached text problems from a variety of angles and hopefully inspired you to dig deeper into this domain.

EXERCISES

Exercise 1

For our Spam detection model we used a `CountVectorizer` with a vocabulary size of 3000. Was this the best size? Let's find out:

- reload the spam dataset
- do a train test split with `random_state=0` on the sms dataframe
- write a function `train_for_vocab_size` that takes `vocab_size` as input and does the following:
 - initialize a `CountVectorizer` with `max_features=vocab_size`
 - fit the vectorizer on the training messages
 - transform both the training and the test messages to count matrices
 - train the model on the training set
 - return the model accuracy on the training and test set
- plot the behavior of the train and test set accuracies as a function of `vocab_size` for a range of different vocab sizes

Exercise 2

Keras provides a large dataset of movie reviews extracted from the [Internet Movie Database](#) for sentiment analysis purposes. This dataset is much larger than the one we have used, and its already encoded as sequences of integers. Let's put what we have learned to good use and build a sentiment classifier for movie reviews:

- decide what size of vocabulary you are going to use and set the `vocab_size` variable
- import the `imdb` module from `keras.datasets`
- load the train and test sets using `num_words=vocab_size`
- check the data you have just loaded, they should be sequences of integers
- pad the sequences to a fix length of your choice. You will need to:
 - decide what is a reasonable length to express a movie review
 - decide if you are going to truncate the beginning or the end of reviews that are longer than such length

- decide if you are going to pad with zeros at the beginning or at the end for reviews that are shorter than such length
- build a model to do sentiment analysis on the truncated sequences
- train the model on the training set
- evaluate the performance of the model on the test set

Bonus points: can you convert back the sentences to their original text form? You should look at `imdb.get_word_index()` to download the word index:

Chapter 9: Training with GPUs

In this chapter, we will learn how to leverage Graphical Processing Units (GPUs) to speed up training of our models. If a model trains faster, we can do more experiments and therefore arrive to good solutions more quickly. Also, leveraging cloud GPUs has become so easy by now that it would be a pity not to take advantage of this opportunity. Only a few years ago, training a deep neural network using a GPU was a skill that demanded very sophisticated knowledge and lot of money. Nowadays, we train a model on many GPUs at a relatively affordable cost.

We will start this chapter by introducing what a GPU is, where it can be found, what kinds of GPUs are available and why they are so useful to do deep learning. Then we will review several cloud providers of GPUs and guide you through how to use them. Once we have a working cloud instance with one or more GPUs we will compare training a model with and without a GPU, to appreciate the speedup especially with Convolutional Neural Networks. We will then extend training to multiple GPUs and introduce a few ways to use multiple GPUs in Keras.

This chapter is a bit different from the other chapters as there will be less Python code and more links to external documentation and services. Also, while we will do our best to have the most up to date guide to currently existing providers, it is important that you understand how fast the landscape is evolving. During the course of the past 6 months each of the providers presented introduced newer and easier ways to access cloud GPUs, making the previous documentation obsolete. Thus, it is important that you understand the principles of why accelerated hardware helps and when. If you do this, it will be easy to adapt to new ways of doing things when they come out. All that said, let's get started!

GRAPHICAL PROCESSING UNITS

Graphical Processing Units are computer chips that specialize in the parallel manipulation of very large, multi-dimensional arrays. Originally developed to accelerate the display of video games graphics, they are today widely used for other purposes like machine learning acceleration.

The term GPU became popular in 1999, when Nvidia - still a major player in the field today - marketed the GeForce 256 as "the world's first GPU". In 2002, ATI Technologies, a competitor of Nvidia, coined the term "visual processing unit" or VPU with the release of the *Radeon 9700*. The following picture shows the original GeForce 256 (left side) and the GeForce GTX 1080 (right), one of the latest released and most powerful graphic cards in the market.



In 2006, Nvidia came out with a high level language called **CUDA** (Compute Unified Device Architecture), that helps software developers and engineers to write programs from graphic processors in a high level language – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). CUDA is a language that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. This was probably one of the most significant changes in the way researchers and developers interacted with GPUs.

But why are GPUs, originally developed for video games graphics, so useful for deep learning?

As you already know, training a neural network requires several operations, many of which involve large matrix multiplications. They perform matrix multiplications in the forward pass, when inputs (or activations) and weights are multiplied (see [Chapter 5](#) if you need a refresher on the math). Matrix multiplications are also

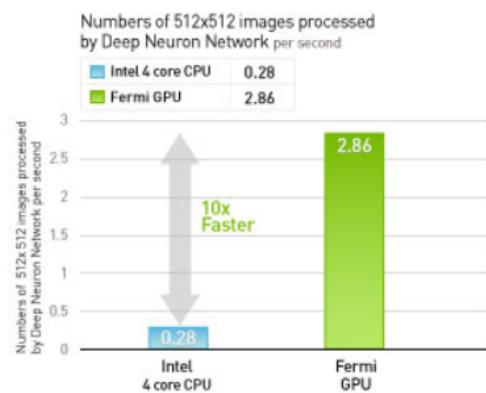
performed during back-propagation, when the error is propagated back through the network to adjust the values of the weights. In practice, training a Neural Network mostly consists of matrix multiplications. Consider for example VGG16 (a frequently used convolutional neural network for image classification. Proposed by K. Simonyan and A. Zisserman), it has approximately 140 million parameters. Using a CPU, it would take weeks to train this model and perform all the matrix multiplications.

GPUs allow to dramatically decrease the time needed for matrix multiplication, offering 10 to 100 times more computational power than traditional CPUs. There are several reasons why they make this computational speed-up possible, well discussed in [this article](#).

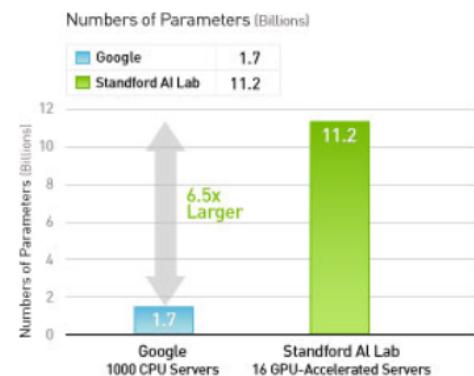
Summarizing the article, GPUs, comprised of thousands of cores unlike CPUs, not only allow for parallel operations, but they are ideal when it comes to fetching very large amounts of memory. The best GPUs can fetch up to 750GB/s, which is huge if compared with the best CPU which can handle only up to 50GB/s memory bandwidth.

Of course, dedicated GPUs, specifically designed for High Performance Computing and Deep Learning, are more performant (and expensive) than gaming GPUs, but the latter, usually available in everyday laptops, are still a good starting option!

The following picture shows a comparison between CPU and GPU performance (source: Nvidia). The left image shows that, using the same Neural Network for image detection, the *Fermi GPU* is able to process more than 10 times the number of images processed (per second) by an *Intel 4 core CPU*. The right image shows that a *16 GPU Accelerated Servers* is able to handle a more than 6 times bigger Neural Network, if compared with a *1000 CPU Servers*.



10x SPEED-UP ON IMAGE DETECTION USING NEURAL NETWORKS
Dr. Dan Ciresan, Swiss Al Lab IDSIA, Switzerland



WORLD'S LARGEST ARTIFICIAL NEURAL NETWORKS WITH GF
Adam Coates et al Sanford Al Lab, U.S.A. - [LEARN MORE](#)

CLOUD GPU PROVIDERS

As of early 2018, all major cloud providers give access to cloud instances with GPUs. The two leaders in the space are [Amazon Web Services \(AWS\)](#) and [Google Cloud Platform \(GCP\)](#). These two companies have been pioneers in providing cloud GPUs at affordable rates and they keep adding new options to their offer. Besides, they both offer additional services specifically built to optimize and serve deep learning models at scale.

Other companies offering cloud GPUs are [Microsoft Azure Cloud](#) and [IBM](#). Also, a few startups have started to offer Deep Learning optimized cloud instances, that are often cheaper and easier to access. In this chapter we will review [Floydhub](#), [Pipeline.ai](#) and [Paperspace](#).

Regardless of the cloud provider, if you have a Linux box with an NVIDIA GPU it is not hard to equip it to run tensorflow-gpu and a Jupyter Notebook.

Google Colab

The easiest way to give GPU acceleration a try is to use [Google Colab](#), also known as Colaboratory. Besides being so easy, Colab is also free to use (you only need a Google account), which makes it perfect to try out GPU acceleration.

Colaboratory is a research tool for machine learning education and research. It's a Jupyter notebook environment that requires no setup to use: you can create and share Jupyter notebooks with others without having to download, install, or run anything on your own computer other than a browser. It works with most major browsers, and is most thoroughly tested with desktop versions of Chrome and Firefox.

This [welcome notebook](#) provides you with the fundamental information to start working with Colab. In addition to all the classic operations in Jupyter you can change the notebook settings to enable GPU support:

Notebook settings

Runtime type

Python 3

Hardware accelerator

GPU

Omit code cell output when saving this notebook

CANCEL SAVE

Once you've done that, you can run this code to verify that GPU is available

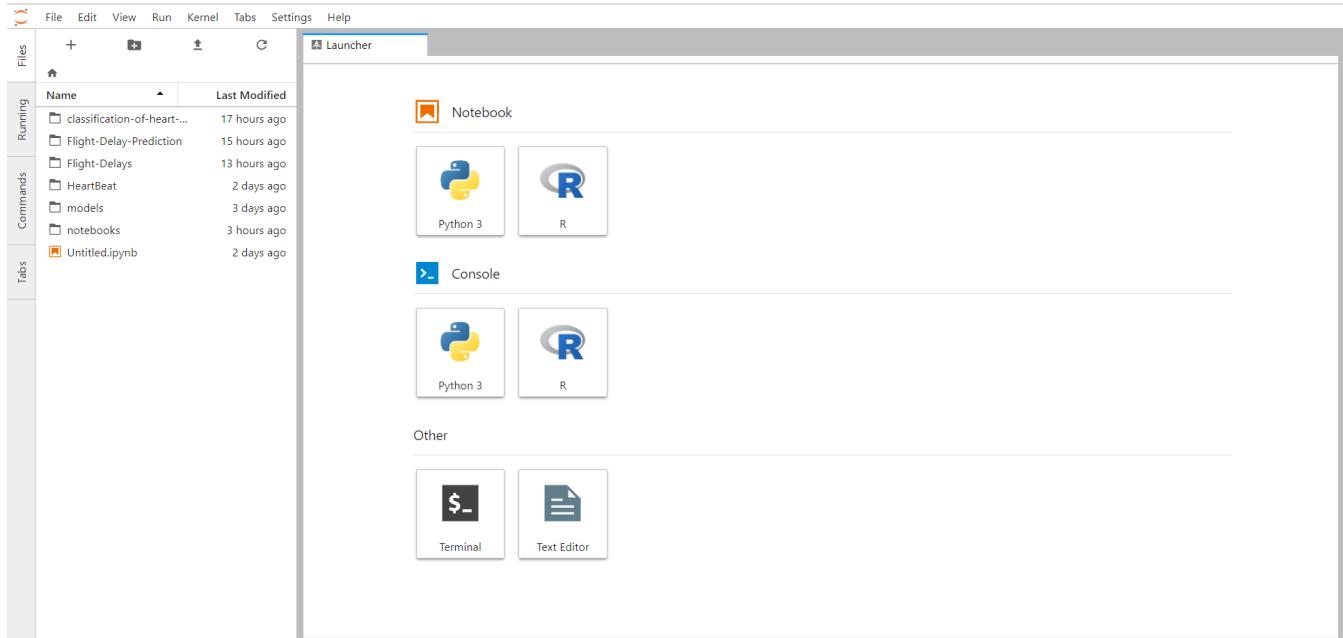
```
import tensorflow as tf  
tf.test.gpu_device_name()
```

Pipeline AI

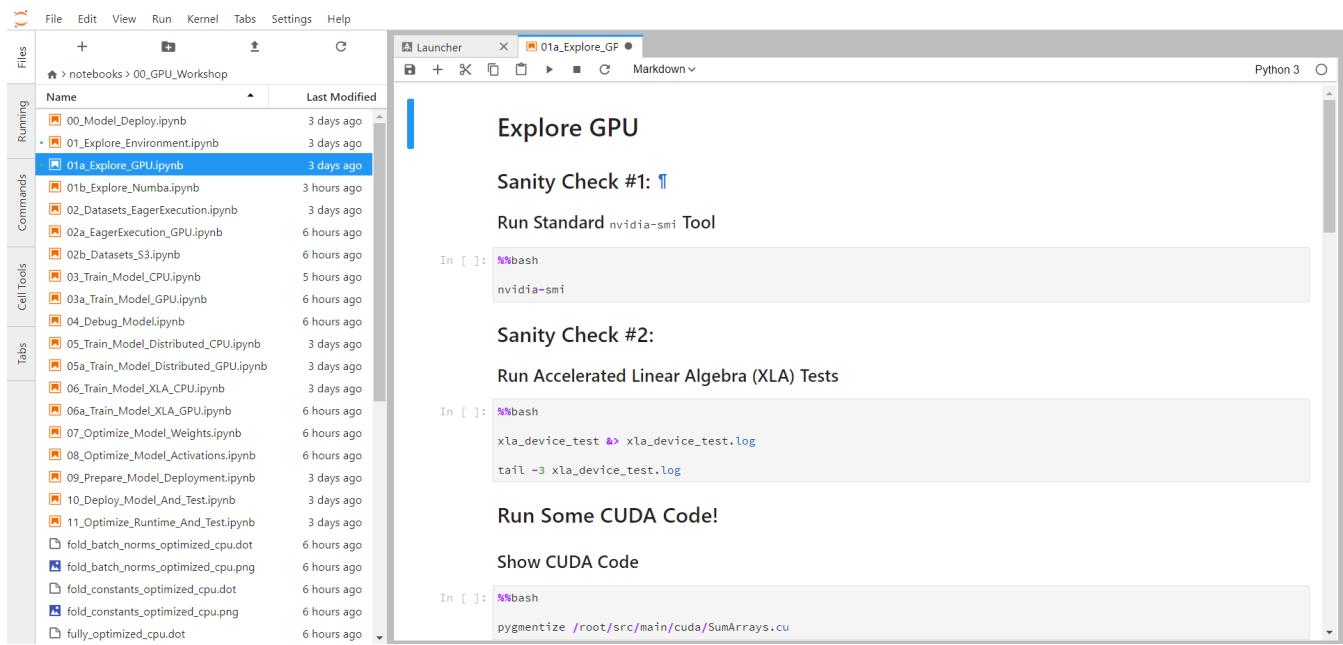
The next best option to try a GPU for free in the cloud is the service offered by [PipelineAI](#). PipelineAI service enables data scientists to rapidly train, test, optimize, deploy, and scale models in production directly from a Jupyter Notebook or command-line interface. It provides you a platform that simplifies the workflow and let the user to focus only on the essential Machine Learning aspects.

The login process to use PipelineAI is quite simple and straightforward:

1. Sign up at [PipelineAI](#).
2. Once your are successfully logged, you should see the following dashboard. You can either launch a new notebook or directly type commands in a terminal.



3. Alternatively, you can use some of the already available resources, accessible from the left menu. For example, you can have a look at the `01a_Explore_GPU.ipynb` notebook, under `notebooks > 00_GPU_Workshop`



PipelineAI is not only a platform providing GPU-powered Jupyter Notebooks, it allows you to do much more, such as monitoring the training of the algorithms, evaluating the results of your model, comparing the performances of different models, browsing among stored models, etc.. The following picture shows some of

the available tools, but have a look of all the options available in the community edition.

PipelineDB Projects

Project	Metrics	min	max	average	Models
mnist_gridsearch_cross_validation	multiclass	0.938	1	0.98	60 models
basic_workflow	accuracy	0.8	0.9	0.85	2 models

To better understand the potential of PipelineAI, we encourage you to take [this tour](#). Pipeline is under active development. You can follow its [Github repository](#)

Floydhub

Floydhub is an other easy and cheap option to access GPU in the cloud.

Floydhub is a platform for training and deploying deep learning and AI applications. FloydHub comes with fully configured CPU and GPU environments ready to use for deep learning. It includes CUDA, cuDNN and popular frameworks like Tensorflow, PyTorch, and Keras. Take a look at the [documentation](#) for a more extended explanation of its features.

This tutorial explains how to start a Jupyter Notebook on Floydhub:

1. Create an account on Floydhub.
2. Install `floyd-cli` on your computer.

```
pip install -U floyd-cli
```

3. Create a project, named for example `my_jupyter_project` :

The screenshot shows the FloydHub web interface for creating a new project. At the top, there's a navigation bar with tabs for 'Jobs', 'Projects', 'Datasets', and 'Explore'. A large blue header bar contains the text 'Create a new project' and a sub-instruction 'A project contains all your jobs and data'. Below this, there are two main input fields: 'Project path' containing 'https://www.floydhub.com/mckay/projects/' and 'Project name' containing 'my_jupyter_project'. There's also a 'Description (optional)' field with the placeholder 'My first Jupyter Notebook on FloydHub!'. Under 'Visibility Level', the 'Public' option is selected, with the note 'Anyone can see this project'. The 'Private' option is also shown with the note 'Only you can see this project'. At the bottom left is a green 'Create project' button, and at the bottom right is a blue circular icon with a white square and a checkmark.

4. From your terminal, use `floyd-cli` to initialize the project (be sure to use the name you gave the project in step 3).

```
floyd init my_jupyter_project
```

TIP: if this is the first time you run `floyd` it will ask you to login. Just type `floyd login` and follow the instructions provided.

5. Use again `floyd-cli` to kick off your first Jupyter Notebook.

```
floyd run --gpu --mode jupyter
```

This will confirm the job:

```
Creating project run. Total upload size: 224.08
Syncing code ...
[=====] 1008/1008 - 00:00:01

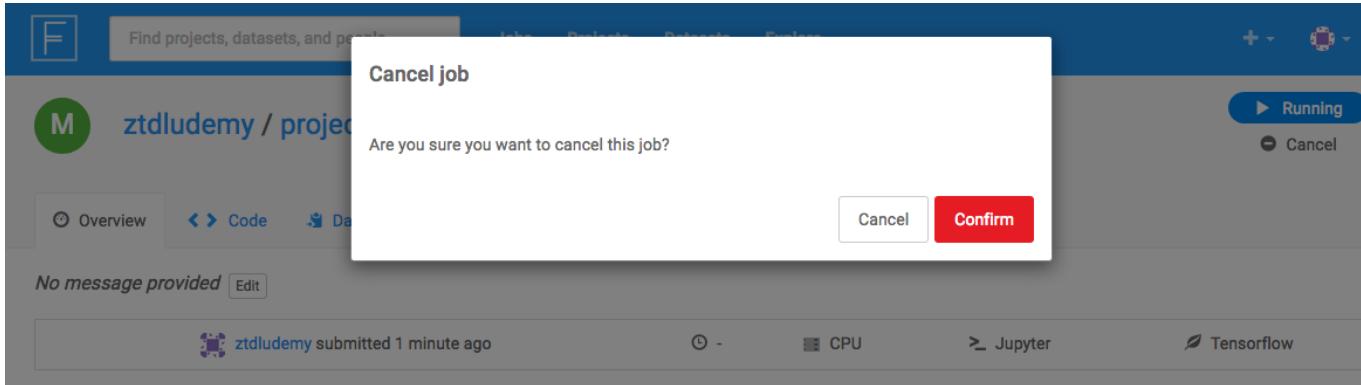
JOB NAME
-----
ztdludemmy/projects/my_jupyter_project/1

URL to job: https://www.floydhub.com/ztdludemmy/projects/my_jupyter_project/1
```

and open your FloydHub web page. Here you'll see a `View` button that will direct you to a Jupyter Notebook. The notebook is running on FloydHub's GPU servers.

The screenshot shows the FloydHub project details page for 'ztdludemmy / projects / my_jupyter_project / 1'. The top navigation bar includes 'Overview', 'Code', 'Data', 'Output', 'CLI', and 'Settings'. Below the navigation, it says 'No message provided' and shows the command 'floyd run --cpu --mode jupyter'. Under the 'Jupyter Notebook' section, there is a 'View' button which is highlighted with a red box. The 'Jupyter' interface shows tabs for 'Files', 'Running', and 'Clusters', and a message stating 'Notebook list empty.'

Once you're finished with your work you can stop the jupyter notebook with the cancel button. Make sure to save your results by downloading the notebook before you terminate it:



Paperspace

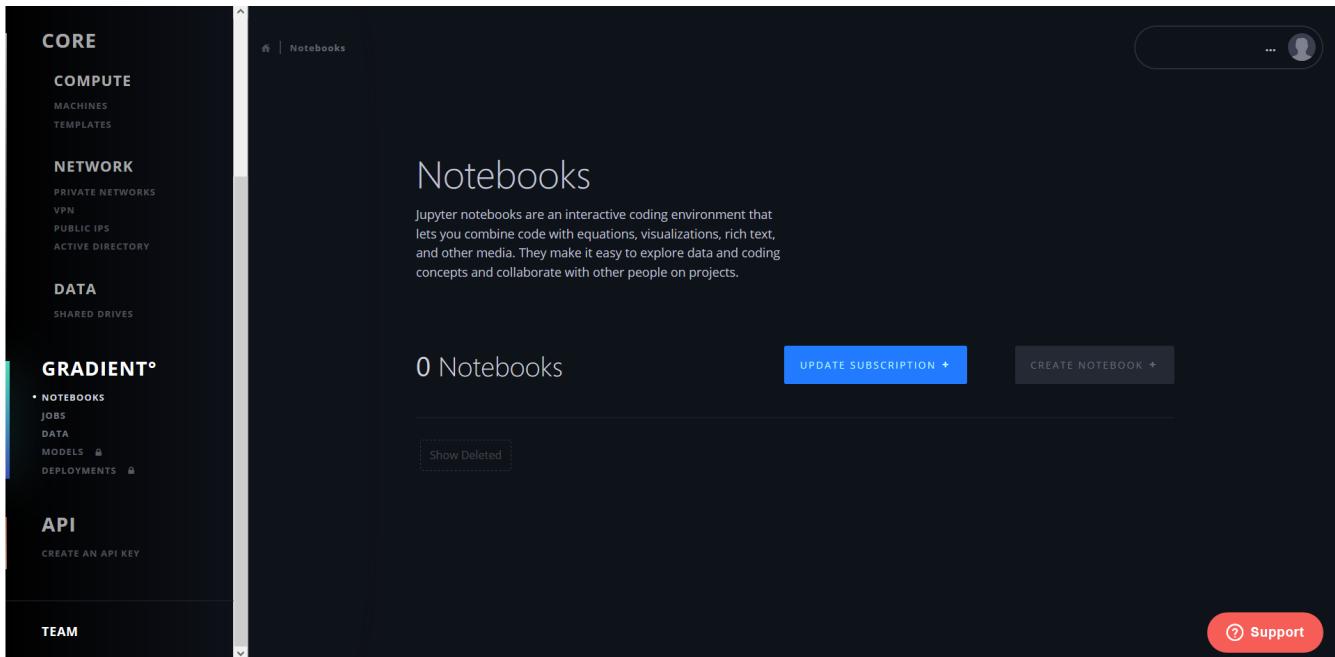
Paperspace is a platform to access a virtual desktop in the cloud. In particular, the Gradient service allows to explore, collaborate, share code and data using Jupyter Notebooks, and submit tasks to the Paperspace GPU cloud.

It is a suite of tools specifically designed to accelerate cloud AI and machine learning. Gradient also includes a powerful job runner (that can even run on the new Google TPUs!), first-class support for containers and Jupyter notebooks, and a new set of language integrations. Gradient has also a job runner, that allows you to work on your local machine and submit "jobs" to the cloud to be processed. Discover more about this service reading this [blog post](#).

The procedure to run a Jupyter Notebooks within Paperspace is similar to what we have seen so far for different GPU services:

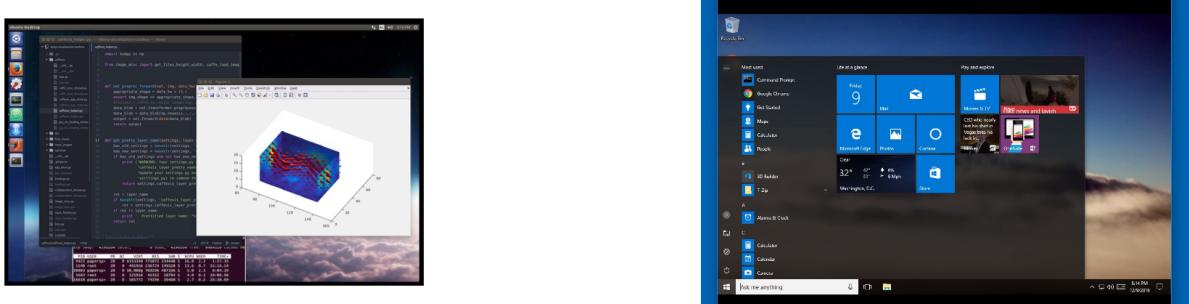
1. Create an account on Paperspace.

2. Access the console.



3. Create a Jupyter Notebook to create your models. (Credit card information on the billing page are required to enable all functionality.).

Paperspace is much more general than simply a hosted Jupyter Notebook service with GPU enabled. Since Paperspace gives you a full virtual desktop (both Linux and Windows, as shown in the following picture), you can install any other applications you need, from 3D rendering software to video editing and more.



AWS EC2 Deep Learning AMI

AWS provides a Deep Learning AMI ready to use with all the NVIDIA drivers pre-installed as well as most Deep Learning frameworks and python packages. It's not free but it's sufficiently simple and versatile to use. We can quickly launch Amazon EC2 instances pre-installed with popular deep learning frameworks such as

Apache MXNet and Gluon, TensorFlow, Microsoft Cognitive Toolkit, Caffe, Caffe2, Theano, Torch, PyTorch, Chainer, and Keras to train sophisticated, custom AI models, experiment with new algorithms, or to learn new skills and techniques.

In order to use any AWS service, we need to [open an account](#). Several resources are available for a free trial period, as described in the [official web page](#). After finishing the trial period, keep in mind that the service will charge you. Also, keep in mind that GPU instances are not included in the free tier so you will incur in charges if you complete the next steps.

Follow this procedure to spin up a GPU enabled machine on AWS with the Deep Learning AMI:

1. Access the AWS console and select EC2 from the *Compute* menu.

The screenshot shows the AWS console interface. At the top, there is a navigation bar with the AWS logo, a 'Services' dropdown, a 'Resource Groups' dropdown, and a search bar. To the right of the search bar are icons for notifications, user profile ('Mosconi Francesco'), location ('Ohio'), and support. Below the navigation bar is a sidebar titled 'AWS services' containing a search bar and a list of recently visited services: EC2, Support, Amazon SageMaker, IAM, and Billing. A link to 'All services' is also present. To the right of the sidebar is a 'Helpful tips' section with two items: 'Manage your costs' (with a link to start) and 'Create an organization' (with a link to start). Below these is an 'Explore AWS' section featuring six quick-start solutions: 'Launch a virtual machine' (With EC2, ~2-3 minutes), 'Build a web app' (With Elastic Beanstalk, ~6 minutes), 'Build using virtual servers' (With Lightsail, ~1-2 minutes), 'Connect an IoT device' (With AWS IoT, ~5 minutes), 'Start a development project' (With CodeStar, ~5 minutes), and 'Register a domain' (With Route 53, ~3 minutes). Each solution has a small icon to its left.

2. Click on the *Launch Instance* button.

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances (with sub-links for Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts), Images (AMIs, Bundle Tasks), Elastic Block Store (Volumes, Snapshots), Network & Security (Security Groups, Elastic IPs, Placement Groups). The main content area has tabs for Resources, Account Attributes, Additional Information, and AWS Marketplace. The Resources tab displays metrics like 0 Running Instances, 0 Elastic IPs, etc. Below it is a callout box about AWS re:Invent 2017. The Create Instance tab is active, showing a large blue 'Launch Instance' button. To its right, Service Health shows 'Service Status: US East (Ohio)' with a green status icon. Scheduled Events shows 'US East (Ohio): No events'. The bottom of the page includes a feedback link, language selection (English (US)), and a copyright notice from 2008-2018.

3. Scroll the page and select an Amazon Machine Image (AMI). The *Deep Learning AMI* is a good option to start. It comes in 2 flavors: Ubuntu and Amazon Linux. Both are good and we recommend you use the flavor you are more comfortable with. Also note that there are both a *Deep Learning AMI* and a *Deep Learning AMI Basic*. The *Basic AMI* has only GPU drivers installed but no Deep Learning software. The full AMI comes pre-packaged with a ton of useful packages including Tensorflow, Keras, Pytorch, MXNet, CNTK and more. We recommend you use this one to start.

AWS Services Resource Groups Mosconi Francesco Ohio Support

1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 1: Choose an Amazon Machine Image (AMI)

[Cancel and Exit](#)

 Deep Learning AMI (Ubuntu) Version 7.0 - ami-1be7d77e	Select
Comes with latest binaries of deep learning frameworks pre-installed in separate virtual environments: MXNet, TensorFlow, Caffe, Caffe2, PyTorch, Keras, Chainer, Theano and CNTK. Fully-configured with NVidia CUDA, cuDNN and NCCL as well as Intel MKL-DNN	
Root device type: ebs Virtualization type: hvm	
 Deep Learning AMI (Amazon Linux) Version 7.0 - ami-e0f7c785	Select
Comes with latest binaries of deep learning frameworks pre-installed in separate virtual environments: MXNet, TensorFlow, Caffe, Caffe2, PyTorch, Keras, Chainer, Theano and CNTK. Fully-configured with NVidia CUDA, cuDNN and NCCL as well as Intel MKL-DNN	
Root device type: ebs Virtualization type: hvm	
 Deep Learning Base AMI (Ubuntu) Version 4.0 - ami-10457475	Select
Comes with foundational platform of NVidia CUDA, cuDNN, NCCL, GPU Drivers, Intel MKL-DNN and other system libraries to deploy your own custom deep learning environment	
Root device type: ebs Virtualization type: hvm	

[Feedback](#) [English \(US\)](#) © 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved. [Privacy Policy](#) [Terms of Use](#)

4. Choose an instance type from the menu. Roughly speaking, instance types are ordered in ascending order considering the computational power and the storage space.

AWS Services Resource Groups Mosconi Francesco Ohio Support

1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 2: Choose an Instance Type

Compute optimized	c4.8xlarge	36	60	EBS only	Yes	10 Gigabit	Yes
GPU graphics	g3.4xlarge	16	122	EBS only	Yes	Up to 10 Gigabit	Yes
GPU graphics	g3.8xlarge	32	244	EBS only	Yes	10 Gigabit	Yes
GPU graphics	g3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes
GPU compute	p2.xlarge	4	61	EBS only	Yes	High	Yes
GPU compute	p2.8xlarge	32	488	EBS only	Yes	10 Gigabit	Yes
GPU compute	p2.16xlarge	64	732	EBS only	Yes	25 Gigabit	Yes
GPU compute	p3.2xlarge	8	61	EBS only	Yes	Up to 10 Gigabit	Yes
GPU compute	p3.8xlarge	32	244	EBS only	Yes	10 Gigabit	Yes
GPU compute	p3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes

[Cancel](#) [Previous](#) [Review and Launch](#) [Next: Configure Instance Details](#)

Here's a summary table of AWS GPU instances. Read the documentation for a detailed description of every instance type.

Once you have chosen the instance go through the other steps:

- Step 3: Configure Instance Details
- Step 4: Add Storage
- Step 5: Add Tags
- Step 6: Configure Security Group: make sure to leave port 22 open for SSH
- Step 7: Review Instance Launch

and finally launch your instance with a key pair you own. Let's assume it's called `your-key.pem`.

You should now be able to see the newly created instance in the dashboard, and you are now ready to connect with it.

Finally, take a look at the [Tutorials and Examples](#) section to better understand how to use Deep Learning AMI service offered by AWS.

Connect to AMI (Linux)

Once your Instance state is `running` you are ready to connect to it. We are going to do that from a terminal. We will use the ssh key we have generated and we will also route remote port 8888 to the local port 8888 so that we get to access jupyter notebook. Go ahead and type:

```
ssh -i your-key.pem -L 8888:localhost:8888 ubuntu@<your-ip>
```

TIP: if you get a message that says your key is not protected, you need to change the permissions of your key to read-only. You can do that by executing the command: `chmod 600 your-key.pem`.

Once you're connected you should see a screen like the following, where all the environments are listed:

```
=====
|   _|_ ) Deep Learning AMI (Ubuntu)
|  \_|_|
=====

Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-1054-aws x86_64)

Please use one of the following commands to start the required environment with the framework of your choice:
for MXNet(+Keras1) with Python3 (CUDA 9/MKL) _____ source activate mxnet_p36
for MXNet(+Keras1) with Python2 (CUDA 9/MKL) _____ source activate mxnet_p27
for TensorFlow(+Keras2) with Python3 (CUDA 9/MKL) _____ source activate tensorflow_p36
for TensorFlow(+Keras2) with Python2 (CUDA 9/MKL) _____ source activate tensorflow_p27
for Theano(+Keras2) with Python3 (CUDA 9) _____ source activate theano_p36
for Theano(+Keras2) with Python2 (CUDA 9) _____ source activate theano_p27
for PyTorch with Python3 (CUDA 9) _____ source activate pytorch_p36
for PyTorch with Python2 (CUDA 9) _____ source activate pytorch_p27
for CNTK(+Keras2) with Python3 (CUDA 9) _____ source activate cntk_p36
for CNTK(+Keras2) with Python2 (CUDA 9) _____ source activate cntk_p27
for Caffe2 with Python2 (CUDA 9) _____ source activate caffe2_p27
for Caffe with Python2 (CUDA 8) _____ source activate caffe_p27
for Caffe with Python3 (CUDA 8) _____ source activate caffe_p35
for Chainer with Python2 (CUDA 9) _____ source activate chainer_p27
for Chainer with Python3 (CUDA 9) _____ source activate chainer_p36
for base Python2 (CUDA 9) _____ source activate python2
for base Python3 (CUDA 9) _____ source activate python3

Official Conda User Guide: https://conda.io/docs/user-guide/index.html
AWS Deep Learning AMI Homepage: https://aws.amazon.com/machine-learning/amis/
Developer Guide and Release Notes: https://docs.aws.amazon.com/dlami/latest/devguide/what-is-dlami.html
Support: https://forums.aws.amazon.com/forum.jspa?forumID=263

* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

6 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ubuntu@ip-172-31-41-230:~$ []
```

We will go ahead and activate the tensorflow_py36 environment with the command:

```
source activate tensorflow_p36
```

and launch jupyter notebook with:

```
nohup jupyter notebook --no-browser &
```

This command launches jupyter in a way that will not stop if you disconnect from the instance. The final step is to retrieve the jupyter address: <http://localhost:8888/?token=<your-token>>. You will find it in the nohup.out file:

```
tail nohup.out
```

Copy it and and paste it into your browser. If you've done everything correctly you should see a screen like this one:

The screenshot shows a Jupyter Notebook interface. At the top, there's a navigation bar with tabs for 'Files', 'Running', 'Clusters', and 'Conda'. On the right side of the header, there's a 'Logout' button. Below the header, a message says 'Select items to perform actions on them.' There are buttons for 'Upload', 'New', and a refresh icon. The main area displays a file list with the following entries:

	Name	Last Modified
<input type="checkbox"/>	0	5 days ago
<input type="checkbox"/>	anaconda3	5 days ago
<input type="checkbox"/>	src	5 days ago
<input type="checkbox"/>	tutorials	5 days ago
<input type="checkbox"/>	nohup.out	seconds ago
<input type="checkbox"/>	Nvidia_Cloud_EULA.pdf	7 days ago

TIP: Aws also has a tutorial here: <https://docs.aws.amazon.com/dlami/latest/devguide/tutorials.html>

Connect to AMI (Windows)

To connect with the AWS EC2 Deep Learning AMI from Windows similar steps must be followed, but in this case it is convenient to use PuTTY, an SSH client specifically developed for the Windows platform.

After the installation of Putty in your machine, the procedure to connect with the cloud instance is as follows:

1. In the *Session* palette:

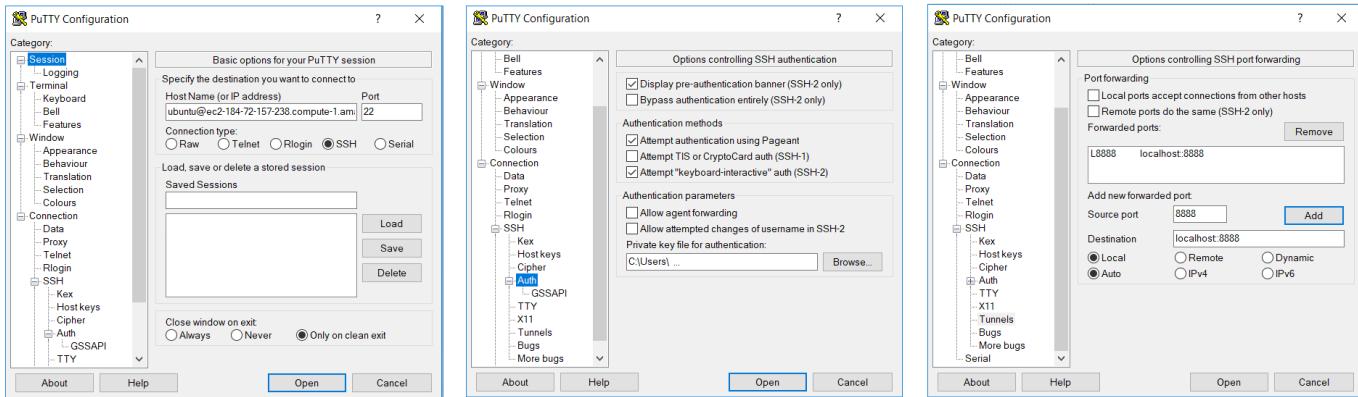
- Host Name (or IP address): `ubuntu@<your-ip>`
- Port: 22
- Connection type: SSH

2. In the *Connection > SSH > Auth* palette:

- Private key file for authentication: browse the key generated by PuttyGen

3. In the Connection > SSH > Tunnels palette:

- Source port: 8888
- Destination: localhost:8888



Once you are connected follow the same steps as for the linux case.

Turning off the instance

It is really important that once you are done with your experiments you turn off the instance in order to avoid useless costs. Just go to your AWS console and either *Stop* or *Terminate* the instance, by choosing an action from the *Actions* menu:

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with links like EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Images, AMIs, and Bundle Tasks. The Instances section is currently selected. In the main area, there's a search bar and a table with columns for Name and Instance ID. A row for an instance named 'Book' with ID i-07db3a7b63ac50626 is selected. An 'Actions' dropdown menu is open over this instance, showing options: Connect, Get Windows Password, Launch More Like This, Instance State (which is highlighted), Instance Settings, Image, Networking, and CloudWatch Monitoring. Below the table, there's a detailed view for the selected instance, showing its Public DNS, Instance ID, Instance state (running), and Public DNS (IPv4) and IPv4 Public IP.

AWS Command Line Interface

AWS also supports a command line interface [AWS CLI](#) that allows to perform the same operations from the terminal. If you'd like to try it you can install it using the command:

```
pip install awscli
```

from your terminal. Once you have installed it, you need to add configuration credentials. First you'll have to setup an IAM user in the EC2 dashboard, then run the following configuration command:

```
aws configure
```

That will prompt you to insert some information:

```
AWS Access Key ID [None]: <your access key>
AWS Secret Access Key [None]: <your secret>
Default region name [None]: us-east-1
Default output format [None]: ENTER
```

Aws regions and availability zones are:

Region Name	Region	Endpoint	Protocol
US East (Ohio)	us-east-2	rds.us-east-2.amazonaws.com	HTTPS
US East (N. Virginia)	us-east-1	rds.us-east-1.amazonaws.com	HTTPS
US West (N. California)	us-west-1	rds.us-west-1.amazonaws.com	HTTPS
US West (Oregon)	us-west-2	rds.us-west-2.amazonaws.com	HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	rds.ap-northeast-1.amazonaws.com	HTTPS
Asia Pacific (Seoul)	ap-northeast-2	rds.ap-northeast-2.amazonaws.com	HTTPS
Asia Pacific (Osaka-Local)	ap-northeast-3	rds.ap-northeast-3.amazonaws.com	HTTPS
Asia Pacific (Mumbai)	ap-south-1	rds.ap-south-1.amazonaws.com	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	rds.ap-southeast-1.amazonaws.com	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	rds.ap-southeast-2.amazonaws.com	HTTPS
Canada (Central)	ca-central-1	rds.ca-central-1.amazonaws.com	HTTPS
China (Beijing)	cn-north-1	rds.cn-north-1.amazonaws.com.cn	HTTPS
China (Ningxia)	cn-northwest-1	rds.cn-northwest-1.amazonaws.com.cn	HTTPS
EU (Frankfurt)	eu-central-1	rds.eu-central-1.amazonaws.com	HTTPS
EU (Ireland)	eu-west-1	rds.eu-west-1.amazonaws.com	HTTPS
EU (London)	eu-west-2	rds.eu-west-2.amazonaws.com	HTTPS
EU (Paris)	eu-west-3	rds.eu-west-3.amazonaws.com	HTTPS
South America (Sao Paulo)	sa-east-1	rds.sa-east-1.amazonaws.com	HTTPS

Make sure to choose a region that provides a copy of the Deep Learning AMI.

As explained in the [AWS CLI guide](#), the output format can be `json`:

```
{
  "Reservations": [
    {
      "Groups": [],
      "Instances": [
        {
          "AmiLaunchIndex": 0,
          "ImageId": "ami-916f59f4",
          "InstanceId": "i-0fcc6380969917f3a",
          "InstanceType": "t2.micro",
          "KeyName": "your-key",
          "LaunchTime": "2018-04-25T16:00:07.000Z",
          "Monitoring": {
            "State": "disabled"
          },
          "Placement": {
            "AvailabilityZone": "us-east-2c",
            "GroupName": "",
            "Tenancy": "default"
          },
          "PrivateDnsName": "ip-172-31-46-146.us-east-2.compute.internal",
          "PrivateIpAddress": "172.31.46.146",
          "ProductCodes": [],
          "PublicDnsName": "ec2-18-219-210-227.us-east-2.compute.amazonaws.com",
          "PublicIpAddress": "18.219.210.227",
          "State": {
            "Code": 16,
            "Name": "running"
          },
          "StateTransitionReason": "",
          "SubnetId": "subnet-acbb52e1",
          "VpcId": "vpc-f94096",
          "Architecture": "x86_64",
          "BlockDeviceMappings": [

```

or text:

```
INSTANCES      0      x86_64      False   True    xen    ami-916f59f4    i-0fcc6380969917f3a t
2.micro your-key      2018-04-25T16:00:07.000Z      ip-172-31-46-146.us-east-2.compute.internal 1
72.31.46.146      ec2-18-219-210-227.us-east-2.compute.amazonaws.com      18.219.210.227 /dev/sda1 e
bs      True      subnet-acbb52e1 hvm      vpc-f94096
BLOCKDEVICEMAPPINGS      /dev/sda1
EBS      2018-04-25T16:00:07.000Z      True      attached      vol-0994a246607453cbf
MONITORING      disabled
NETWORKINTERFACES      0a:71:f4:9d:fc:5e      eni-2aed5e00      165301701209      ip-172-31-46-
146.us-east-2.compute.internal 172.31.46.146      True      in-use      subnet-acbb52e1 vpc-f94096
ASSOCIATION      amazon      ec2-18-219-210-227.us-east-2.compute.amazonaws.com      18.219.210.227
ATTACHMENT      2018-04-25T16:00:07.000Z      eni-attach-b5f6635a      True      0      attached
GROUPS      sg-eff9f384      launch-wizard-1
PRIVATEIPADDRESSES      True      ip-172-31-46-146.us-east-2.compute.internal      172.31.46.146
ASSOCIATION      amazon      ec2-18-219-210-227.us-east-2.compute.amazonaws.com      18.219.210.227
PLACEMENT      us-east-2c      default
SECURITYGROUPS      sg-eff9f384      launch-wizard-1
STATE      16      running
```

Once configured you can start your Deep Learning instance with the following command:

```
aws ec2 run-instances \
--image-id <DL-AMI-ID-for-your-region> \
--count 1 \
--instance-type <instance-type> \
--key-name <your-ssh-key-name> \
--subnet-id <subnet-id> \
--security-group-ids <security-group-id> \
--tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=<a-name-tag>}]'
```

Where you will need to insert the following parameters:

- <DL-AMI-ID-for-your-region> : the AMI ID for the Deep Learning AMI in the AWS region you've chosen
- <instance-type> : the type of instance, like g2.2xlarge , p3.16xlarge etc.
- <your-ssh-key-name> : then name of your ssh key. You must have this on your disk.
- <subnet-id> : the subnet id, you can find this when you launch an instance from the web interface.
- <security-group-id> : the security group id, you can find this when you launch an instance from the web interface as well.
- <a-name-tag> : a name for your instance, so that you can easily retrieve it by name

You can query the status of your launch with the command:

```
aws ec2 describe-instances
```

and remember to stop or terminate the instance when you are done, for example using this command:

```
aws ec2 terminate-instances --instance-ids <your-instance-id>
```

which would return something like this:

```
{  
    "TerminatingInstances": [  
        {  
            "CurrentState": {  
                "Code": 32,  
                "Name": "shutting-down"  
            },  
            "InstanceId": "i-0fcc6380969917f3a",  
            "PreviousState": {  
                "Code": 16,  
                "Name": "running"  
            }  
        }  
    ]  
}
```

AWS Sagemaker

AWS Sagemaker is an AWS managed solution that allows to perform all the steps involved in a deep learning pipeline. In fact, on Sagemaker you can define, train and deploy a machine learning model in just a few steps.

Sagemaker provides an integrated Jupyter notebook instance that can be used to access data stored in other AWS services, explore it, clean it and analyze it as well as to define a machine learning model. It also provides common machine learning algorithms that are optimized to run efficiently against extremely large data in a distributed environment.

Detailed information about this service can be found in the official [documentation](#).

The procedure to spin up a notebook is similar to what previously seen:

1. Create an AWS account and access the console.

2. From the AWS console, select the *Amazon SageMaker* service, under the Machine Learning group.

The screenshot shows the AWS Services dashboard. The 'Machine Learning' group is expanded, revealing services like EC2, CloudTrail, Billing, AWS Organizations, Compute, Management Tools, Mobile Services, Storage, AR & VR, Application Integration, Media Services, and Database. 'Amazon SageMaker' is highlighted in blue, indicating it is selected. On the right side, there are 'Helpful tips' for managing costs and creating organizations, and sections for 'Explore AWS' featuring Amazon RDS and Amazon Kinesis.

3. Click the button *Create notebook instance*.

The screenshot shows the Amazon SageMaker Dashboard. The left sidebar has 'Dashboard' selected. The main area is titled 'Overview' and contains four sections: 'Notebook instance', 'Jobs', 'Models', and 'Endpoint'. Each section has a description and a button: 'Create notebook instance', 'View jobs', 'View models', and 'View endpoints'. Below this is a 'Recent activity' section which is currently empty. At the bottom, there are links for 'Feedback', 'English (US)', and legal notices.

4. Assign a *Notebook instance name*, for example "my_first_notebook" and click the button *Create notebook instance*. Sagemaker offers several types of instances, including a cheap option for development of your notebook, a CPU-heavy instance if your code requires a lot of CPUs and a GPU-enabled instance if you need it. Notice that the instance types available for the notebook instance are different from the ones available for model training and deployment.

The screenshot shows the 'Create notebook instance' page in the AWS SageMaker console. On the left, there's a sidebar with options like Dashboard, Notebook instances (selected), Lifecycle configuration, Jobs, Resources, Models, Endpoint configuration, and Endpoints. The main area has a title 'Create notebook instance' and a sub-header: 'Amazon SageMaker provides pre-built fully managed notebook instances that run Jupyter notebooks. The notebook instances include example code for common model training and hosting exercises.' Below this is a 'Notebook instance settings' section with fields for 'Notebook instance name' (set to 'my-first-notebook'), 'Notebook instance type' (set to 'ml.t2.medium'), 'IAM role' (set to 'AmazonSageMaker-ExecutionRole-20180202T215962'), and 'VPC - optional' (set to 'No VPC'). At the bottom, there's a note about 'Lifecycle configuration - optional'. The footer includes links for Feedback, English (US), and a copyright notice.

5. Start working on the newly created notebook

The screenshot shows the 'Notebook instances' page in the AWS SageMaker console. A green banner at the top says 'Success! Your notebook instance is being created. Open the notebook instance when status is InService and open a template notebook to get started.' Below this is a search bar and a table with columns: Name, Instance, Creation time, Status, and Actions. The table contains one row for 'my-first-notebook' with details: ml.t2.medium, Mar 21, 2018 20:04 UTC, Pending, and a link icon. The footer includes links for Feedback, English (US), and a copyright notice.

Once you're done developing your model, Sagemaker allows to export, train and deploy the model with very easy steps. Please refer to the [User guide](#) for more information on these steps.

Google Cloud and Microsoft Azure

Although we reviewed in detail the solutions offered by Amazon AWS, both Google Cloud and Microsoft Azure offer similarly priced GPU-enabled cloud instances. We invite you check their offering here:

- Google Cloud
- Microsoft Azure

The DIY solution (on Ubuntu)

If you'd like start from scratch on a barebone linux machine with a GPU, here are the steps you will need to follow:

1. Install [NVIDIA Cuda Drivers](#). CUDA is a language that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.
- Download and install [CUDNN](#). CUDNN is an NVIDIA library built on CUDA that implements a lot of common Neural Network algorithms.
 - Install [Miniconda](#). Miniconda is a minimal installation of Python and the `conda` package manager that we will use to install other packages.
 - Install common packages `conda install pip numpy pandas scikit-learn scipy matplotlib seaborn h5py` . This command will install the packages in the `base` environment.
 - Install Tensorflow compiled with GPU support: `pip install tensorflow-gpu` .
 - (Optional) Install Keras: `pip install keras` .

GPU VS CPU TRAINING

Regardless of how you decided to get access to a GPU-enabled cloud instance, in the following code we will assume that you have access to such an instance and review some functionality that is available in Keras and Tensorflow when running on a GPU instance.

TIP: the code that follows relies on functionality that is specific to Tensorflow. This implies that we cannot change backend.

Let's start by comparing training speed on a CPU vs a GPU for a convolutional neural network. We will train this on the CIFAR10 data that we have encountered also in [Chapter 6](#). Let's load the usual packages of Numpy, Pandas and Matplotlib:

```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

Convolutional model comparison

First we load the data using a helper function that also rescales it and expands the labels to binary categories. If you're unfamiliar with these steps we recommend you review [Chapter 3](#), [Chapter 4](#) and [Chapter 6](#) where they are repeated multiple times and explained in detail.

```
from keras.datasets import cifar10
from keras.utils import to_categorical
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/__init__.py:36: Fu
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
def cifar_train_data():
    print("Loading CIFAR10 Data")
    (X_train, y_train), _ = cifar10.load_data()
    X_train = X_train.astype('float32') / 255.0
    y_train_cat = to_categorical(y_train, 10)
```

```
    return X_train, y_train_cat  
  
X_conv, y_conv = cifar_train_data()
```

Loading CIFAR10 Data

Next we define a function that creates the convolutional model. By now you should be familiar with every line of code that follows, but just as a reminder, we create a `Sequential` model adding layers in sequence, like pancakes in a stack. The layers in this network are:

- **2D Convolutional layer** with 32 filters, each of size 3x3 and ReLU activation. Notice that in the first layer we also specify the input shape of `(32, 32, 3)` which means our images are 32x32 pixels with 3 colors: RGB.
- **2D Convolutional layer** with 32 filters, each of size 3x3 and ReLU activation. We add a second convolutional layer immediately after the first to effectively convolve over larger regions in the input image.
- **Max Pooling layer** 2 D with a pool size of 2x2. This will cut in half the height and the width of our feature maps, effectively making the calculations 4 times faster.
- **Flatten layer** to go from the order 4 tensors used by convolutional layers to an order 2 tensor suitable for fully connected networks.
- **Fully connected layer** with 512 nodes and a ReLU activation
- **Output layer** with 10 nodes and a Softmax activation

If you need to review these concepts make sure to check out Chapter 6 for more details.

We also compile the model for a classification problem using the **Categorical Crossentropy** loss function and the **RMSProp** optimizer. These are explained in detail in Chapter 5.

Notice also that we import the `time` module to track the performance of our model:

```
from keras.models import Sequential  
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense  
from time import time
```

```
def convolutional_model():  
    print("Defining convolutional model")  
    t0 = time()
```

```

model = Sequential()
model.add(Conv2D(32, (3, 3),
                padding='same',
                input_shape=(32, 32, 3),
                kernel_initializer='normal',
                activation='relu'))
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='normal'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

print("{:0.3f} seconds.".format(time() - t0))

print("Compiling the model...")
t0 = time()
model.compile(loss='categorical_crossentropy',
               optimizer='rmsprop',
               metrics=['accuracy'])

print("{:0.3f} seconds.".format(time() - t0))
return model

```

Now we are ready to do a comparison between the CPU training time and the GPU training time. We can force tensorflow to create the model on the the CPU with the context setter with `tf.device('cpu:0')`. First let's import Tensorflow:

```
import tensorflow as tf
```

And then let's create a model on the CPU:

```
with tf.device('cpu:0'):
    model = convolutional_model()
```

```
Defining convolutional model
0.069 seconds.
Compiling the model...
0.039 seconds.
```

Now let's train the CPU model for 2 epochs:

```
print("Training convolutional CPU model...")
t0 = time()
model.fit(X_conv, y_conv,
           batch_size=1024,
           epochs=2,
           shuffle=True)
print("{:0} seconds.".format(time() - t0))
```

```
Training convolutional CPU model...
Epoch 1/2
50000/50000 [=====] - 31s 613us/step - loss: 1.9827 - acc: 0.296
Epoch 2/2
50000/50000 [=====] - 29s 589us/step - loss: 1.6269 - acc: 0.431
60.39647626876831 seconds.
```

Now let's compare the model with a model living on the GPU. We use a similar context setter: `with tf.device('gpu:0'):`

```
with tf.device('gpu:0'):
    model = convolutional_model()
```

```
Defining convolutional model
0.065 seconds.
Compiling the model...
0.037 seconds.
```

And then we train the model on the GPU:

```
print("Training convolutional GPU model...")
t0 = time()
model.fit(X_conv, y_conv,
           batch_size=1024,
           epochs=2,
           shuffle=True)
print("{:0.3f} seconds.".format(time() - t0))
```

```
Training convolutional GPU model...
Epoch 1/2
50000/50000 [=====] - 5s 110us/step - loss: 2.0690 - acc: 0.2739
Epoch 2/2
50000/50000 [=====] - 4s 71us/step - loss: 1.7199 - acc: 0.4012
9.319 seconds.
```

As you can see training on the GPU is much faster than on the CPU. Also notice that the second epoch runs much faster than the first one. The first epoch also includes the time to transfer the model to the GPU, while for the following ones the model has already been transferred to the GPU. Pretty cool!

NVIDIA-SMI

We can check that the GPU is actually being utilized using `nvidia-smi`. The [NVIDIA System Management Interface](#) is a tool that allows us to check the operation of our GPUs. To better understand how it works, have a look at the [documentation](#).

In order to use the NVIDIA System Management Interface:

1. Open a new terminal from the Jupyter interface



2. Type `nvidia-smi` in the command line.

MULTIPLE GPUS

If your machine has more than one GPU you can use multiple gpus to speed up your training even more. This can be done in 2 ways:

- distributing different batches to different GPUs, also called **data parallelization**.
- distributing different parts of the model to different GPUs, also called **model parallelization**.

Let's have a look at them in detail.

Data Parallelization

Keras makes it really easy to parallelize training by distributing data across multiple GPUs through the recently introduced `multi_gpu_model` command. Let's import it from `keras.utils` :

```
from keras.utils import multi_gpu_model
```

TIP: if you're on floydhub the keras version is probably earlier than the one we are using in the book. If you don't find `keras.utils.multi_gpu_model` try with

```
from keras.utils.training_utils import multi_gpu_model
```

or update keras with `pip install --upgrade keras`

Now let's create a new convolutional model (on the cpu):

```
with tf.device("/cpu:0"):  
    model = convolutional_model()
```

```
Defining convolutional model  
0.067 seconds.  
Compiling the model...  
0.038 seconds.
```

and let's distribute it over 2 GPUs (this will only work if you have at least 2 GPUs on your machine):

```
#adjust this to the number of gpus in your machine  
NGPU = 2
```

```
model = multi_gpu_model(model, NGPU)
```

Once the model has been parallelized, we need to re-compile it:

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

Finally we can train the model in the exact same way as we did before. Notice that the `multi_gpu_model` documentation explains how a batch is divided to the GPUs:

E.g. if your `batch_size` is 64 and you use `gpus=2`,
then we will divide the input into 2 sub-batches of 32 samples,
process each sub-batch on one GPU, then return the full
batch of 64 processed samples.

This also means that if we want to maximize GPU utilization we want to increase the batch size by a factor equal to the number of GPUs, so we will use `batch_size=1024*NGPU`.

```
print("Training recurrent GPU model on 2 GPUs ...")  
t0 = time()  
model.fit(X_conv, y_conv,  
          batch_size=1024*NGPU,  
          epochs=2,  
          shuffle=True)  
print("{:0.3f} seconds.".format(time() - t0))
```

```
Training recurrent GPU model on 2 GPUs ...  
Epoch 1/2  
50000/50000 [=====] - 4s 82us/step - loss: 2.1858 - acc: 0.2299  
Epoch 2/2  
50000/50000 [=====] - 2s 46us/step - loss: 1.8290 - acc: 0.3607  
6.817 seconds.
```

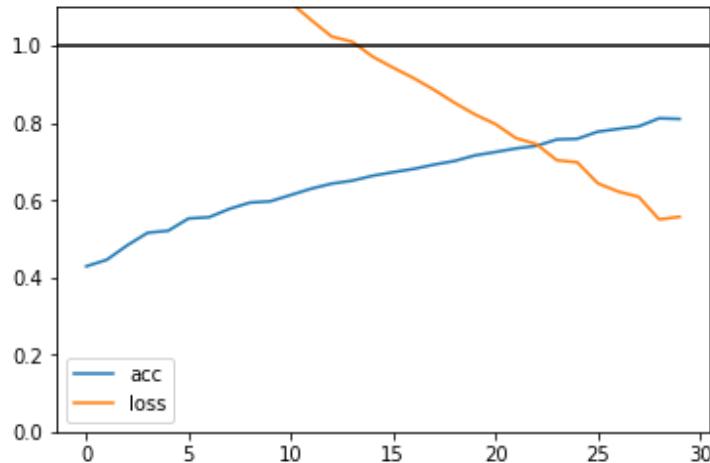
Since with 2 GPUs each epoch takes only a few seconds, let's run the training for a few more epochs:

```
h = model.fit(X_conv, y_conv,
               batch_size=1024*NGPU,
               epochs=30,
               shuffle=True,
               verbose=0)
```

and let's plot the history like we've done many times in this book:

```
pd.DataFrame(h.history).plot()
plt.ylim(0, 1.1)
plt.axhline(1, color='black')
```

```
<matplotlib.lines.Line2D at 0x7f28c805ac18>
```



As you can see, with 30 epochs the model seems to be still improving. Having multiple GPUs allowed us to iterate fast and explore the performance of a powerful convolutional model in a very short time. Cool!

CONCLUSION

In this chapter we have seen how GPUs can easily be used to train faster on larger data. Before you move on to the next chapter make sure to terminate all instances or you'll incur in charges!

EXERCISE 1

In Exercise 2 of Chapter 8 we introduced a model for sentiment analysis of the IMDB dataset provided in Keras.

- Reload that dataset and prepare it for training a model:
 - choose vocabulary size
 - pad the sequences to a fixed length
- define a function `recurrent_model(vocab_size, maxlen)` similar to the `convolutional_model` function defined earlier. The function should return a recurrent model.
- Train the model on CPU and measure the training time
- Train the model on 1 GPU and measure the training time
- Bonus points if you run it on a machine with more than 1 GPU using `multi_gpu_model`

EXERCISE 2

Model parallelism is a technique that is used for very large models that cannot fit in the memory of a single GPU. While this is not the case for the model we developed in Exercise 1, it is still possible to distribute the model across multiple GPUs using the `with context setter`. Define a new model with the following architecture:

1. Embedding

- LSTM
- LSTM
- LSTM
- Dense

Place layers 1 and 2 on the first GPU, layers 3 and 4 on the second GPU and the final Dense layer on the CPU.

Train the model and see if the performance improves.

Chapter 10: Performance Improvement

Congratulations! We've traveled very far along this deep learning journey together! We have learned about fully connected, convolutional and recurrent architectures and we applied them to a variety of problems, from image recognition to sentiment analysis.

One question we haven't really answered yet is what to do when a model is not performing well. This is very common for deep learning models. We train a model and the performance on the test set is disappointing.

This issue could be due to many reasons:

- too little data
- wrong architecture
- too little training
- wrong hyper-parameters

How do we approach debugging and improving a model?

This chapter is about a few techniques to do that. We will start by introducing **Learning Curves**, a tool that is useful to decide if more data is needed. Then we will introduce several **regularization** techniques, that may be useful to fight **Overfitting**. Some of these techniques have been invented very recently.

Finally, we will discuss **data augmentation**, which is useful in some cases, e.g. when the input data is made of images. We will conclude the chapter with a brief part on **hyperparameter optimization**. This is a vast topic, that can be approached in several ways which we'll look into.

Let's start as usual with a few imports:

```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```


LEARNING CURVES

The first tool we present is the **Learning Curve**. A learning curve plots the behavior of the training and validation scores as a function of how much training data we fed to the model.

Let's load a simple dataset and explore how to build a learning curve. We will use the digits dataset from Scikit Learn, which is quite small. First of all we import the `load_digits` function and use it:

```
from sklearn.datasets import load_digits
```

Now let's create a variable called `digits` we'll fill as the result of calling `load_digits()`:

```
digits = load_digits()
```

Then we assign `digits.data` and `digits.target` to `x` and `y` respectively:

```
x, y = digits.data, digits.target
```

Let's look at the shape of the `x` data:

```
x.shape
```

```
(1797, 64)
```

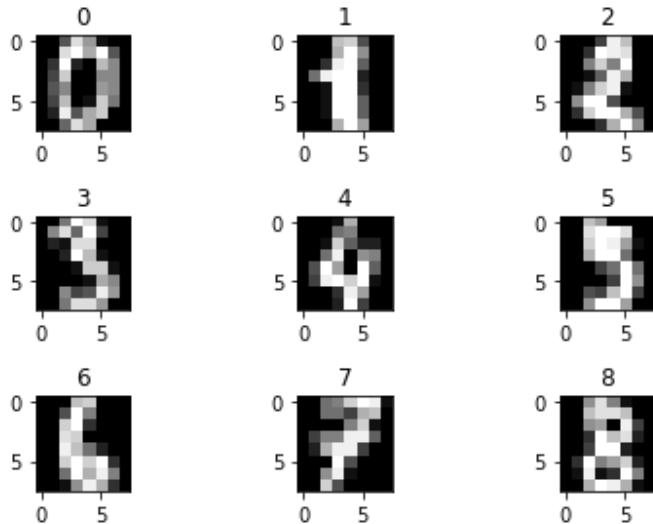
`x` is an array of 1797 images that have been unrolled as feature vectors of length 64.

```
y.shape
```

```
(1797, )
```

In order to see the images we can always reshape them to the original 8x8 format. Let's plot a few digits:

```
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(X.reshape(-1, 8, 8)[i], cmap='gray')
    plt.title(y[i])
plt.tight_layout()
```



TIP: the function `tight_layout` automatically adjusts subplot params so that the subplot(s) fits in to the figure area. See the [Documentation](#) for further details.

Since `digits` is a Scikit Learn `Bunch` object, it has a property with the description of the data (in the `DESCR` key). Let's print it out:

```
print(digits.DESCR)
```

```
Optical Recognition of Handwritten Digits Data Set
=====
Notes
-----
Data Set Characteristics:
  :Number of Instances: 5620
  :Number of Attributes: 64
  :Attribute Information: 8x8 image of integer pixels in the range 0..16.
...
  :Attribute Information: 8x8 image of integer pixels in the range 0..16.
```

From `digits.DESCR` we find that the input is made of integers in the range (0,16). Let's check that it's true by calculating the minimum and maximum values of X:

```
X.min()
```

```
0.0
```

```
X.max()
```

```
16.0
```

Let's also check the data type of `x`:

```
X.dtype
```

```
dtype('float64')
```

As previously seen in [Chapter 3](#), it's a good practice to rescale the input so that it's close to 1. Let's do this by dividing by the maximum possible value (`16.0`):

```
X_sc = X / 16.0
```

`y` contains the labels as a list of digits:

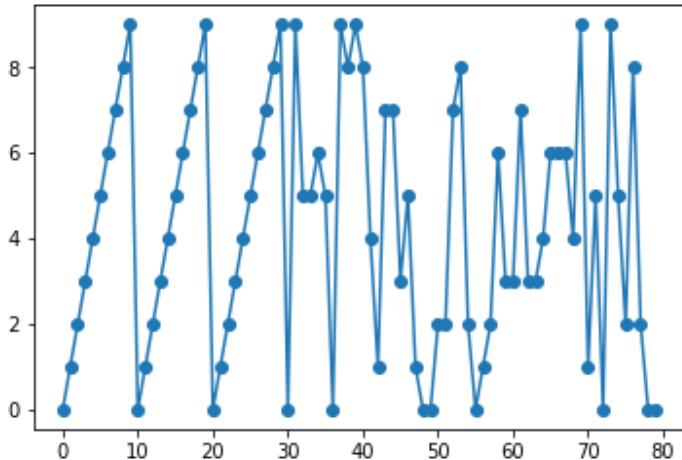
```
y[:20]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Although it could appear that the digits are sorted, actually they are not:

```
plt.plot(y[:80], 'o-')
```

```
[<matplotlib.lines.Line2D at 0x7ff9d2079780>]
```



As seen in [Chapter 3](#), let's convert them to 1-hot encoding, to substitute the categorical column with a set of boolean columns, one for each category. First, let's import the `to_categorical` method from `keras.utils`:

```
from keras.utils import to_categorical
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/_init__.py:36: Fu
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Then let's set the variable of `y_cat` to these categories:

```
y_cat = to_categorical(y, 10)
```

Now we can split the data into a training and a test set. Let's import the `train_test_split` function and call it against our data and the target categories:

```
from sklearn.model_selection import train_test_split
```

We will split the data with a 70/30 ratio and we will use a `random_state` here, so that we all get the exact same train/test split. We will also use the option `stratify`, to require the ratio of classes be balanced, i.e. about 10% for each class (we already introduced this concept in Chapter 3 for the *stratified K-fold* cross validation).

```
x_train, x_test, y_train, y_test = train_test_split(X_sc, y_cat,
                                                    test_size=0.3,
                                                    random_state=0,
                                                    stratify=y)
```

Let's double check that we have balanced the classes correctly. Since `y_test` is now a 1-hot encoded vector, we need first to recover the corresponding digits. We can do this using the function `argmax`:

```
y_test_classes = np.argmax(y_test, axis=1)
```

`y_test_classes` is an array of digits:

```
y_test_classes
```

```
array([1, 4, 5, 6, 9, 1, 2, 2, 2, 0, 7, 5, 4, 8, 6, 6, 8, 2, 0, 9, 7, 3,
       9, 1, 3, 5, 2, 2, 9, 9, 8, 9, 7, 6, 1, 3, 1, 4, 7, 6, 7, 3, 5, 0,
       1, 1, 7, 5, 4, 6, 0, 5, 8, 9, 0, 5, 4, 5, 3, 5, 5, 6, 5, 4, 9, 6,
       5, 9, 6, 5, 7, 6, 6, 3, 0, 8, 4, 4, 3, 2, 9, 7, 2, 7, 9, 8, 8, 0,
       1, 7, 2, 3, 3, 5, 5, 6, 0, 4, 3, 7, 1, 4, 1, 9, 0, 5, 3, 8, 9, 6,
       4, 9, 2, 9, 2, 0, 6, 7, 8, 1, 9, 2, 8, 6, 3, 6, 5, 1, 3, 6, 2, 3,
       0, 6, 5, 5, 9, 2, 8, 1, 0, 1, 4, 5, 1, 0, 3, 0, 0, 9, 8, 9, 2, 2,
       5, 8, 1, 9, 3, 7, 6, 8, 7, 3, 1, 2, 5, 1, 1, 6, 3, 9, 6, 9, 8, 9,
       9, 8, 9, 9, 8, 8, 4, 7, 6, 2, 6, 4, 3, 4, 4, 3, 8, 5, 4, 8, 3, 1,
       ...
       0, 0, 7, 8, 5, 2, 3, 5, 2, 6, 1, 3])
```

There are many ways to count the number of each digit, the simplest is to temporarily wrap the array in a Pandas Series and use the `.value_counts()` method:

```
pd.Series(y_test_classes).value_counts()
```

```
5    55
3    55
1    55
9    54
7    54
6    54
4    54
0    54
2    53
...
dtype: int64
```

Great! Our classes are balanced, with around 54 samples per class. Let's quickly train a model to classify these digits. First we load the necessary libraries:

```
from keras.models import Sequential
from keras.layers import Dense
```

We create a small, fully connected network with 64 inputs, a single inner layer with 16 nodes and 10 outputs with a Softmax activation function:

```
model = Sequential()
model.add(Dense(16, input_shape=(64,), activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile('adam', 'categorical_crossentropy', metrics=['accuracy'])
```

Let's also save the initial weights so that we can always re-start from the same initial configuration:

```
initial_weights = model.get_weights()
```

Now we fit the model on the training data for 100 epochs:

```
model.fit(X_train, y_train, epochs=100, verbose=1)
```

```
Epoch 1/100
1257/1257 [=====] - 0s 333us/step - loss: 2.2361 - acc: 0.1575
Epoch 2/100
```

```
1257/1257 [=====] - 0s 83us/step - loss: 2.0315 - acc: 0.3413
Epoch 3/100
1257/1257 [=====] - 0s 81us/step - loss: 1.8469 - acc: 0.4638
Epoch 4/100
1257/1257 [=====] - 0s 81us/step - loss: 1.6543 - acc: 0.5975
Epoch 5/100
...
Epoch 5/100
```

```
<keras.callbacks.History at 0x7ff97801ffd0>
```

The model converged and we can evaluate the final training performance and test accuracies:

```
_, train_acc = model.evaluate(X_train, y_train, verbose=0)
_, test_acc = model.evaluate(X_test, y_test, verbose=0)
```

TIP: The character `_` means that part of the function result can be deliberately ignored, and that variable can be thrown away.

And print them out:

```
print("Train accuracy: {:.4f}".format(train_acc))
print("Test accuracy: {:.4f}".format(test_acc))
```

```
Train accuracy: 0.9952
Test accuracy: 0.9759
```

The performance on the test set is lower than the performance on the training set, which indicates the model is *overfitting*.

TIP: Overfitting is a fundamental concept in Machine Learning and Deep Learning. If you are not familiar with it, have a look at [Chapter 3](#).

Before we start playing with different techniques to reduce overfitting, it is legitimate to ask if we simply don't have enough data to solve the problem.

This is a very common situation: you collect data with labels, you train a model, and the model does not perform as well as you hoped.

What should you do at that point? Should you collect more data? Or should you invest time in searching for better features or a different model?

With the little information we have, it is hard to know which of these alternatives is more likely to help. What is sure, on the other hand, is that all these alternatives carry a cost. For example, let's say you think that more data is what you need.

Collecting more labeled data could be as cheap and simple as downloading a new dataset from your source, or it could be as involved and complex as coordinating with the data collection team at your company, hiring contractors to label the new data, and so on. In other words, the time and cost associated with new data collection strongly vary and need to be assessed case by case.

If, on the other hand, you decided to experiment with new features and model architectures, this could be as simple as adding a few layers and nodes to your model, or as complex as an R&D team dedicating several months to discovering new features for your particular dataset. Again, the actual cost of this option strongly depends on your particular use case.

Is there a way to know which of the two ways is more promising?



The learning curve is a tool we can use to answer that question. Here is how we build it.

First, we set the `x_test` aside, then, we take increasingly large fraction of `x_train` and use them to train the model. For each of these fractions, we fit the model, then we evaluate the model on this fraction and on the test set.

Since the training data is small, we expect the model to overfit the training data and perform quite poorly on the test set.

As we gradually take more training data, the model should improve and learn to generalize better, i.e. the test score should increase. We proceed like this until we have used all our training data.

At this point two cases are possible. If it looks like the test performance stopped increasing with the size of the training set, we probably reached the maximum performance of our model. In this case we should invest time in looking for a better model to improve the performance.

In the second case, it could seem that the test error would continue to decrease if only we had access to more training data. If that's the case, we should probably go out looking for more labeled data first and then worry about changing model.

So, now you know how to answer the big question of more data or better model: *use a learning curve*.

Let's draw one together. First we take increasing fractions of the training data using the function `np.linspace`.

TIP: `np.linspace` returns evenly spaced numbers over a specified interval. In this case we are creating 4 fractions, from 10% to 90% of the data.

```
fractions = np.linspace(0.1, 0.90, 5)
fractions
```

```
array([0.1, 0.3, 0.5, 0.7, 0.9])
```

```
train_sizes = list((len(X_train) * fractions).astype(int))
train_sizes
```

```
[125, 377, 628, 879, 1131]
```

Then we loop over the train sizes, and for each `train_size` we do the following:

- take exactly `train_size` data from the `X_train`
- reset the model to the initial weights

- train the model using only the fraction of training data
- evaluate the model on the fraction of training data
- evaluate the model on the test data
- append both scores to an arrays for plotting

Handling this in the first case (i.e. the first `train_size` in our `train_sizes` array), we'll use our work to then iterate over a longer list of all the `train_sizes`.

Let's create some variables where we'll store our scores:

```
train_scores = []
test_scores = []
```

Now let's break up the test data using the `train_test_split` function as we usually would:

```
x_train_frac, _, y_train_frac, _ = \
    train_test_split(X_train, y_train,
                     train_size=0.1,
                     test_size=None,
                     random_state=0,
                     stratify=y_train)
```

Let's reset the weights to their initial values:

```
model.set_weights(initial_weights)
```

Now we can train our model using the `fit` function, as normal:

```
h = model.fit(X_train_frac, y_train_frac,
               verbose=0,
               epochs=100)
```

With our model trained, let's evaluate it over our training set and save it into the `train_scores` variable from above:

```
r = model.evaluate(X_train_frac, y_train_frac, verbose=0)
train_scores.append(r[-1])
```

Let's do the same with our test set:

```
e = model.evaluate(X_test, y_test, verbose=0)
test_scores.append(e[-1])
```

It's kind of silly to do this manually for every `train_size` entry. Instead, let's iterate over them and build up our `train_scores` and `test_scores` variables:

```
train_scores = []
test_scores = []

for train_size in train_sizes:
    X_train_frac, _, y_train_frac, _ = \
        train_test_split(X_train, y_train,
                         train_size=train_size, test_size=None,
                         random_state=0, stratify=y_train)

    model.set_weights(initial_weights)

    h = model.fit(X_train_frac, y_train_frac,
                  verbose=0,
                  epochs=100)

    r = model.evaluate(X_train_frac, y_train_frac, verbose=0)
    train_scores.append(r[-1])

    e = model.evaluate(X_test, y_test, verbose=0)
    test_scores.append(e[-1])

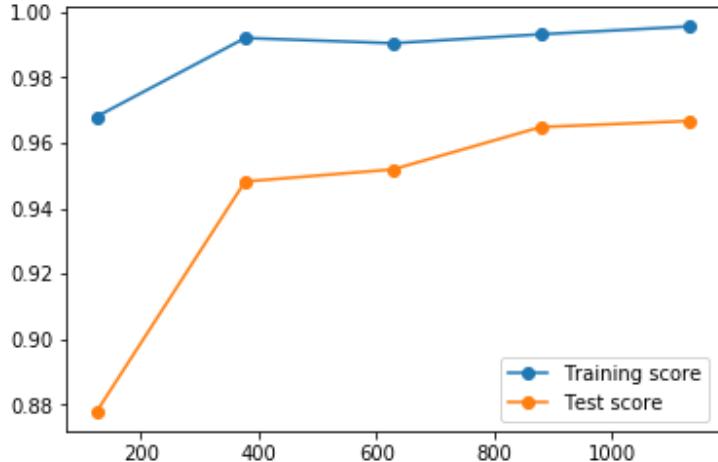
    print("Done size: ", train_size)
```

```
Done size:  125
Done size:  377
Done size:  628
Done size:  879
Done size:  1131
```

Let's plot the training score and the test score as a function of increasing training size:

```
plt.plot(train_sizes, train_scores, 'o-', label="Training score")
plt.plot(train_sizes, test_scores, 'o-', label="Test score")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x7ff8d03732e8>
```



Judging from the curve, it appears the test score would keep improving if we added more data. This is the indication we were looking for. If on the other hand the test score was not improving, it would have been more promising to improve the model first and only then go look for more data if needed.

REDUCING OVERFITTING

Sometimes it's not easy to go out and look for more data. It could be time consuming and expensive. There are a few ways to improve a model and reduce its propensity to overfit without requiring additional data. These fall into the big family of **Regularization techniques**.

The general idea here is the following. By now you should be familiar with the idea that the complexity of a model is somewhat represented by the number of parameters the model has. In simple terms, a model with many layers and many nodes is more complex than a model with a single layer and few nodes. More complexity gives the model, more freedom to learn nuances in our training data. This is what makes neural networks so powerful.

On the other hand, the more freedom a model has, the more likely it will be to overfit on the training data, losing the ability to generalize. We could try to reduce the model freedom by reducing the model complexity, but this would not always be a great idea as it would make the model less able to pick up subtle patterns in our data.

A different approach would be to keep the model very complex, but change something else in the model in order to push it towards less complex solutions. In other words, instead of removing the complexity completely, we allow the model to choose complex solutions, but we push the model towards simpler, more *regular*, solutions. This is what regularization is about. Regularization refers to techniques to keep the complexity of a model from spinning out of control.

Let's review a few ways to regularize a model, and to ease our comparison we will define a few helper functions.

First, let's define a helper function to repeat the training several times. This helper function will be useful to average out any statistical fluctuations in the model behavior due to the random initialization of the weights. We will reset the backend at each iteration in order to save memory and erase any previous training.

Let's load the backend first:

```
import keras.backend as K
```

And then let's define the `repeat_train` helper function. This function expects an already created `model_fn` as input, i.e. a function that returns a model, and it repeats the following process a number of times specified by the input `repeats`:

1. clear the session

```
K.clear_session()
```

- create a model using the `model_fn`

```
model = model_fn()
```

- train the model using the training data

```
h = model.fit(X_train, y_train,
               validation_data=(X_test, y_test),
               verbose=verbose,
               batch_size=batch_size,
               epochs=epochs)
```

- retrieve the accuracy of the model on training data (`acc`) and test data (`val_acc`) and append the results to the `histories` array

```
histories.append([h.history['acc'], h.history['val_acc']])
```

Finally, the `repeat_train` function calculates the average history along with its standard deviation and returns them.

```
def repeat_train(model_fn, repeats=3, epochs=40,
                 verbose=0, batch_size=256):
    """
    Repeatedly train a model on (X_train, y_train),
    averaging the histories.

    Parameters
    -----
    model_fn : a function with no parameters
        Function that returns a Keras model

    repeats : int, (default=3)
        Number of times the training is repeated

    epochs : int, (default=40)
```

```

    Number of epochs for each training run

verbose : int, (default=0)
    Verbose option for the `model.fit` function

batch_size : int, (default=256)
    Batch size for the `model.fit` function

Returns
-----
mean, std : np.array, shape: (epochs, 2)
    mean : array contains the accuracy
    and validation accuracy history averaged
    over the different training runs
    std : array contains the standard deviation
    over the different training runs of
    accuracy and validation accuracy history
"""

histories = []

# repeat model definition and training
for repeat in range(repeats):
    K.clear_session()
    model = model_fn()

    # train model on training data
    h = model.fit(X_train, y_train,
                  validation_data=(X_test, y_test),
                  verbose=verbose,
                  batch_size=batch_size,
                  epochs=epochs)

    # append accuracy and validation accuracy to list
    histories.append([h.history['acc'], h.history['val_acc']])
    print(repeat, end=" ")

histories = np.array(histories)
print()

# calculate mean and standard deviation across repeats:
mean = histories.mean(axis=0)
std = histories.std(axis=0)
return mean, std

```

The `repeat_train` function expects an already created `model_fn` as input. Hence, let's define a new function that will create a fully connected Neural Network with 3 inner layers. We'll call this function

`base_model` , since we will use this basic model for further comparison:

```
def base_model():
    """
    Return a fully connected model with 3 inner layers
    with 1024 nodes each and relu activation function
    """
    model = Sequential()
    model.add(Dense(1024, input_shape=(64,),
                   activation='relu'))
    model.add(Dense(1024,
                   activation='relu'))
    model.add(Dense(1024,
                   activation='relu'))
    model.add(Dense(10, activation='softmax'))
    model.compile('adam', 'categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

TIP: Notice that this model is quite big for the problem we are trying to solve. We purposefully make the model big, so that there are lots of parameters and it can overfit easily.

Now we repeat 5 times the training of the base (non-regularized) model using the `repeat_train` helper function:

```
(m_train_base, m_test_base), (s_train_base, s_test_base) = \
repeat_train(base_model, repeats=5)
```

```
0 1 2 3 4
```

We can plot the histories for training and test. First, let's define an additional helper function `plot_mean_std()` , which plots the average history as a line and add a colored area around it corresponding to +/- 1 standard deviation:

```
def plot_mean_std(m, s):
    """
    Plot the average history as a line
    and add a colored area around it corresponding
    to +/- 1 standard deviation
    """

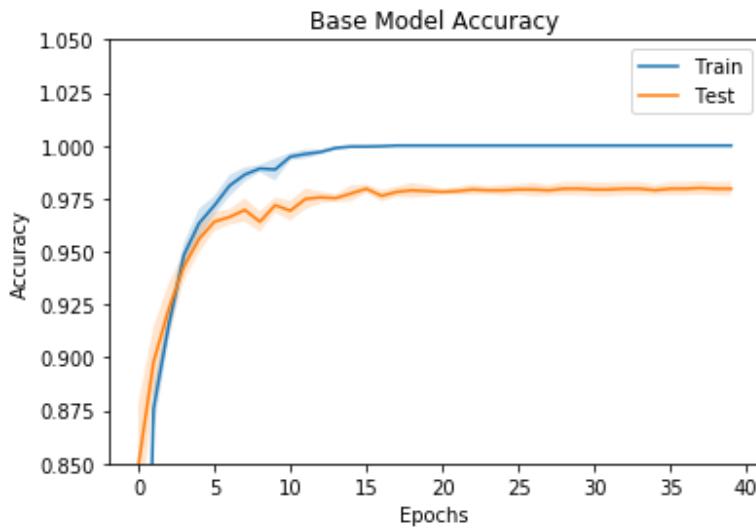
```

```
plt.plot(m)
plt.fill_between(range(len(m)), m-s, m+s, alpha=0.2)
```

Then, let's plot the results obtained training 5 times the base model:

```
plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

plt.title("Base Model Accuracy")
plt.legend(['Train', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05)
plt.show()
```



Overfitting in this case is evident, with the test score saturating at a lower value than the training score.

Model Regularization

[Regularization]([https://en.wikipedia.org/wiki/Regularization_\(mathematics%29](https://en.wikipedia.org/wiki/Regularization_(mathematics%29)) is a common procedure in machine learning and it has been used to improve the performance of complex models with many parameters.

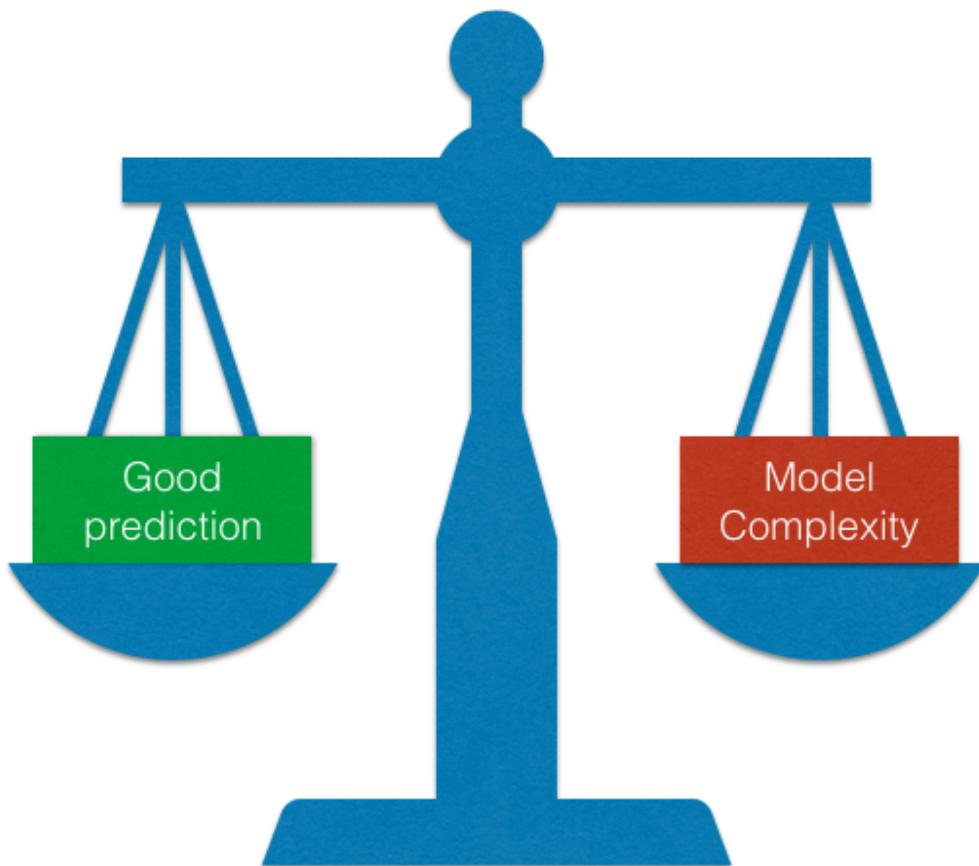
Remember the **Cost Function** we have introduced in **Chapter 3**? The main goal of the cost function is to make sure that the predictions of the model are close to the correct labels.

Regularization works by modifying the original cost function C with an additional term λC_r , that somehow penalizes the complexity of the model:

$$C' = C + \lambda C_r$$

The original cost function C would decrease as the model predictions got closer and closer to the actual labels. In other words, the gradient descent algorithm for the original cost would push the parameters to the region of parameter space that would give the best predictions on the training data. In complex models with many parameters, this could result in overfitting because of all the freedom the model had.

The new penalty C_r pushes the model to be "simple", in other words it grows with the parameters of the model, but it is completely unrelated to the goodness of the prediction.



The total cost C' is a combination of the two terms and therefore the model will have to try to generate the best predictions possible, while retaining simplicity. In other words, the gradient descent algorithm is now solving a constrained minimization problem, where some regions of the parameter space are too expensive to be used for a solution.

The hyper-parameter λ controls the relative strength of the regularization and we can control it.

But how do we implement C_r in practice? There are several ways to do it. **Weight Regularization** assigns a penalty proportional to the size of the weights, for example:

$$C_w = \sum_w |w|$$

or:

$$C_w = \sum_w w^2$$

The first one is called **I1-regularization** and it is the sum of the absolute values of each weight. The second one is called **I2-regularization** and it is the sum of the square values of each weight. While they both suppress complexity, their effect is different.

I1-regularization pushes most weights to be exactly zero, with the exception of a few that will be non-zero. In other words, the net effect of I1-regularization is to make the weight matrix *sparse*.

I2-regularization, on the other hand, suppresses weights quadratically. Suppressing weights quadratically means that any weight larger than the rest will have a much greater contribution to C_r and therefore to the overall cost. The net effect of this is to make all weights equally small.

Similarly to weight regularization, **Bias Regularization** and **Activity Regularization** penalize the cost function with a term proportional to the size of the biases and to the activations respectively.

Let's compare the behavior of our base model with a model with exact the same architecture but endowed with the `l2` weight regularization.

We start by defining a helper function that creates a model with weight regularization: we start from the function `base_model`, and we create the function `regularized_model`, adding the `kernel_regularizer` option to each layer. First of all let's import keras's `l2` regularizer function:

```
from keras.regularizers import l2
```

```
def regularized_model():
    """
    Return an l2-weight-regularized, fully connected model
    with 3 inner layers with 1024 nodes each
    and relu activation function.
    """
```

```

reg = 12(0.005)

model = Sequential()
model.add(Dense(1024,
               input_shape=(64,),
               activation='relu',
               kernel_regularizer=reg))
model.add(Dense(1024,
               activation='relu',
               kernel_regularizer=reg))
model.add(Dense(1024,
               activation='relu',
               kernel_regularizer=reg))
model.add(Dense(10, activation='softmax'))
model.compile('adam', 'categorical_crossentropy',
              metrics=['accuracy'])
return model

```

Now we compare the results of no regularization and l2-regularization. Let's repeat the training 3 times.

```
(m_train_reg, m_test_reg), (s_train_reg, s_test_reg) = \
repeat_train(regularized_model)
```

```
0 1 2
```

TIP: Notice that, since we didn't specified the number of time to train the model, it will repeat the training according to the default parameter, i.e. 3 times.

Let's now compare the performance of the weight regularized model with our base model. We will also plot a dashed line at the maximum test accuracy obtained by the base model:

```

plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

plot_mean_std(m_train_reg, s_train_reg)
plot_mean_std(m_test_reg, s_test_reg)

plt.axhline(m_test_base.max(), linestyle='dashed', color='black')

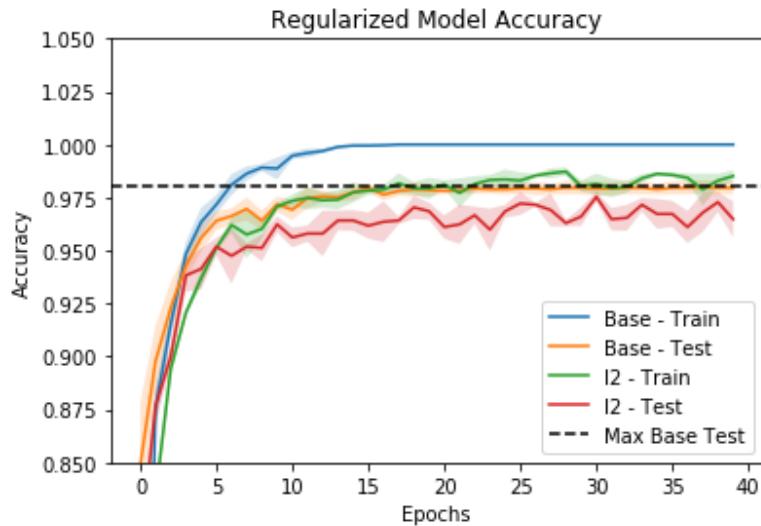
plt.title("Regularized Model Accuracy")
plt.legend(['Base - Train', 'Base - Test',
           'l2 - Train', 'l2 - Test'],
           loc='lower right')

```

```

    'Max Base Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05)
plt.show()

```



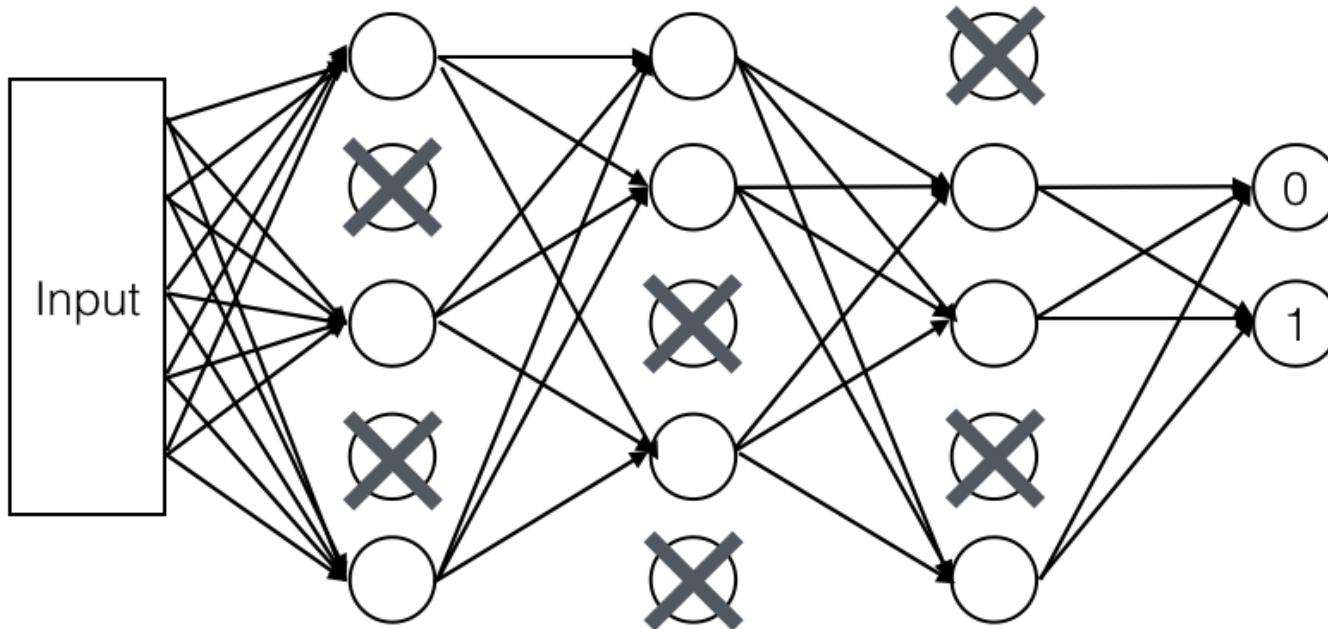
With this particular dataset, weight regularization does not seem to improve the model performance.

This is visually true at least within the small number of epochs we are running. It may be the case that if we let the training run much longer regularization would help, but we don't know for sure and that can cost a lot of time and expense on our compute/memory.

It's however good to know that this technique exists and keep it in mind as one of the options to try. In practice, weight regularization has been superseded by more modern regularization techniques such as **Dropout** and **Batch Normalization**.

Dropout

[Dropout]([https://en.wikipedia.org/wiki/Dropout_\(neural_networks%29](https://en.wikipedia.org/wiki/Dropout_(neural_networks%29)) was introduced in 2014 by Srivastava et al. at University of Toronto in order to address the problem of overfitting in large networks. The key idea of Dropout is to randomly drop units (along with their connections) from the neural network during training.



In other words during the training phase each unit has a non-zero probability not to emit its output to the next layer. This prevents units from co-adapting too much.

Let's reflect on this for a second. Apparently we are damaging the network by dropping a fraction of the units with non zero probability during training time. We are crippling the network and making it a lot harder for it to learn. This is clearly counter-intuitive! Why are we weakening our network?

It turns out the underlying principle is actually quite common in machine learning: we make the network less stable so that the solution found during training is more general, more robust, and more resilient to failure. Another way to look at this is to say that we are adding noise at training time, so that the network will need to learn more general patterns that are resistant to noise.

The technique has similarities with ensemble techniques, because it's as if, during training the network sampled from an many different "thinned" networks, where a fraction of the nodes are not working. At test time, dropout is turned off and a single network with smaller weights is used. This technique has been shown to improve the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification, and many others.

We strongly encourage you to read the [paper](#) if you want to fully understand how dropout is implemented.

On the other hand, if you are eager to apply it, you'll be happy to hear that Dropout is implemented in Keras as a layer, so all we need to do is to add it between the layers. We'll import it first:

```
from keras.layers import Dropout
```

And then we define a `dropout_model`, again starting from the `base_model` and adding the dropout layers. We've tested several configurations and we've found that with this dataset good results can be obtained with a dropout rate of 10% at the input and 50% in the inner layers. Feel free to experiment with different numbers and see what results you get.

TIP: according to the [Documentation](#), in the `Dropout` layer the argument `rate` is a float between 0 and 1, that gives the fraction of the input units to drop.

```
def dropout_model():
    """
    Return a fully connected model
    with 3 inner layers with 1024 nodes each
    and relu activation function. Dropout can
    be applied by selecting the rate of dropout
    """
    input_rate = 0.1
    rate = 0.5

    model = Sequential()
    model.add(Dropout(input_rate, input_shape=(64,)))
    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(rate))
    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(rate))
    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(rate))
    model.add(Dense(10, activation='softmax'))
    model.compile('adam', 'categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Let's train 3 times our network using the `dropout_model`:

```
(m_train_dro, m_test_dro), (s_train_dro, s_test_dro) = \
repeat_train(dropout_model)
```

```
0 1 2
```

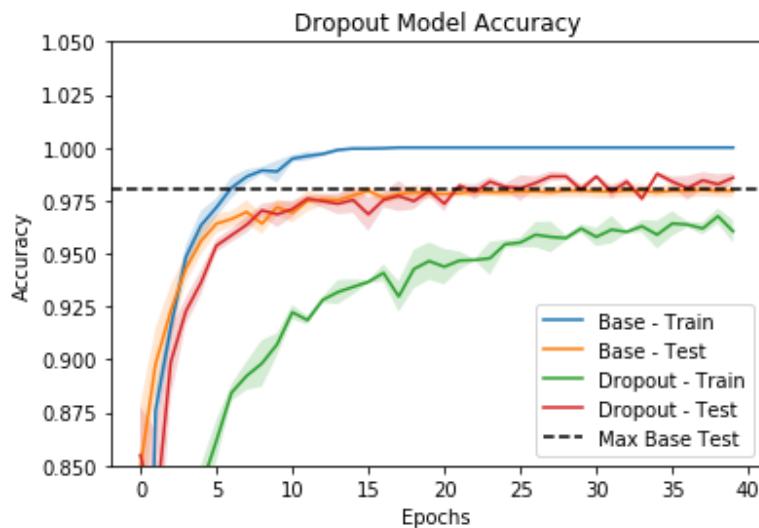
Next, let's plot the accuracy of the dropout model against the base model:

```
plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

plot_mean_std(m_train_dro, s_train_dro)
plot_mean_std(m_test_dro, s_test_dro)

plt.axhline(m_test_base.max(), linestyle='dashed', color='black')

plt.title("Dropout Model Accuracy")
plt.legend(['Base - Train', 'Base - Test',
           'Dropout - Train', 'Dropout - Test',
           'Max Base Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05)
plt.show()
```



Nice! Adding Dropout to our model pushed our test score above the base model for the first time (although not by much)! This is great because we didn't have to add more data. Also, notice how the training score is lower than the test score, which indicates the model is not overfitting and also there seem to be even more room for improvement if we run the training for more epochs!

The Dropout paper also mentions the use of a global constraint to further improve the behavior of a Dropout network. Constraints can be added in Keras through the `kernel_constraint` parameter available in the definition of a layer.

Following the paper, let's see what happens if we impose a `max_norm` constraint to the weights of the model. According to the [Documentation](#), this is equivalent to say that the sum of the square of the weights cannot be higher than a certain *constant*, which can be specified by the user with the argument `c`.

Let's load the `max_norm` constraint first:

```
from keras.constraints import max_norm
```

Let's define a new model function `dropout_max_norm`, that has both `dropout` and the `max_norm` constraint:

```
def dropout_max_norm():
    """
    Return a fully connected model with Dropout
    and Max Norm constraint.
    """
    input_rate = 0.1
    rate = 0.5
    c = 2.0

    model = Sequential()
    model.add(Dropout(input_rate, input_shape=(64,)))
    model.add(Dense(1024, activation='relu',
                   kernel_constraint=max_norm(c)))
    model.add(Dropout(rate))
    model.add(Dense(1024, activation='relu',
                   kernel_constraint=max_norm(c)))
    model.add(Dropout(rate))
    model.add(Dense(1024, activation='relu',
                   kernel_constraint=max_norm(c)))
    model.add(Dropout(rate))
    model.add(Dense(10, activation='softmax'))
    model.compile('adam', 'categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

As usual we can run 3 repeated trainings and average the results:

```
(m_train_dmn, m_test_dmn), (s_train_dmn, s_test_dmn) = \
repeat_train(dropout_max_norm)
```

```
0 1 2
```

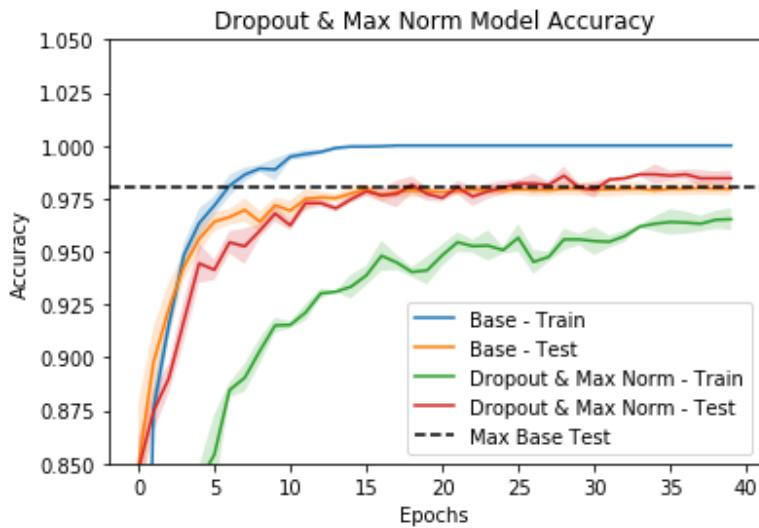
And plot the comparison with the base model:

```
plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

plot_mean_std(m_train_dmn, s_train_dmn)
plot_mean_std(m_test_dmn, s_test_dmn)

plt.axhline(m_test_base.max(), linestyle='dashed', color='black')

plt.title("Dropout & Max Norm Model Accuracy")
plt.legend(['Base - Train', 'Base - Test',
           'Dropout & Max Norm - Train', 'Dropout & Max Norm - Test',
           'Max Base Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05)
plt.show()
```



In this particular case the Max Norm constraint does not seem to produce results that are qualitatively different from the simple Dropout, but there may be datasets where this constraint helps make the network converge to a better result.

Batch Normalization

Batch Normalization was introduced in 2015 as an even better regularization technique, as described in this paper. The authors of the paper started from the observation that training of deep neural networks is slow because the distribution of the inputs to a layer changes during training, as the parameters of the previous layers change. Since the inputs to a layer are the outputs of the previous layer, and these are determined by the parameters of the previous layer, as training proceeds the distribution of the output may drift, making it harder for the next layer to adapt.

The authors' solution to this problem is to introduce a normalization step between layers, that will take the output values for the current batch and normalize them by removing the mean and dividing by the standard deviation. They observe that their technique allows to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout.

Let's walk through the batch algorithm with a small code example. First we calculate the mean and standard deviation of the batch:

```
mu_B = X_batch.mean()  
std_B = X_batch.std()
```

Then we subtract the mean and divide by the standard deviation:

```
X_batch_scaled = (X_batch - mu_B) / np.sqrt(std_B**2 + 0.0001)
```

Finally we rescale the batch with 2 parameters γ and β that are learned during training:

```
X_batch_norm = gamma * X_batch_rescaled + beta
```

TIP: Using math notation, the complete algorithm for Batch normalization is the following. Given a mini-batch $B = \{x_1 \dots m\}$

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

Batch Normalization is very powerful, and Keras makes it available as a layer too, as described in the [Documentation](#). One important thing to note is that BN needs to be applied before the nonlinear activation function. Let's see how it's done. First we load the `BatchNormalization` and `Activation` layers:

```
from keras.layers import BatchNormalization, Activation
```

Then we define again a new model function `batch_norm_model` that adds Batch Normalization to our fully connected network defined in the `base_model`:

```
def batch_norm_model():
    """
    Return a fully connected model with
    Batch Normalization.

    Returns
    ----
    model : a compiled keras model
    """
    model = Sequential()

    model.add(Dense(1024, input_shape=(64,)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Dense(1024))
    model.add(BatchNormalization())
    model.add(Activation('relu'))

    model.add(Dense(1024))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
```

```

model.add(Dense(10))
model.add(BatchNormalization())
model.add(Activation('softmax'))

model.compile('adam', 'categorical_crossentropy',
              metrics=['accuracy'])
return model

```

Batch Normalization seems to work better with smaller batches, so we will run the `repeat_train` function with a smaller `batch_size`.

Since smaller batches mean more weight updates at each epoch we will also run the training for less epochs.

Let's do a quick back of the envelope calculation.

We have 1257 points in the training set. Previously, we used batches of 256 points, which gives 5 weight updates per epoch, and a total of 200 updates in 40 epochs. If we reduce the batch size to 32, we will have 40 updates at each epoch, so we should run the training for only 5 epochs.

We will actually run it a bit longer in order to see the effectiveness of Batch Normalization. 10-15 epochs will suffice to bring the model accuracy to a much higher value on the test set.

```
(m_train_bn, m_test_bn), (s_train_bn, s_test_bn) = \
repeat_train(batch_norm_model, batch_size=32, epochs=15)
```

0 1 2

Let's plot the results and compare with the base model:

```

plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

plot_mean_std(m_train_bn, s_train_bn)
plot_mean_std(m_test_bn, s_test_bn)

plt.axhline(m_test_base.max(), linestyle='dashed', color='black')

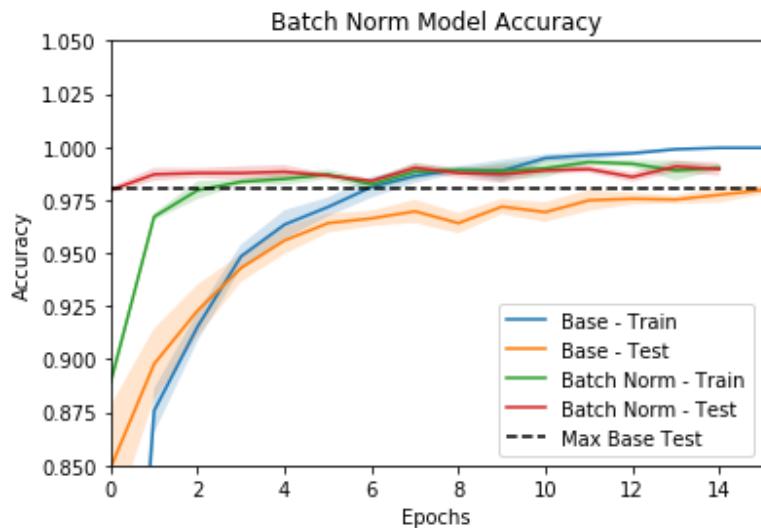
plt.title("Batch Norm Model Accuracy")
plt.legend(['Base - Train', 'Base - Test'],

```

```

        'Batch Norm - Train', 'Batch Norm - Test',
        'Max Base Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05)
plt.xlim(0, 15)
plt.show()

```



Awesome! With the addition of Batch Normalization, the model converged to a solution that is better able to generalize on the Test set, i.e. it is overfitting a lot less than the base solution.

DATA AUGMENTATION

Another very powerful technique to improve the performance of a model without requiring the collection of new data is **Data Augmentation**. Let's consider the problem of image recognition, and to make things practical, let's consider this nice picture of a squirrel:



squirrel

If your goal was to recognize the animal in this picture, you would still be able to solve the task effectively even if we distorted the image or rotated it. In fact there's a variety of transformations that we could apply to the image, without altering its information content, including:

- rotation
- shift (up, down, left, right)
- shear
- zoom
- flip (vertical, horizontal)
- rescale
- color correction and changes
- partial occlusion

All these transformations would not destroy the information contained in the image. They would just change the absolute values of the pixels. A human would still be able to recognize a rotated squirrel or a shifted panda, very much like you can still recognize your friends after all the filters they apply to their selfies. This property means a good image recognition algorithm should also be resilient to this kind of transformations.

If we apply these transformation to an image in our training dataset, we can generate an infinite number of variations of such image, giving us access to a much much larger synthetic training dataset. This process is what data augmentation is about: generating new labeled data points starting from existing data through the use of valid transformations.

Although the example we provided is in the domain of image recognition, the same process can be applied to augment other kinds of data, for example speech samples for a speech recognition task. Given a certain sound file we can change its speed and pitch, add background noise, add silences to generate variations of the speech snippet that would still be perfectly understood by a human.

Let's see how Keras allows us to do it easily for images. We need to load the `ImageDataGenerator` object:

```
from keras.preprocessing.image import ImageDataGenerator
```

This class creates a generator that can apply all sorts of variations to an input image. Let's initialize it with a few parameters:

- We'll set the `rescale` factor to $1/255$ to normalize pixel values to the interval [0-1]
 - We'll set the `width_shift_range` and `height_shift_range` to $\pm 10\%$ of the total range
 - We'll set the `rotation_range` to ± 20 degrees
 - We'll set the `shear_range` to ± 0.3 degrees
 - We'll set the `zoom_range` to $\pm 30\%$
 - We'll allow for `horizontal_flip` of the image

See the [Documentation](#) for a complete overview of all the available arguments.

```
idg = ImageDataGenerator(rescale = 1./255,
                        width_shift_range=0.1,
                        height_shift_range=0.1,
                        rotation_range = 20,
                        shear_range = 0.3,
                        zoom_range = 0.3,
                        horizontal_flip = True)
```

The next step is to create an iterator that will generate images with the image data generator. We basically need to tell where our training data are. Here we use the method `flow_from_directory`, which is useful when we have images stored in a directory, and we tell it to produce target images of size 128x128. The input folder structure need to be:

top/
 class_0/
 class_1/

Where `top` is the folder we will flow from, and the images are organized into one subfolder for each class.

```
Found 1 images belonging to 1 classes.
```

Let's generate a few images and display them:

```
plt.figure(figsize=(12, 12))

for i in range(16):
    img, label = train_gen.next()
    plt.subplot(4, 4, i+1)
    plt.imshow(img[0])
```



Great! In all of the images the squirrel is still visible and from a single image we have generated 16 different images that we can use for training!

Let's apply this technique to our digits and see if we can improve the score on the test set. We will use slightly less dramatic transformations and also fill the empty space with zeros along the border.

```
digit_idg = ImageDataGenerator(width_shift_range=0.1,
                                height_shift_range=0.1,
                                rotation_range = 10,
                                shear_range = 0.1,
                                zoom_range = 0.1,
                                fill_mode='constant')
```

We will need to reshape our data into tensors with 4 axes, in order to use it with the `ImageDataGenerator`, so let's do it:

```
X_train_t = X_train.reshape(-1, 8, 8, 1)
X_test_t = X_test.reshape(-1, 8, 8, 1)
```

We can use the method `.flow` to flow directly from a dataset. We will need to provide the labels as well.

```
train_gen = digit_idg.flow(X_train_t, y=y_train)
```

Notice that by default the `.flow` method generates a batch of 32 images with corresponding labels:

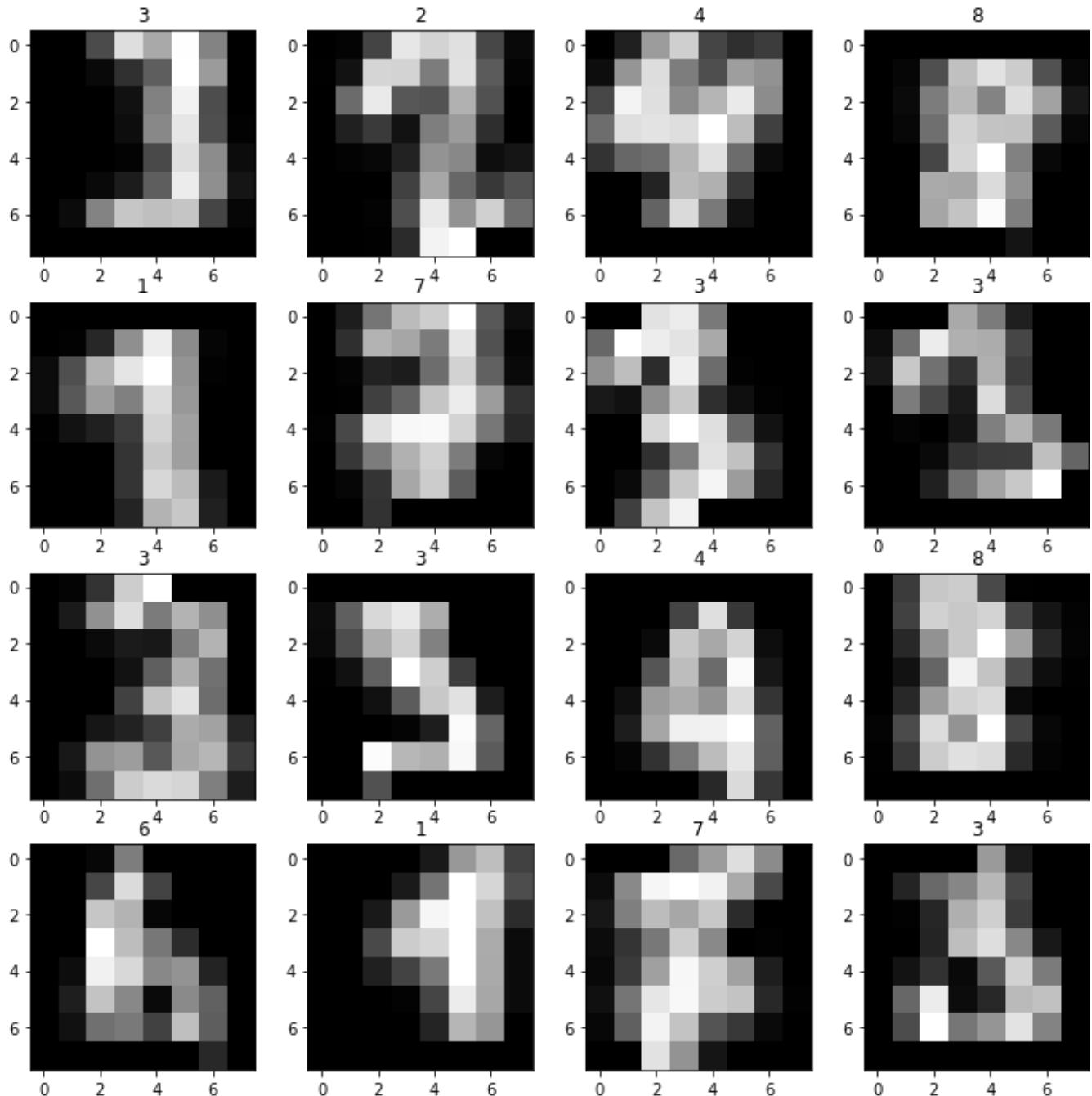
```
imgs, labels = train_gen.next()
```

```
imgs.shape
```

(32, 8, 8, 1)

Let's display a few of them:

```
plt.figure(figsize=(12, 12))
for i in range(16):
    plt.subplot(4, 4, i+1)
    plt.imshow(imgs[i,:,:,0], cmap='gray')
    plt.title(np.argmax(labels[i]))
```



As you can see the digits are deformed, due to the very low resolution of the images. Will this help our network or confuse it? Let's find out!

We will need a model that is able to deal with a tensor input, since the images are now tensors of order 4. Luckily, it's very simple to adapt our base model to have a `Flatten` layer as input:

```
from keras.layers import Flatten
```

```
def tensor_model():
    model = Sequential()
    model.add(Flatten(input_shape=(8, 8, 1)))
    model.add(Dense(1024, activation='relu'))
    model.add(Dense(1024, activation='relu'))
    model.add(Dense(1024, activation='relu'))
    model.add(Dense(10, activation='softmax'))
    model.compile('adam', 'categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

We also need to define a new `repeat_train_generator` function that allows to train a model from a generator. We can take the original `repeat_train` function and modify it. We will follow the same procedure used in with 2 difference:

1. We'll define a generator that yields batches from `x_train_t` using the image data generator
2. We'll replace the `.fit` function:

```
h = model.fit(X_train, y_train,
               validation_data=(X_test, y_test),
               verbose=verbose,
               batch_size=batch_size,
               epochs=epochs)
```

with the `.fit_generator` function:

```
h = model.fit_generator(train_gen,
                        steps_per_epoch=steps_per_epoch,
                        epochs=epochs,
                        validation_data=(X_test_t, y_test),
                        verbose=verbose)
```

Notice that, since we are now feeding variations of the data in the training set the concept of an *epoch* becomes blurry. When does an epoch terminate if we flow random variations of the training data? The `model.fit_generator` function allows us to define how many `steps_per_epoch` we want. We will use the value of 5, with a `batch_size` of 256 like in most of the examples above.

```
def repeat_train_generator(model_fn, repeats=3,
                           epochs=40, verbose=0,
                           steps_per_epoch=5,
                           batch_size=256):
    """
    Repeatedly train a model on (X_train, y_train),
    averaging the histories using a generator.

    Parameters
    -----
    model_fn : a function with no parameters
        Function that returns a Keras model

    repeats : int, (default=3)
        Number of times the training is repeated

    epochs : int, (default=40)
        Number of epochs for each training run

    verbose : int, (default=0)
        Verbose option for the `model.fit` function

    steps_per_epoch : int, (default=5)
        Steps_per_epoch for the `model.fit` function

    batch_size : int, (default=256)
        Batch size for the `model.fit` function

    Returns
    -----
    mean, std : np.array, shape: (epochs, 2)
        mean : array contains the accuracy
        and validation accuracy history averaged
        over the different training runs
        std : array contains the standard deviation
        over the different training runs of
        accuracy and validation accuracy history
    """
    # generator that flows batches from X_train_t
    train_gen = digit_idg.flow(X_train_t, y=y_train,
                               batch_size=batch_size)

    histories = []

    # repeat model definition and training
    for repeat in range(repeats):
        K.clear_session()
```

```

model = model_fn()

# to train with a generator use .fit_generator()
h = model.fit_generator(train_gen,
    steps_per_epoch=steps_per_epoch,
    epochs=epochs,
    validation_data=(X_test_t, y_test),
    verbose=verbose)

# append accuracy and validation accuracy to list
histories.append([h.history['acc'], h.history['val_acc']])
print(repeat, end=" ")

histories = np.array(histories)
print()

# calculate mean and standard deviation across repeats:
mean = histories.mean(axis=0)
std = histories.std(axis=0)
return mean, std

```

Once the function is defined, we can train it as usual:

```
(m_train_gen, m_test_gen), (s_train_gen, s_test_gen) = \
repeat_train_generator(tensor_model)
```

```
0 1 2
```

And compare the results with our base model:

```

plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

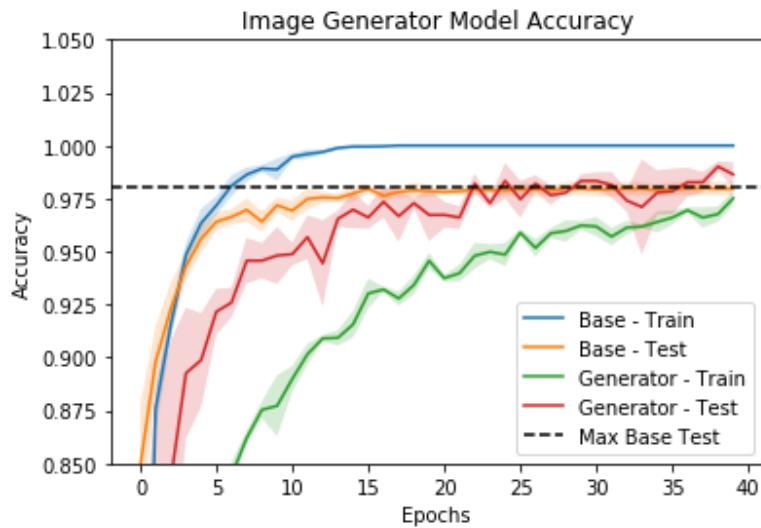
plot_mean_std(m_train_gen, s_train_gen)
plot_mean_std(m_test_gen, s_test_gen)

plt.axhline(m_test_base.max(), linestyle='dashed', color='black')

plt.title("Image Generator Model Accuracy")
plt.legend(['Base - Train', 'Base - Test',
           'Generator - Train', 'Generator - Test',
           'Max Base Test'])

```

```
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05)
plt.show()
```



As you can see, the Data Augmentation process improved the performance of the model on our test set. This makes a lot of sense, because by feeding variations of the input data as training we have made the model more resilient to changes in the input features.

HYPERPARAMETER OPTIMIZATION

One final note on hyper-parameter optimization. Neural network models have a lot of hyper-parameters.

These are things like:

- model architecture
 - number of layers
 - type of layers
 - number of nodes
 - activation functions
 - ...
- optimizer parameters
 - optimizer type
 - learning rate
 - momentum
 - ...
- training parameters
 - batch size
 - learning rate scheduling
 - number of epochs
 - ...

These parameters are called **Hyper**-parameters because they define the training experiment and the model is not allowed to change them while training. That said, they turn out to be really important in determining the success of a model in solving a particular problem.

The topic of hyper-parameter tuning is really vast and we don't have space to cover it but we want to mention here a few tools that can help you achieve optimal hyper-parameter tuning.

Hyperopt and Hyperas

[Hyperopt](#) is a python library that can perform generalized hyper-parameter tuning using a technique called Bayesian Optimization.

[Hyperas](#) is a library that connects Hyperopt and Keras, making it easy to run parallel trainings of a keras model with variations in the values of the hyper-parameters.

Cloud based tools

SigOpt is a cloud based implementation of Bayesian hyperparameter search.

AWS SageMaker and Google Cloud ML offer options for spawning parallel training experiments with different hyper-parameter combinations.

Determined.ai and Pipeline.ai also offer this feature as part of their cloud training platform.

EXERCISES

Exercise 1

This is a long and complex exercise, that should give you an idea of a real world scenario. Feel free to look at the solution if you feel lost. Also, feel free to run this on a GPU.

First of all download and unpack the male/female pictures from [here](#) into a subfolder of the `../data` folder. These images and labels were obtained from [Crowdflower](#).

Your goal is to build an image classifier that will recognize the gender of a person from pictures.

- Have a look at the directory structure and inspect a couple of pictures
- Design a model that will take a color image of size 64x64 as input and return a binary output (`female=0/male=1`)
- Feel free to introduce any regularization technique in your model (Dropout, Batch Normalization, Weight Regularization)
- Compile your model with an optimizer of your choice
- Using `ImageDataGenerator`, define a train generator that will augment your images with some geometric transformations. Feel free to choose the parameters that make sense to you.
- Define also a test generator, whose only purpose is to rescale the pixels by `1./255`
- use the function `flow_from_directory` to generate batches from the train and test folders. Make sure you set the `target_size` to `64x64`.
- Use the `model.fit_generator` function to fit the model on the batches generated from the `ImageDataGenerator`. Since you are streaming and augmenting the data in real time you will have to decide how many batches make an epoch and how many epochs you want to run
- Train your model (you should get to at least 85% accuracy)
- Once you are satisfied with your training, check a few of the misclassified pictures.
- Read about [human bias in machine learning datasets](#)

Chapter 11: Pre-trained models for images

As a recap of all the great work we've done so far:

We started our journey from the basics of [Data Manipulation](#), and [\[Machine Learning\]](#), ([./3_Machine_Learning.ipynb](#)), and then we introduced [Deep Learning](#) and neural networks. We learned about [Deep Learning Internals](#) and the math that makes neural networks function. Then we explored more complex architectures like [Convolutional Neural Networks for Images](#) and [Recurrent Neural Networks for Time Series](#) and for [Text Data](#). Finally, we learned how to [train our models](#) [GPUs](#) to speed up training and how to [improve](#) a model if it's overfitting. With [Chapter 10](#) we conclude the part of the book that deals with understanding how neural networks work and how they can be trained and we shift gears to more recent applications.

Many of the techniques we learned are only a few years old, yet the field of deep learning is evolving **really fast** and in the last years many new techniques have been invented and discovered.

This chapter walks through how to piggyback on the shoulders of giants. In fact, will learn how to use pre-trained networks, i.e. networks that have already been trained on a similar task, and adapt them to the task we would like to perform.

These are often very large networks, with tens of millions of parameters, that have been trained on very large datasets. It would cost us a lot of computing power to re-train them from scratch. Luckily, we don't need to do that. Let's see how.

As usual, we start by importing Numpy, Pandas and Matplotlib.

```
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
```

RECOGNIZING SPORTS FROM IMAGES

Let's say we'd like to classify a set of images related to sports. We know that a convolutional neural network can solve the image classification task, but we do not have tens of thousands of images to train it. Can we still achieve the goal? The answer is yes! Let's see how.

First let's load a dataset containing links to images of sports:

```
df = pd.read_csv('..../data/sports.csv')
```

and let's inspect it with the `.head()` command:

```
df.head()
```

As you can see the dataset contains 3 columns:

- the image url
 - the class
 - the label confidence

Let's first have a look at how many classes there are using the `.value_counts()` method on the `df['class']` column. This method groups the entries in the column by equal type and counts how many occurrences there are in each group:

```
df['class'].value_counts()
```

```
Cross-country skiing    1003
Beach volleyball        1002
Formula racing          1001
Name: class, dtype: int64
...
Name: class, dtype: int64
...
Name: class, dtype: int64
```

There are 3 classes, with approximately a thousand images each. This is not enough examples to train a convolutional network from scratch. We'll need to use *transfer learning* to solve the problem.

Before we dive into it, let's prepare train and test datasets and let's download all the images to disk. We first import the `train_test_split` function from Scikit-Learn:

```
from sklearn.model_selection import train_test_split
```

Then we split the dataframe `df` into 70% train and 30% test. Notice that we stratify the split according to the class distribution, i.e. we make sure that the train set (and the test set) is composed by 1/3 skiing, 1/3 volley and 1/3 formula racing.

```
train_df, test_df = train_test_split(
    df,
    test_size=0.3,
    random_state=2,
    stratify=df['class']
)
```

Now that we have set up our datasets, we need to download the images. Let's define a helper function that checks if an image has already been downloaded and it downloads it if it doesn't exist. We import the `os` module to be able to create folders and files:

```
import os
```

We also load the `urlretrieve` (from the `urllib.request` library) function that allows us to download an image from a url:

```
from urllib.request import urlretrieve
```

Then let's define a `maybe_download_image` function:

```
def maybe_download_image(save_dir, image_url, label):
    """
    Download image to save_dir/label/. The function will
    first check if the image already exists. Returns 0 if found,
    1 if downloaded

    Args:
        save_dir: The output path where to save the images)
        image_url: An image url
        image: A label
    """

    # create the output path if it doesn't exist
    os.makedirs(save_dir, exist_ok=True)

    # create label subfolder if it doesn't exist
    label_dir = os.path.join(save_dir, label)
    os.makedirs(label_dir, exist_ok=True)

    # split the file name from the url
    url, fname = os.path.split(image_url)

    # return 0 if file already there, 1 if downloaded
    save_path = os.path.join(save_dir, label, fname)
    if os.path.isfile(save_path):
        return 0
    else:
        urlretrieve(image_url, save_path)
        return 1
```

Let's test our function on the first item in the dataframe. Let's retrieve the first url:

```
image_url = df['image_url'][0]
image_url
```

```
'https://multimedia-commons.s3-us-west-2.amazonaws.com/data/images/7ad/a7b/7ada7b21d67124:
```

and the first label:

```
label = df['class'][0]  
label
```

```
'Formula racing'
```

Now let's download the image using our helper function:

```
maybe_download_image('/tmp/ztdlbook/', image_url, label)
```

```
0
```

The function returns 1, since the image was not there. Notice that if we run it again, this time the function will return 0:

```
maybe_download_image('/tmp/ztdlbook/', image_url, label)
```

```
0
```

Now we need to download all the images in the dataframe. We could simply loop over the rows, but this can be tediously slow. Instead we'll resort to asynchronous downloading and we'll start many threads to download the images concurrently. To do this we need to import the `ThreadPoolExecutor` from the `concurrent.futures` library, as well as the `as_completed` function:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
```

The `as_completed` function is an iterator over the given futures that yields each as it completes. With these two components, let's build another helper function that distributes all the urls to a `ThreadPoolExecutor` and runs them in parallel.

```
def get_images(save_dir, image_urls, image_labels, max_workers=20):
    """
    Download a list of images with labels using parallel threads.

    Args:
        save_dir: The output path where to save the images)
        image_urls: A list of image urls
        image_labels: A list of image labels
        max_workers: Number of concurrent threads to start (default=20)
    """
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        # we build a dictionary with executors as keys and the urls as values
        future_to_url = {executor.submit(maybe_download_image, save_dir, url, label):url
                         for url, label in zip(image_urls, image_labels)}

        # we loop over executors as they complete and print their return values
        for future in as_completed(future_to_url):
            url = future_to_url[future]
            try:
                result = future.result()
                print(result, end=' ')
            except Exception as ex:
                print('%r generated an exception: %s' % (url, ex))
```

Let's run the `get_images` function on the train dataset. We'll save them in a `sports/train` folder inside `../data`:

```
train_path = '../data/sports/train/'
```

```
get_images(train_path,  
          train_df['image_url'].values,  
          train_df['class'].values)
```

Similarly we'll save the test images in a `sports/test` folder inside `./data`:

```
test_path = '../data/sports/test/'
```

```
get_images(test_path,
           test_df['image_url'].values,
           test_df['class'].values)
```

Now that we have downloaded the data, we are ready to tackle transfer learning

KERAS APPLICATIONS

Keras offers many pre-trained models in the `keras.applications` module. All of them are models trained for image classification on the Imagenet dataset and they have different architectures. Here we summarize their main properties:

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters (M)	Depth	Load time (s)	Predict time (s)
Xception	88	0.790	0.945		23	126	6.3
VGG16	528	0.715	0.901		138	23	6.5
VGG19	549	0.727	0.910		144	26	7.3
ResNet50	99	0.759	0.929		26	168	10.9
InceptionV3	92	0.788	0.944		24	159	17.4
InceptionResNetV2	215	0.804	0.953		56	572	47.6
MobileNet	17	0.665	0.871		4	88	9.7
DenseNet121	33	0.745	0.918		8	121	32.6
DenseNet169	57	0.759	0.928		14	169	57.6
DenseNet201	80	0.770	0.933		20	201	80.3

As you can see, some of them have a large memory footprint (up to over 500Mb), while some others trade a bit of accuracy for a smaller footprint that makes them perfect to run on a mobile phone.

Pre-trained models are awesome for two reasons:

1. we can use them without training to classify images of everyday objects
- we can partially retrain them and adapt them to classify new objects, using only a few input images and a laptop (no need for GPU)!

Let's go ahead and explore both these applications. First of all we're going to load the `image` module from `keras.preprocessing`, which will allow us to load images from disk:

```
from keras.preprocessing import image
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-packages/h5py/_init__.py:36: FutureWarning: from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Now let's load an image from the ones we have downloaded previously. Let's define the `input_path`:

```
input_path = '../data/sports/train/Beach volleyball/1d8a1f53f36487ac4f10e47e6937308.jpg'
```

and then load the image:

```
img = image.load_img(input_path, target_size=(299, 299))
```

Jupyter notebook can display images inline, so let's have a look at it:

```
img
```



What type of python object is `img`? We can check it with the `type` function and see that it's a python Image data type.

```
type(img)
```

```
PIL.Image.Image
```

Now let's convert it to a numpy array so that we can feed it to the model. We will use the `img_to_array` function from the `image` module we've just loaded:

```
img_array = image.img_to_array(img)
```

Now the image is an order-3 tensor with 229 pixels in Height and Width and 3 color channels for RGB:

```
img_array.shape
```

```
(299, 299, 3)
```

Keras convolutional models require an input with 4 axes, i.e. an order-4 tensor, where the first axis locates the image in the dataset (in this case we only have one image, but we can still think of it as the first element in an order-4 array. We can add this "dummy" dimension with the `np.expand_dims` function:

```
img_tensor = np.expand_dims(img_array, axis=0)
```

Let's double check that the shape of this new tensor is the one we want:

```
img_tensor.shape
```

```
(1, 299, 299, 3)
```

PREDICT CLASS WITH PRE-TRAINED XCEPTION

Amongst all the pre-trained models provided we really like the `Xception` network. Not only it provides great accuracy with a small footprint and load time, but also it was invented by the founder of Keras, François Chollet. Let's go ahead and import the `Xception` class:

```
from keras.applications.xception import Xception
```

We can create a pre-trained model simply by creating an instance of `Xception` with the `weights='imagenet'` parameter. This command will download the pre-trained weights and create a model with the `Xception` architecture.

TIP: note that it could take a few minutes to download the weights.

```
model = Xception(weights='imagenet')
```

Let's have a look at the model architecture by printing the summary:

```
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	(None, None, None, 3)	0	
block1_conv1 (Conv2D)	(None, None, None, 3)	864	input_1[0][0]
block1_conv1_bn (BatchNormaliza	(None, None, None, 3)	128	block1_conv1[0][0]
...			

Wow! What a huge model! Notice that it has almost 23 Million parameters and many convolutional layers stacked on top of one another. Let's test the pre-trained model on the task of recognizing an image without training.

We will need to pre-process the image so that it has the correct format for the network. Luckily for us, the `keras.applications.xception` module also contains a `preprocess_input` function that is exactly what we need. Let's load it:

```
from keras.applications.xception import preprocess_input
```

and let's apply it to a copy of our tensor image:

TIP: we apply it to a copy because the function alters the argument itself and we don't want to alter the original version of the image.

```
img_scaled = preprocess_input(np.copy(img_tensor))
```

`img_scaled` is scaled such that the minimum value is -1 and the maximum value is 1. We can pass it to the model and generate a prediction:

```
preds = model.predict(img_scaled)
```

What do our predictions look like? What is the output of the model? Let's first look at the shape of the `preds` object:

```
preds.shape
```

```
(1, 1000)
```

`preds` is a vector with 1000 entries. This makes sense: they are the probabilities associated with each of the 1000 classes of objects in the Imagenet dataset.

To recap, our pre-trained Xception model takes an image in input and returns a softmax classification output with 1000 classes. To interpret the prediction we will load the `decode_predictions` function:

```
from keras.applications.xception import decode_predictions
```

and apply it to the `preds` vector to get the top three most likely labels for the photo.

```
decode_predictions(preds, top=3)[0]
```

```
[('n04540053', 'volleyball', 0.9431326),  
 ('n09421951', 'sandbar', 0.021305446),  
 ('n04371430', 'swimming_trunks', 0.006839624)]  
...  
('n04371430', 'swimming_trunks', 0.006839624)]  
...  
('n04371430', 'swimming_trunks', 0.006839624)]
```

Not bad! Our model thinks it's very likely that the photo is about volleyball, which is not far from beach volleyball at all! How awesome is that?! Now let's do even better, let's repurpose the model so that it will work with exactly the 3 categories of pictures we have. This is called *Transfer Learning*.

Transfer Learning

Transfer Learning consists in leveraging a pre-trained model to solve a similar task. In this case, we're going to use a network that was trained on imagenet and re-purpose it to solve the sport image classification task. By using a pre-trained network we don't need to train it completely from scratch, a great advantage both in terms of computing power required and in terms of amount of data needed.

We will be able to adapt a very large network like Xception, that has more than 20 Million parameters, using a laptop and a few thousand images. This is an incredibly powerful function! Let's see how it's done.

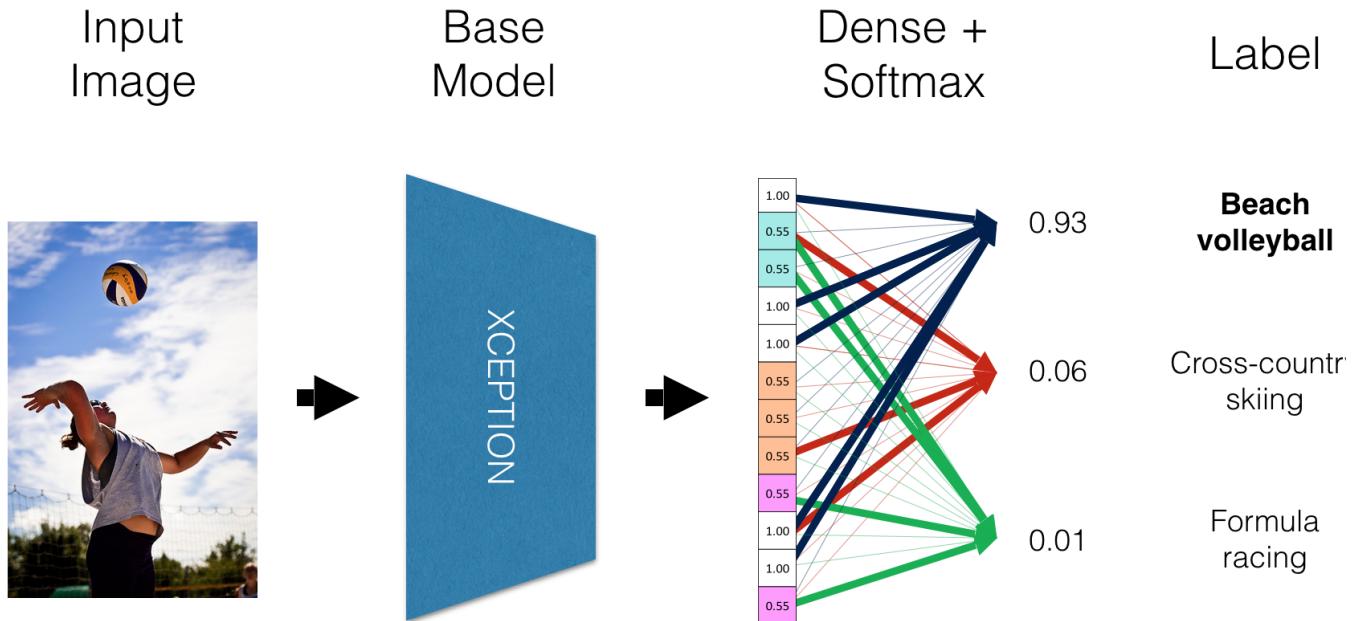
First of all we're going to set a value for the `img_size = 299`. This is the correct input size for Xception and it corresponds to the size of images it was originally trained on.

```
img_size = 299
```

We reload the `xception` model, but this time we include a couple more arguments besides the weights.

First of all we specify `include_top=False`. This option says we don't want the full model, but only the convolutional part. If you remember what we've learned in [Chapter 6 on CNNs](#), convolutional models are composed by a cascade of convolutional layers that yield more and more specialized feature maps.

At some point the feature maps are flattened to an array which is fed to a `Dense` layer (or a series of `Dense` layers) and finally to the output of the classification. Here we want to load all the layers of Xception up to the layer before the last fully connected (the *top* layer). The reason we want to do this is simple to explain. We want to use the pre-trained model as a giant pre-processing layer that takes an image in input and returns a few thousand high-level features (we well see these are called *bottleneck features*). We will then use these high level features to perform a standard classification with only the classes of images present in our dataset, i.e. the 3 sports.



When loading the Xception model, we also specify the `input_shape` and an additional parameter `pooling='avg'`. This last one specifies how we'd like to go from the order-4 tensor of the feature maps to the order-2 tensor that goes in the fully connected top. `pooling='avg'` means we're going to apply a Global Average Pooling layer at the end.

Let's load this into a variable called `base_model`:

```
base_model = Xception(include_top=False, weights='imagenet',
                      input_shape=(img_size, img_size, 3),
                      pooling='avg')
```

Now that we've loaded the base model, we're going to complete the model with a couple of dense layers. First we load the `Sequential` model and the `Dense` and `Dropout` layers:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
```

Let's create model with the following architecture.

First we pass the whole `base_model` as the first layer. This will take an image in input, process it with the pre-trained `xception` weights, and pass an array of numbers to the next layer. Then we'll load a fully

connected layer with 256 nodes and a ReLU activation, then Dropout and finally the output layer with 3 nodes and a Softmax. Remember that we have only 3 classes:

- Beach volleyball
- Cross-country skiing
- Formula racing

that are mutually exclusive, i.e. a picture is only about one of the 3 sports, so our output needs to have 3 nodes with a Softmax:

```
model = Sequential()
model.add(base_model)
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(3, activation='softmax'))
```

Let's take a quick look at the model summary:

```
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
xception (Model)	(None, 2048)	20861480
dense_1 (Dense)	(None, 256)	524544
dropout_1 (Dropout)	(None, 256)	0
<hr/> ... <hr/>		

Wow! This model still has 20+ millions of parameters. Now here's the trick: we're going to set most of them to be frozen, i.e backpropagation will not touch them at all! This is obtained by setting the `.trainable` attribute of a layer to `False`. Since we've added the `base_model` as the first layer, we'll only need to set that flag:

```
model.layers[0].trainable = False
```

Let's check the model summary again.

```
model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
xception (Model)	(None, 2048)	20861480
dense_1 (Dense)	(None, 256)	524544
dropout_1 (Dropout)	(None, 256)	0
<hr/> ... <hr/>		

Of the total number of parameters only a half a million are now trainable, the ones that belong to the 2 dense layers we've added after the `base_model`. This seems a much more tractable model than the original one!

Let's go ahead and compile the model:

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

We are now ready to train it.

DATA AUGMENTATION

Since we don't have too much data available, we'll use the trick of data augmentation learned in Chapter 10. This consists in generating variations of an image with transformations such as zoom, rotate and shear. Let's load the `ImageDataGenerator` :

```
from keras.preprocessing.image import ImageDataGenerator
```

Let's set a `batch_size = 32`. The choice of this number is somewhat arbitrary, but since we have 3 classes only, a batch of 32 images will contain on average about 10 images for each class. This seems a good number of examples to learn something from:

batch size = 32

Now let's create an instance of `ImageDataGenerator` that applies transformations to the training set. It will do the following operations:

- apply the `preprocess_input` function to an image
 - rotate it with an angle between -15 and 15 degrees
 - apply a shift both in width and height up to $\pm 20\%$
 - apply a shear up to 5 degrees
 - apply a zoom between 0.8 and 1.2
 - possibly flip the image left to right
 - fill the borders with the nearest pixel

The `train_datagen` object contains the instructions for the transformation we want to apply, now we need to tell it where to source the images from. We'll use the very convenient `.flow_from_directory` method, specifying the path of the training images, the target size and the batch size:

```
train_generator = train_datagen.flow_from_directory(  
    train_path,  
    target_size=(img_size, img_size),  
    batch_size=batch_size)
```

```
Found 2100 images belonging to 3 classes.
```

As you can see, it found 2100 images with 3 classes. This is because the images are organized in 3 subfolders of the train path:

```
train_path  
| - Beach volleyball  
|   | - img1  
|   | - img2  
|   | - ...  
| - Cross-country skiing  
|   | - img1  
|   | - img2  
|   | - ...  
| - Formula racing  
|   | - img1  
|   | - img2  
|   | - ...
```

Now let's also create a generator for the test images. Note that the `test_path` must have the same subfolders in order to be compatible with the training set. We will not apply any transformation to test images and we will flow them as they are. The reason for this is to have reproducible test results:

```
test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
```

We will flow test images from the `test_path` directory:

```
test_generator = test_datagen.flow_from_directory(  
    test_path,
```

```
target_size=(img_size, img_size),  
batch_size=batch_size,)
```

```
Found 902 images belonging to 3 classes.
```

We are now ready to train the model with our generator. Since we are generating images with a generator, the concept of Epoch is no longer well defined. For this reason we will specify how many steps of update an epoch includes.

Let's see: we have 2100 images in the training folder and we feed batches of 32 images. This means that with approximately 65 steps we have sent roughly as many images as there are in the training set. Let's do this: we train the model for 1 epoch with 65 update steps:

```
model.fit_generator(  
    train_generator,  
    steps_per_epoch=65,  
    epochs=1)
```

```
Epoch 1/1  
65/65 [=====] - 53s 813ms/step - loss: 0.5376 - acc: 0.7894
```

```
<keras.callbacks.History at 0x7f0d8471ddd8>
```

It took a little bit of time but it looks really good. In a single epoch we have re-purposed a huge convolutional neural network that can now perform image recognition on new classes of images, never before encountered. Cool!

Let's assess the accuracy of our model with the `.evaluate_generator` method on the test set:

```
model.evaluate_generator(test_generator)
```

```
[0.15093981836428927, 0.9401330378261744]
```

Not bad at all, considering that it only trained for 1 epoch. Also, let's check the prediction on our original image of the volleyball player:

```
model.predict_classes(img_tensor)
```

```
array([2])
```

It's predicted to be in class 0 , which is the correct class. We can check that by looking at the `class_indices` defined in the `test_generator` :

```
train_generator.class_indices
```

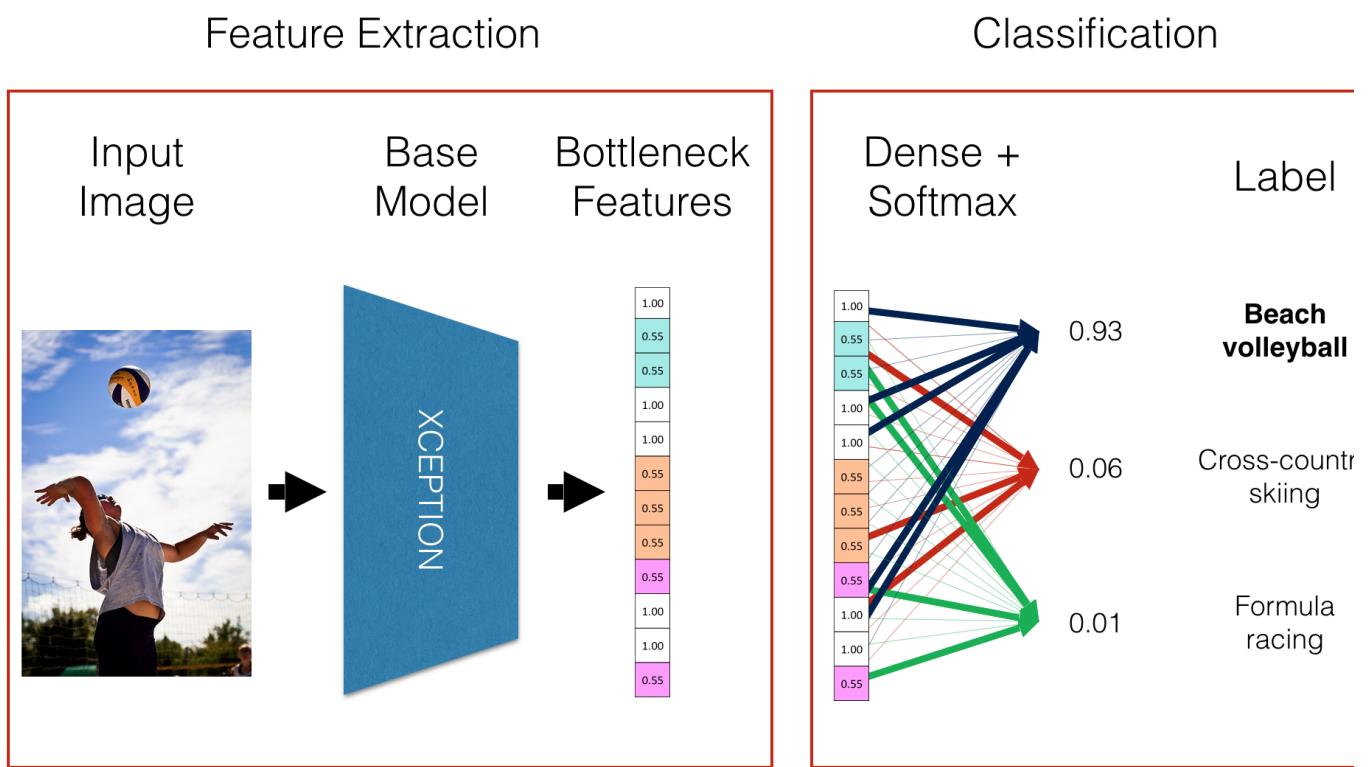
```
{'Beach volleyball': 0, 'Cross-country skiing': 1, 'Formula racing': 2}
```

Awesome! We've performed transfer learning for the first time and we've reused a giant pre-trained model for our goal. This was really good, but it did take a bit of a long time to train. Can we speed that up? The answer is yes! Let's introduce bottleneck features.

BOTTLENECK FEATURES

Let's stop for a second and think back about what we've just done. First we've loaded a large convolutional network, whose weights have been pre-trained on the Imagenet problem.

Then we've used the convolutional part of this network as the first layer of a new network, followed by fully connected layers. Since its weights are frozen, the convolutional part of the network is basically acting as a feature extractor, that extracts a vector of features from the given input image.



These features are then fed to the fully connected layers who perform the classification. The training process is still slow because all of the convolutions are applied to the image at each training step.

On the other hand, since the weights are frozen, we could take a different approach. We could pre-process all the images once: send them through the convolutional part of the network and extract a feature vector for each of them. We could use this dataset of feature vectors to train a fully connected network for the classification.

Another way to look at this process is to say that we are using the pre-trained network as a feature extraction pipeline, not dissimilar from traditional pipelines involving Wavelets, Histograms and so on. The difference here is that the bottleneck features are obtained through a network that's been trained on image classification on millions of images and are therefore optimized for that task.

Let's start by wrapping the `ImageDataGenerator` and the `.flow_from_directory` method in a single function. This function takes the `input_path` of the images and a couple of other parameters and returns a generator ready to receive all the images in the `input_path`. We'll feed this generator to the `base_model.predict_generator` function, which will return the values of the last layer before the output layer of the full model (remember we loaded the model with the parameter `include_top=False`). Also notice that we will set `shuffle=False` so that the image order is the same as that contained in `generator.classes` and we can later use it.

Here's the function:

```
def bottleneck_generator(input_path,
                        img_size=299,
                        batch_size=32,
                        shuffle=False):

    # ImageDataGenerator that applies preprocess_input to each image
    datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

    # return batches of preprocessed and scaled images
    # together with their labels. Images are taken
    # from input_path, labels are generated from the
    # subdir structure. In input_path there are
    # 3 subfolders, so there'll be 3 labels.
    generator = datagen.flow_from_directory(
        input_path,
        target_size=(img_size, img_size),
        batch_size=batch_size,
        class_mode='categorical',
        shuffle=shuffle)

    return generator
```

Let's use this function to generate the bottlenecks for the training images:

```
train_generator = bottleneck_generator(train_path)
bottlenecks_train = base_model.predict_generator(train_generator, verbose=1)
```

```
Found 2100 images belonging to 3 classes.  
66/66 [=====] - 33s 505ms/step
```

Let's also recover the training labels from the same generator.

```
labels_train = train_generator.classes
```

Depending on your system, generating bottlenecks may take a long time and having one or more GPU available will surely speed up the process. Since they are quite small, the code repository already contains a saved version of these bottlenecks.

Now that we have created the bottlenecks let's check what they look like. Let's start from the shape of the tensors:

```
bottlenecks_train.shape
```

```
(2100, 2048)
```

Train bottlenecks are a matrix with as many rows as there are images in the training set and with a number of columns equal to the number of outputs of the base model's last layer, i.e. the `GlobalAveragePooling` layer from `xception`, i.e. 2048 features. Let's plot a few of them to see what they look like. Let's get a bunch of images and labels from the train generator:

```
images, labels = bottleneck_generator(train_path, shuffle=True, batch_size=256).next()
```

```
Found 2100 images belonging to 3 classes.
```

And let's create a list with the label names for each of the images in the batch. We will do this in two steps. First let's create a label map with the label names corresponding to the class indices:

```
label_map = list(train_generator.class_indices.keys())  
label_map
```

```
['Beach volleyball', 'Cross-country skiing', 'Formula racing']
```

Then let's use the `label_map` to convert the labels into the corresponding label names:

```
label_names = [label_map[i] for i in labels.argmax(axis=1)]  
label_names[:10]
```

```
['Formula racing',  
'Cross-country skiing',  
'Cross-country skiing',  
'Cross-country skiing',  
'Formula racing',  
'Formula racing',  
'Beach volleyball',  
'Beach volleyball',  
'Beach volleyball',  
...  
'Beach volleyball']
```

Great. Now let's generate bottleneck features for the images in the batch:

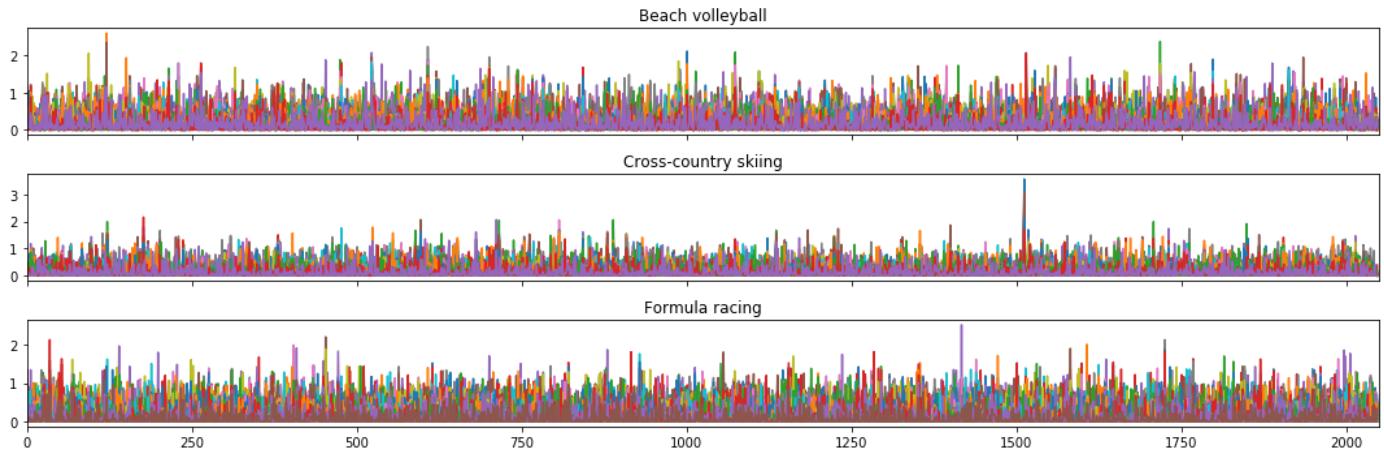
```
bottlenecks = base_model.predict(images, verbose=1)
```

```
256/256 [=====] - 4s 15ms/step
```

Will bottlenecks of images with the same label be similar? Let's take a look at them on a plot and see if there's any pattern we can recognize. We will create a figure with three plots, one for each of the three classes, and plot the values of the bottlenecks:

```
fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True, figsize=(15, 5))  
  
for bn, label in zip(bottlenecks, label_names):  
    idx = train_generator.class_indices[label]  
    ax[idx].plot(bn)  
    ax[idx].set_title(label)
```

```
plt.xlim(0, 2050)
plt.tight_layout()
```

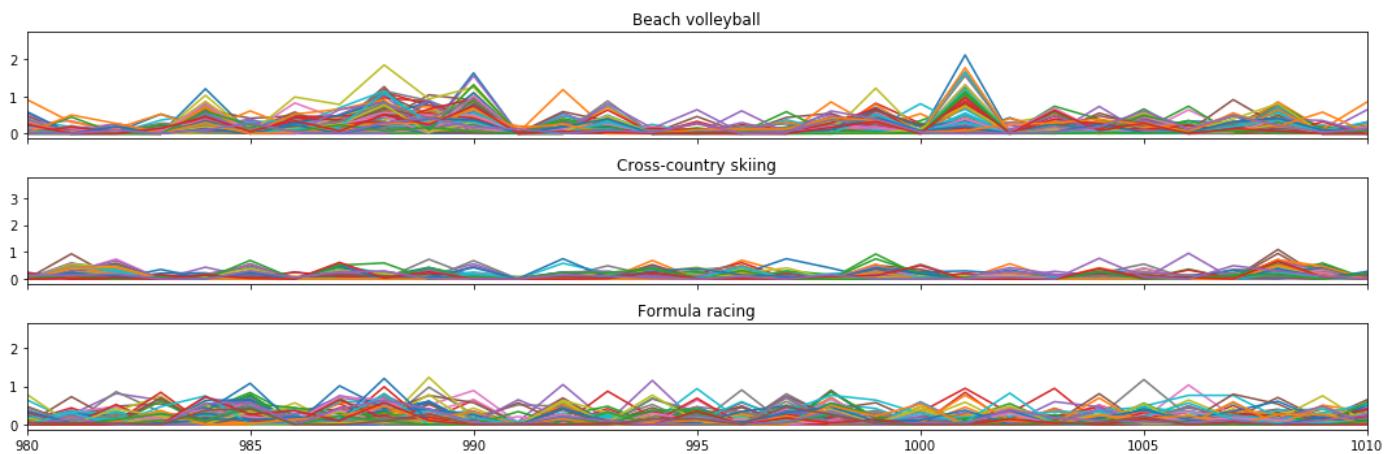


Hmm, although the 3 plots look somewhat different, it's hard to tell if there's anything interesting. Let's zoom in to the range 980-1010:

```
fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True, figsize=(15, 5))

for bn, label in zip(bottlenecks, label_names):
    idx = train_generator.class_indices[label]
    ax[idx].plot(bn)
    ax[idx].set_title(label)

plt.xlim(980, 1010)
plt.tight_layout()
```



Here it's clearer that several of the Beach Volleyball bottlenecks have a high spike at features 984, 990 and 1001, while the other two sports do not have those peaks. Bottlenecks are like fingerprints of an image: features extracted through convolutions that encode the content of the image.

Now that we understand a little more what bottleneck features are, we can save them to disk once and for all. We can now experiment with fully connected architectures that classify the bottlenecks as input data. Let's save the bottlenecks and the labels as numpy arrays.

We will use the `gzip` library for efficiency.

```
import gzip
```

```
np.save(gzip.open('../data/sports/bottlenecks_train.npy.gz', 'wb'), bottlenecks_train)
```

```
np.save(open('../data/sports/labels_train.npy', 'wb'), labels_train)
```

Let's also generate the bottlenecks for the test set:

```
test_generator = bottleneck_generator(test_path)
bottlenecks_test = base_model.predict_generator(test_generator, verbose=1)
```

```
Found 902 images belonging to 3 classes.
29/29 [=====] - 14s 495ms/step
```

and the test labels:

```
labels_test = test_generator.classes
```

and let's save them too:

```
np.save(gzip.open('../data/sports/bottlenecks_test.npy.gz', 'wb'), bottlenecks_test)
```

```
np.save(open('../data/sports/labels_test.npy', 'wb'), labels_test)
```

Great! Now let's see how we use the bottlenecks.

TRAIN A FULLY CONNECTED ON BOTTLENECKS

Bottlenecks saved to disk can be restored by reading them. Let's read them into two variables called `x_train` and `x_test`.

```
x_train = np.load(gzip.open('../data/sports/bottlenecks_train.npy.gz', 'rb'))
x_test = np.load(gzip.open('../data/sports/bottlenecks_test.npy.gz', 'rb'))
```

We can check the shape of the train data, and verify that it's a matrix:

```
x_train.shape
```

```
(2100, 2048)
```

Similarly, let's load the labels:

```
y_train = np.load(open('../data/sports/labels_train.npy', 'rb'))
y_test = np.load(open('../data/sports/labels_test.npy', 'rb'))
```

and check the shape as well:

```
y_train.shape
```

```
(2100, )
```

It looks like we have to one-hot encode the labels, so let's do that using the `keras.utils.to_categorical` function that we have used many times in the book:

```
from keras.utils import to_categorical
```

```
y_train_cat = to_categorical(y_train)
y_test_cat = to_categorical(y_test)
```

Now we are finally ready to train a fully connected network on the bottleneck features. Let's import the `Sequential` model and `Dense` and `Dropout` layers.

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
```

We'll define the model using the sequential API. We will build a very simple model with just 2 layers: a `Dropout` layer as input, that will receive the bottleneck features and a `Dense` output layer for the output. The output layer must have 3 nodes because there are 3 classes and it must have a `softmax` activation function because the classes are mutually exclusive. Feel free to change the model definition to something else if you'd like, keeping in mind that we only have a few thousand training data points so giving the model too much freedom may lead to overfitting.

Notice that instead of adding the layers like we did in other parts of the book we can pass a list of layers to the model constructor:

```
fc_model = Sequential([
    Dropout(0.5, input_shape=(2048,)),
    Dense(16, activation='relu'),
    Dropout(0.5),
    Dense(3, activation='softmax')
])
```

Let's compile the model with our preferred optimizer using the `categorical_crossentropy` loss:

```
fc_model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

Here's a summary of the model:

```
fc_model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
dropout_2 (Dropout)	(None, 2048)	0
dense_3 (Dense)	(None, 16)	32784
dropout_3 (Dropout)	(None, 16)	0
<hr/>		
<hr/>		

This simple model has a little over 6000 parameters, so it will be very fast to train. Let's train it for a few epochs:

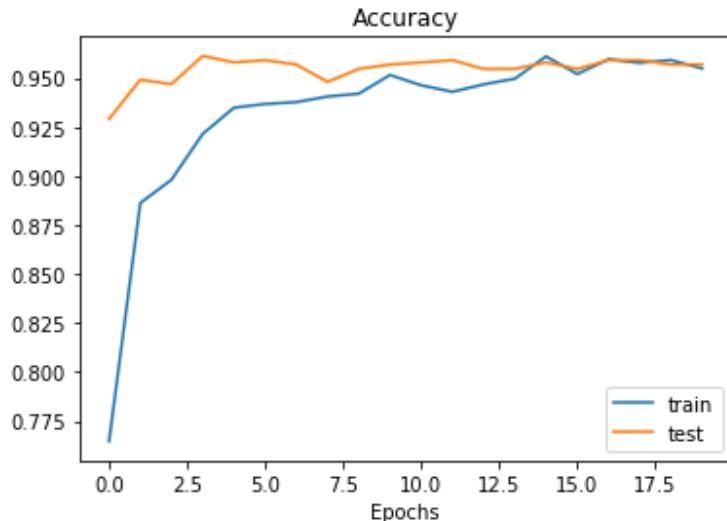
```
history = fc_model.fit(X_train, y_train_cat,
                       epochs=20,
                       batch_size=batch_size,
                       validation_data=(X_test, y_test_cat))
```

```
Train on 2100 samples, validate on 902 samples
Epoch 1/20
2100/2100 [=====] - 1s 654us/step - loss: 0.5420 - acc: 0.7648 -
Epoch 2/20
2100/2100 [=====] - 0s 106us/step - loss: 0.3491 - acc: 0.8862 -
Epoch 3/20
2100/2100 [=====] - 0s 106us/step - loss: 0.2794 - acc: 0.8981 -
Epoch 4/20
2100/2100 [=====] - 0s 106us/step - loss: 0.2516 - acc: 0.9214 -
...
2100/2100 [=====] - 0s 106us/step - loss: 0.2516 - acc: 0.9214 -
```

And let's plot the accuracy:

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Accuracy')
plt.legend(['train', 'test'])
plt.xlabel('Epochs')
```

```
Text(0.5, 0, 'Epochs')
```



This model trained really fast and the accuracy on the test set is higher than the accuracy on the training set, which is a really good sign. This is the value of bottleneck features, we can use them as proxies for our original images and train a simple model using them using a laptop.

Image search

A fun application of pre-trained models is image search.

Imagine the following situation: we have a dataset of images and we would like to find the most similar images to a specific one, for example we have our collection of pictures on our laptop and we'd like to find all the pictures of a friend.

Solving this problem requires the definition of a distance measure between images, so that, given an image, we can look for images that are close to it. This is hard to do using the raw pixels as features because, as we have seen many times, images with similar content may look completely different on every single pixel.

On the other hand, since bottleneck features capture high level features from the images, we exploit them to locate similar images. We will do this using the `DistanceMetric` class from Scikit Learn. Let's start by importing it:

```
from sklearn.neighbors import DistanceMetric
```

and let's load an instance of the Euclidean metric, which is the usual vector difference distance:

```
dist = DistanceMetric.get_metric('euclidean')
```

We have used to define the Mean Squared Error in Chapter 3 and it is obtained through the sum of the squares of the differences along each coordinate:

$$d(\mathbf{x}', \mathbf{x}) = \sqrt{(\mathbf{x}' - \mathbf{x})^2} = \sqrt{\sum_i (x'_i - x_i)^2}$$

and given the two vectors:

```
a = np.array([1, 2])
```

```
b = np.array([2, -1])
```

it is calculated as:

```
np.sqrt(np.square(a - b).sum())
```

3.1622776601683795

Now that we have defined the euclidean distance metric we can calculate the pairwise distances between all the bottlenecks and then use that for our image search engine.

Let's take a few images as example. Let's get the training images from the training set using the bottleneck data generator:

```
images, labels = bottleneck_generator(train_path, batch_size=2100).next()
```

Found 2100 images belonging to 3 classes.

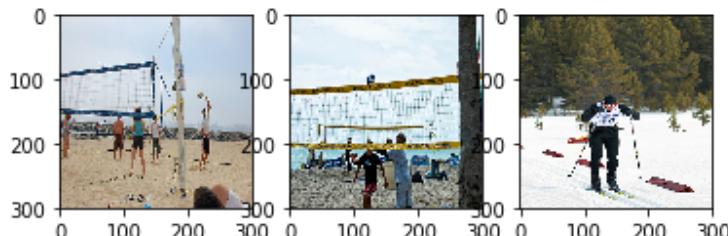
And let's display a few of them. Note that since our images have been normalized during pre-processing, their pixels are now values between -1 and 1. `plt.imshow` requires floating point images to have values between 0 and 1 so we will need to add 1 and divide by 2 each pre-processed image in order to display it correctly. Let's define a helper function that does that.

```
def imshow_scaled(img):
    plt.imshow((img + 1) / 2)
```

```
plt.subplot(1, 3, 1)
imshow_scaled(images[0])

plt.subplot(1, 3, 2)
imshow_scaled(images[1])

plt.subplot(1, 3, 3)
imshow_scaled(images[900])
```



The first two are images of beach volleyball while the third one is of skiing.

The distance between the first and the second image is:

```
np.sqrt(np.square(X_train[0] - X_train[1]).sum())
```

6.608992

while the distance between the first and the third is:

```
np.sqrt(np.square(X_train[0] - X_train[900]).sum())
```

```
10.535508
```

As you can see, the first and the third image are further apart, which makes sense since the last one is very different from the previous two. We will proceed now to calculate all the distances between all of the images in the training set using their bottleneck features.

We will do this calculation using the `.pairwise` method of the euclidean distance object we have created previously. We could also do a double for loop over bottlenecks and calculate the distances manually, however, this is more efficient:

```
bn_dist = dist.pairwise(X_train)
```

Let's check the shape of the matrix we have obtained:

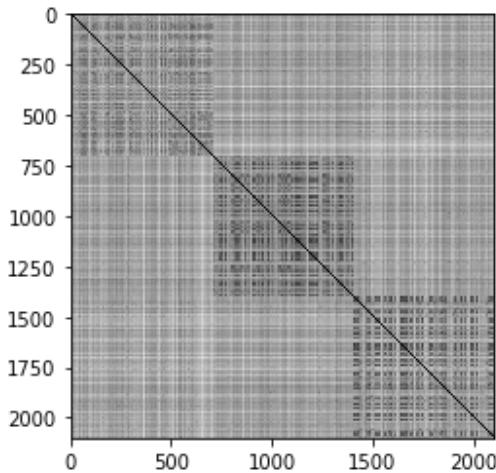
```
bn_dist.shape
```

```
(2100, 2100)
```

Since we have 2100 images in the training set, the pairwise matrix is a square simmetric matrix that contains all pairwise distances. Let's visualize it to understand it a little bit better:

```
plt.imshow(bn_dist, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f0bb8113ac8>
```



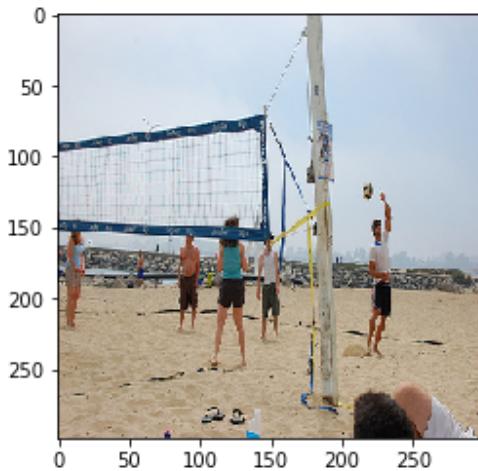
Notice a couple of things about this matrix:

1. The darker a pixel the closer two corresponding images are.
- The matrix is symmetric with respect to the diagonal, which makes sense since the distance between image 1 and image 2 is the same as the distance between image 2 and image 1
- The diagonal is the darkest of all, which also makes sense, since an image will be identical to itself and therefore have a distance of zero from itself
- Three blocks are clearly distinguishable along the diagonal, although a little fuzzy. This makes a lot of sense, because images are sorted by class and generally speaking all the images in a class are expected to be more similar to one another than to images in other classes

Notice that we have obtained these distances using the bottlenecks from the pre-trained model, no additional training needed. Awesome!

Let's put this to use in our search engine! Let's take an image:

```
imshow_scaled(images[0])
```



Let's look for the top 9 closest images. All we have to do is select the row in the `bn_dist` matrix corresponding to the index of the image selected, which is zero in this case. We will wrap this with a Pandas Series so that we can use the indices later:

```
distances_from_selected = pd.Series(bn_dist[0])
```

Let's sort these and display the first few images:

```
distances_from_selected.sort_values().head(9)
```

```
0      0.000000
431    5.348792
15     5.404698
32     5.787818
636    5.809687
648    5.847936
429    5.848352
48     5.952393
656    5.967599
...
dtype: float64
```

Let's display these. We will display 9 images, in a grid of 3x3. Let's make this configurable by defining a few parameters:

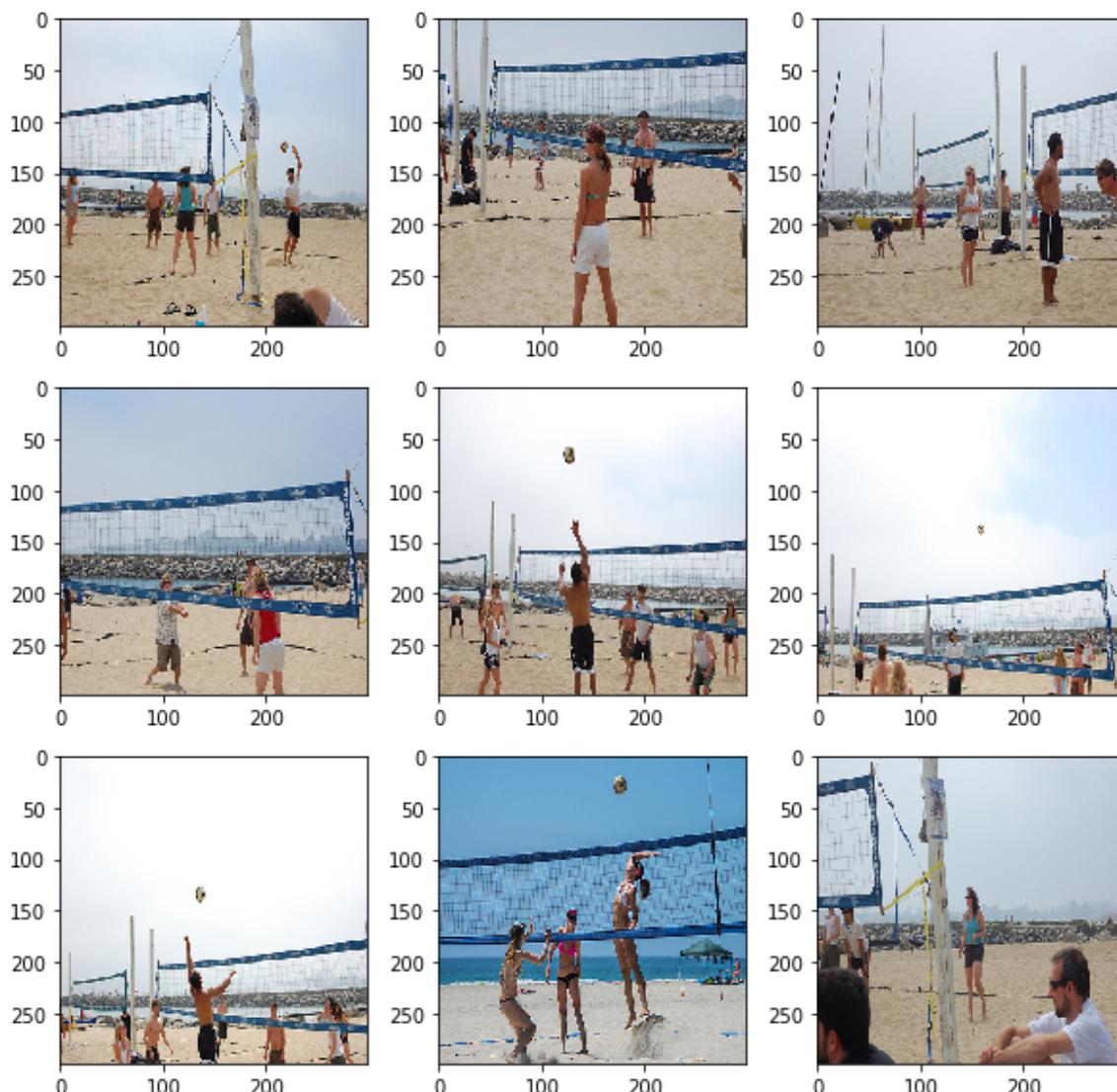
```
n_rows = 3
n_cols = 3
n_images = n_rows * n_cols
```

Now let's take the top 9 images with the shortest distance from our original image. This can be done by sorting the values in `distances_from_selected` and then using the `.head` command that retrieves the first elements in the series:

```
retrieved = distances_from_selected.sort_values().head(n_images)
```

Now let's loop over the index of the retrieved images and plot the images:

```
plt.figure(figsize=(10, 10))
i = 1
for idx in retrieved.index:
    plt.subplot(n_rows, n_cols, i)
    imshow_scaled(images[idx])
    i += 1
```



Nice! The first image displayed is the one we had selected, and as you can see the other ones are all very similar! Let's try again with another image. We will define a function to make things easy:

```
def image_search(img_index, n_rows=3, n_columns=3):
    n_images = n_rows * n_columns

    # create Pandas Series with distances from image
    distances_from_selected = pd.Series(bn_dist[img_index])

    # sort Series and get top n_images
    retrieved = distances_from_selected.sort_values().head(n_images)

    # create figure, loop over closest images indices
```

```
# and display them
plt.figure(figsize=(10, 10))
i = 1
for idx in retrieved.index:
    plt.subplot(n_rows, n_cols, i)
    imshow_scaled(images[idx])
    if i == 1:
        plt.title('Selected image')
    else:
        plt.title("Dist: {:.4f}".format(retrieved[idx]))
    i += 1
```

```
image_search(900)
```



```
image_search(1600)
```

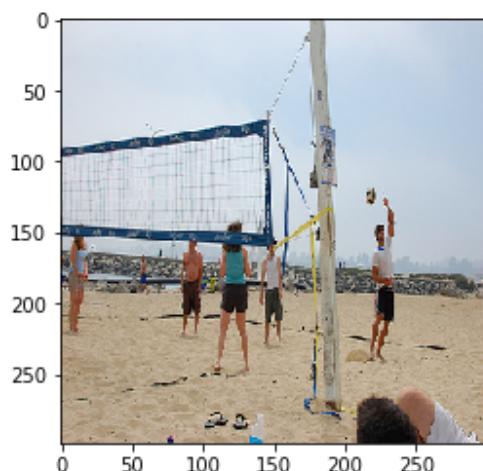


```
image_search(100)
```



Notice that we can also sort the distances in reverse order and find the images which are the furthest away from a selected image. E.g., for this image:

```
imshow_scaled(images[0])
```



The most distant images are:

```
retrieved = pd.Series(bn_dist[0]).sort_values(ascending=False).head(9)
```

```
plt.figure(figsize=(10, 10))
i = 1
for idx in retrieved.index:
    plt.subplot(n_rows, n_cols, i)
    imshow_scaled(images[idx])
    plt.title("Dist: {:.4f}".format(retrieved[idx]))
    i += 1
```



Which clearly have very little in common with the above image!

In conclusion, Keras offers several pre-trained models for images, that can be used for a variety of tasks including image recognition, transfer learning and image similarity search.

EXERCISES

Exercise 1

Use a pre-trained model on a different image.

- Download an image from the web
- Upload the image through the jupyter home page
- load the image as a numpy array
- re-run the pre-train to see if the pre-trained model can guess your image
- can you find an image that is outside of the Imagenet classes? (you can see which classes are available here.)

Exercise 2

Choose another pre-trained model from the ones provided at <https://keras.io/applications/> and use it to predict the same image. Do the predictions match?

Exercise 3

The [Keras documentation](#) shows how to fine-tune the Inception V3 model by unfreezing some of the convolutional layers. Try reproducing the results of the documentation on our dataset using the Xception model and unfreezing some of the top convolutional layers.