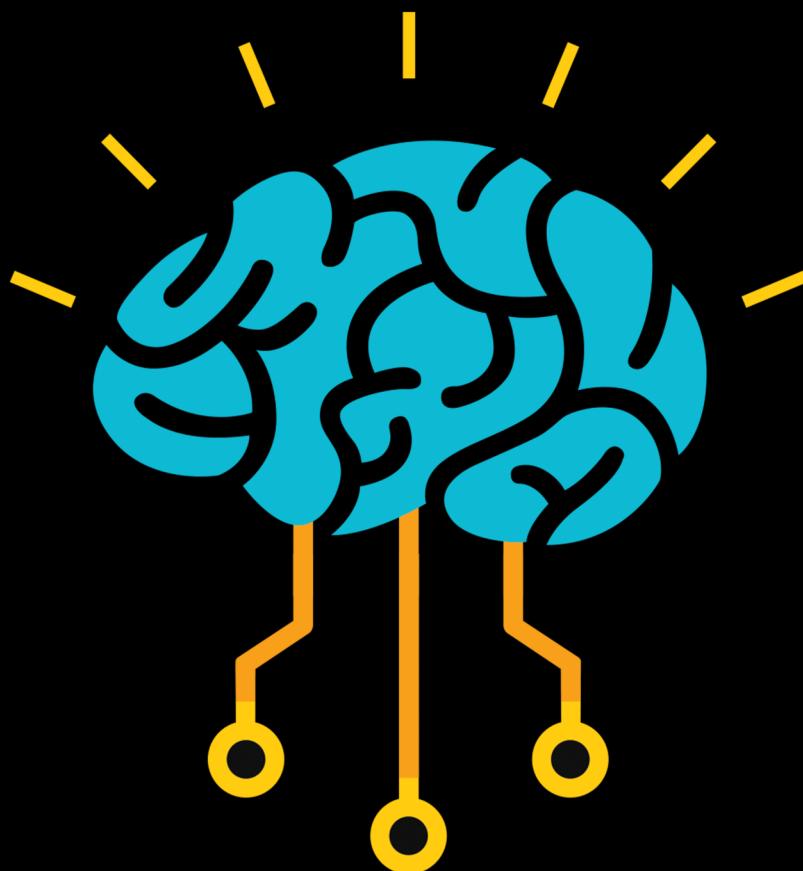




# ZERO TO DEEP LEARNING WITH KERAS AND TENSORFLOW

DEEP LEARNING  
FOR THE REST  
OF US



# FRANCESCO MOSCONI

EDITORS: NATE MURRAY, ARI LERNER



# Zero to Deep Learning

Francesco Mosconi

September 7, 2018

Copyright © 2018 Francesco Mosconi. All rights reserved. Printed in the United States of America

Published by Fullstack.io

Editors: Nate Murray and Ari Lerner

September 2018: vo.9.3

Zero to Deep Learning is a registered trademark of Catalit LLC.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Catalit LLC was aware of a trademark claim, the designations have been printed.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

*Dedicated to our Future Selves. May we take care of one another  
and the world making good use of Artificial Intelligence. And to my  
nieces, who will inherit the future we build today.*



# Table of Contents

<b>o</b>	<b>Introduction</b>	<b>1</b>
o.1	Welcome . . . . .	1
o.2	Acknowledgements . . . . .	1
o.3	About the author . . . . .	2
o.4	How to use this book . . . . .	2
o.4.1	Exercises . . . . .	3
o.4.2	Notation . . . . .	3
o.5	Prerequisites - Is this book right for me? . . . . .	4
o.5.1	Sum . . . . .	4
o.5.2	Partial derivatives . . . . .	5
o.5.3	Dot product . . . . .	5
o.5.4	Python . . . . .	5
o.6	Our development environment . . . . .	6
o.6.1	Miniconda Python . . . . .	6
o.6.2	Conda Environment . . . . .	7
o.6.3	GPU enabled environment . . . . .	9
o.6.4	Jupyter notebook . . . . .	9
o.6.5	Environment check . . . . .	12
o.6.6	Python 3.6 . . . . .	14
o.6.7	Jupyter . . . . .	15
o.6.8	Other packages . . . . .	15
o.6.9	Troubleshooting installation . . . . .	17
o.6.10	Updating Conda . . . . .	18
<b>1</b>	<b>Getting Started</b>	<b>19</b>
1.1	Deep Learning in the real world . . . . .	19
1.2	First Deep Learning Model . . . . .	22
1.2.1	Numpy . . . . .	22
1.2.2	Matplotlib . . . . .	32

1.2.3	Scikit-Learn . . . . .	38
1.2.4	Keras . . . . .	41
1.3	Exercises . . . . .	51
1.3.1	Exercise 1 . . . . .	51
1.3.2	Exercise 2 . . . . .	51
1.3.3	Exercise 3 . . . . .	51
1.3.4	Exercise 4 . . . . .	51
<b>2</b>	<b>Data Manipulation</b>	<b>53</b>
2.1	Many types of data . . . . .	53
2.1.1	Tabular Data . . . . .	54
2.2	Data Exploration with Pandas . . . . .	55
2.2.1	Indexing . . . . .	57
2.2.2	Selections . . . . .	59
2.2.3	Unique Values . . . . .	61
2.2.4	Sorting . . . . .	61
2.2.5	Aggregations . . . . .	61
2.2.6	Merge . . . . .	64
2.2.7	Pivot Tables . . . . .	65
2.2.8	Correlations . . . . .	66
2.3	Visual data exploration . . . . .	68
2.3.1	Line Plot . . . . .	69
2.3.2	Scatter plot . . . . .	70
2.3.3	Histograms . . . . .	72
2.3.4	Cumulative Distribution . . . . .	73
2.3.5	Box plot . . . . .	74
2.3.6	Subplots . . . . .	76
2.3.7	Pie charts . . . . .	77
2.3.8	Hexbin plot . . . . .	78
2.4	Unstructured data . . . . .	80
2.4.1	Images . . . . .	81
2.4.2	Sound . . . . .	82
2.4.3	Text data . . . . .	84
2.5	Feature Engineering . . . . .	84
2.6	Exercises . . . . .	85
2.6.1	Exercise 1 . . . . .	85
2.6.2	Exercise 2 . . . . .	85
2.6.3	Exercise 3 . . . . .	85

2.6.4	Exercise 4 . . . . .	85
2.6.5	Exercise 5 . . . . .	86
<b>3</b>	<b>Machine Learning</b>	<b>87</b>
3.1	The purpose of Machine Learning . . . . .	87
3.2	Different types of learning . . . . .	90
3.3	Supervised Learning . . . . .	91
3.4	Configuration File . . . . .	92
3.5	Linear Regression . . . . .	93
3.5.1	Let's draw some examples. . . . .	96
3.5.2	Cost Function . . . . .	100
3.5.3	Finding the best model . . . . .	102
3.5.4	Linear Regression with Keras . . . . .	106
3.5.5	Evaluating Model Performance . . . . .	110
3.5.6	Train / Test split . . . . .	111
3.6	Classification . . . . .	113
3.6.1	Linear regression fail . . . . .	115
3.6.2	Logistic Regression . . . . .	117
3.6.3	Train/Test split . . . . .	128
3.7	Overfitting . . . . .	130
3.7.1	How to avoid overfitting . . . . .	130
3.8	Cross Validation . . . . .	132
3.9	Confusion Matrix . . . . .	134
3.9.1	Precision . . . . .	136
3.9.2	Recall . . . . .	136
3.9.3	F1 Score . . . . .	137
3.10	Feature Preprocessing . . . . .	138
3.10.1	Categorical Features . . . . .	138
3.10.2	Feature Transformations . . . . .	139
3.11	Exercises . . . . .	142
3.11.1	Exercise 1 . . . . .	142
3.11.2	Exercise 2 . . . . .	143
<b>4</b>	<b>Deep Learning</b>	<b>145</b>
4.1	Beyond linear models . . . . .	145
4.2	Neural Network Diagrams . . . . .	147

4.2.1	Linear regression . . . . .	147
4.2.2	Logistic regression . . . . .	148
4.2.3	Perceptron . . . . .	149
4.2.4	Deeper Networks . . . . .	151
4.3	Activation functions . . . . .	154
4.3.1	Tanh . . . . .	155
4.3.2	ReLU . . . . .	156
4.3.3	Softplus . . . . .	157
4.3.4	SeLU . . . . .	158
4.4	Binary classification . . . . .	160
4.4.1	Logistic Regression . . . . .	162
4.4.2	Deep model . . . . .	165
4.5	Multiclass classification . . . . .	170
4.5.1	Tags . . . . .	170
4.5.2	Mutually exclusive classes . . . . .	170
4.5.3	The Iris dataset . . . . .	171
4.6	Conclusion . . . . .	178
4.7	Exercises . . . . .	178
4.7.1	Exercise 1 . . . . .	178
4.7.2	Exercise 2 . . . . .	179
4.7.3	Exercise 3 . . . . .	180
4.7.4	Exercise 4 . . . . .	180
<b>5</b>	<b>Deep Learning Internals</b>	<b>181</b>
5.1	This is a special chapter . . . . .	181
5.2	Derivatives . . . . .	182
5.2.1	Finite differences . . . . .	185
5.2.2	Partial derivatives and the gradient . . . . .	188
5.3	Backpropagation intuition . . . . .	189
5.4	Learning Rate . . . . .	191
5.5	Gradient descent . . . . .	192
5.6	Gradient calculation in Neural Networks . . . . .	192
5.7	The math of backpropagation . . . . .	196
5.7.1	Forward Pass . . . . .	197
5.7.2	Weight updates . . . . .	198

5.8	Fully Connected Backpropagation . . . . .	203
5.8.1	Forward Pass . . . . .	203
5.8.2	Backpropagation . . . . .	205
5.9	Matrix Notation . . . . .	206
5.9.1	Forward Pass . . . . .	206
5.9.2	Backpropagation . . . . .	206
5.10	Gradient descent . . . . .	207
5.10.1	Random Forest . . . . .	212
5.10.2	Logistic Regression Model . . . . .	213
5.10.3	Learning Rates . . . . .	218
5.10.4	Batch Sizes . . . . .	221
5.11	Optimizers . . . . .	223
5.11.1	Stochastic Gradient Descent (or Simply Go Down) and its variations . . . . .	224
5.12	Initialization . . . . .	229
5.13	Inner layer representation . . . . .	231
5.14	Exercises . . . . .	236
5.14.1	Exercise 1 . . . . .	236
5.14.2	Exercise 2 . . . . .	237
5.14.3	Exercise 3 . . . . .	237
5.14.4	Exercise 4 . . . . .	238
<b>6</b>	<b>Convolutional Neural Networks</b> . . . . .	<b>239</b>
6.1	Intro . . . . .	239
6.2	Machine Learning on images with pixels . . . . .	239
6.2.1	MNIST . . . . .	244
6.2.2	Pixels as features . . . . .	247
6.2.3	Multiclass output . . . . .	248
6.2.4	Fully connected on images . . . . .	250
6.3	Beyond pixels as features . . . . .	255
6.3.1	Using local information . . . . .	256
6.3.2	Images as tensors . . . . .	257
6.3.3	Colored images . . . . .	259
6.4	Convolutional Neural Networks . . . . .	263
6.4.1	Convolutional Layers . . . . .	265
6.4.2	Pooling layers . . . . .	274
6.4.3	Final architecture . . . . .	276
6.4.4	Convolutional network on images . . . . .	277

6.5	Beyond images . . . . .	281
6.6	Conclusion . . . . .	281
6.7	Exercise . . . . .	281
6.7.1	Exercise 1 . . . . .	281
6.7.2	Exercise 2 . . . . .	282
<b>7</b>	<b>Time Series and Recurrent Neural Networks</b>	<b>283</b>
7.1	Time Series . . . . .	283
7.2	Time series classification . . . . .	286
7.2.1	Fully connected networks . . . . .	289
7.2.2	Fully connected networks with feature engineering . . . . .	291
7.2.3	Fully connected networks with 1D Convolution . . . . .	293
7.3	Sequence Problems . . . . .	296
7.3.1	1-to-1 . . . . .	296
7.3.2	1-to-many . . . . .	296
7.3.3	many-to-1 . . . . .	298
7.3.4	asynchronous many-to-many . . . . .	298
7.3.5	synchronous many-to-many . . . . .	298
7.3.6	RNN allow graphs with cycles . . . . .	298
7.4	Time series forecasting . . . . .	299
7.4.1	Fully connected network . . . . .	304
7.4.2	Recurrent Neural Networks . . . . .	308
7.4.3	Recurrent Neural Network Maths . . . . .	315
7.4.4	Long Short-Term Memory Networks (LSTM) . . . . .	320
7.4.5	LSTM forecasting . . . . .	325
7.5	Improving forecasting . . . . .	328
7.5.1	Conclusion . . . . .	333
7.6	Exercises . . . . .	334
7.6.1	Exercise 1 . . . . .	334
7.6.2	Exercise 2 . . . . .	334
7.6.3	Exercise 3 . . . . .	335
7.6.4	Exercise 4 . . . . .	335
<b>8</b>	<b>Natural Language Processing and Text Data</b>	<b>339</b>
8.1	Use cases . . . . .	339
8.2	Text Data . . . . .	341
8.2.1	Loading text data . . . . .	341

8.2.2	Feature extraction from text . . . . .	346
8.2.3	Bag of Words features . . . . .	349
8.2.4	Sentiment classification . . . . .	355
8.2.5	Text as a sequence . . . . .	359
8.3	Sequence generation and language modeling . . . . .	372
8.3.1	Character sequences . . . . .	374
8.3.2	Recurrent Model . . . . .	376
8.3.3	Sampling from the model . . . . .	380
8.3.4	Sequence to sequence models and language translation . . . . .	386
8.4	Exercises . . . . .	387
8.4.1	Exercise 1 . . . . .	387
8.4.2	Exercise 2 . . . . .	387
<b>9</b>	<b>Training with GPUs</b>	<b>389</b>
9.1	Graphical Processing Units . . . . .	389
9.2	Cloud GPU providers . . . . .	391
9.2.1	Google Colab . . . . .	391
9.2.2	Pipeline AI . . . . .	392
9.2.3	Floydhub . . . . .	394
9.2.4	Paperspace . . . . .	397
9.2.5	AWS EC2 Deep Learning AMI . . . . .	398
9.2.6	AWS Sagemaker . . . . .	407
9.2.7	Google Cloud and Microsoft Azure . . . . .	409
9.2.8	The DIY solution (on Ubuntu) . . . . .	409
9.3	GPU VS CPU training . . . . .	410
9.3.1	Convolutional model comparison . . . . .	410
9.4	Multiple GPUs . . . . .	414
9.4.1	Data Parallelization . . . . .	414
9.5	Conclusion . . . . .	417
9.6	Exercises . . . . .	417
9.6.1	Exercise 1 . . . . .	417
9.6.2	Exercise 2 . . . . .	418
<b>10</b>	<b>Performance Improvement</b>	<b>419</b>
10.1	Learning curves . . . . .	420
10.2	Reducing Overfitting . . . . .	432
10.2.1	Model Regularization . . . . .	437

10.2.2	Dropout . . . . .	441
10.2.3	Batch Normalization . . . . .	446
10.3	Data augmentation . . . . .	450
10.4	Hyperparameter optimization . . . . .	459
10.4.1	Hyperopt and Hyperas . . . . .	459
10.4.2	Cloud based tools . . . . .	460
10.5	Exercises . . . . .	460
10.5.1	Exercise 1 . . . . .	460
<b>11</b>	<b>Pretrained Models for Images</b>	<b>461</b>
11.1	Recognizing sports from images . . . . .	462
11.2	Keras applications . . . . .	467
11.3	Predict class with pre-trained Xception . . . . .	470
11.4	Transfer Learning . . . . .	481
11.5	Data augmentation . . . . .	484
11.6	Bottleneck features . . . . .	487
11.7	Train a fully connected on bottlenecks . . . . .	493
11.7.1	Image search . . . . .	496
11.8	Exercises . . . . .	508
11.8.1	Exercise 1 . . . . .	508
11.8.2	Exercise 2 . . . . .	509
11.8.3	Exercise 3 . . . . .	509
<b>12</b>	<b>Pretrained Embeddings for Text</b>	<b>511</b>
12.1	“Unsupervised”-“supervised learning” . . . . .	512
12.2	GloVe embeddings . . . . .	513
12.3	Loading pre-trained embeddings in Keras . . . . .	518
12.4	Gensim . . . . .	521
12.4.1	Word Analogies . . . . .	523
12.5	Visualization . . . . .	523
12.6	Other pre-trained embeddings . . . . .	528
12.6.1	Word2Vec . . . . .	528
12.6.2	FastText . . . . .	530
12.7	Exercises . . . . .	530

12.7.1	Exercise 1 . . . . .	530
12.7.2	Exercise 2 . . . . .	531
<b>13</b>	<b>Serving Deep Learning Models</b>	<b>533</b>
13.1	The model development cycle . . . . .	533
13.1.1	Data Collection . . . . .	534
13.1.2	Labels . . . . .	536
13.1.3	Data Processing . . . . .	536
13.1.4	Model Development . . . . .	537
13.1.5	Model Evaluation . . . . .	537
13.1.6	Model Exporting . . . . .	538
13.1.7	Model Deployment . . . . .	539
13.1.8	Model Monitoring . . . . .	540
13.2	Deploy a model to predict indoor location . . . . .	540
13.2.1	Data exploration . . . . .	541
13.2.2	Model definition and training . . . . .	543
13.2.3	Export the model with Keras . . . . .	546
13.3	A simple deployment with Flask . . . . .	552
13.3.1	Full script . . . . .	555
13.3.2	Run the script . . . . .	557
13.3.3	Get Predictions from the API . . . . .	558
13.4	Deployment with Tensorflow Serving . . . . .	560
13.4.1	Saving a model for Tensorflow Serving . . . . .	560
13.4.2	Inference with Tensorflow Serving using Docker and the Rest API . . . . .	563
13.4.3	The gRPC API . . . . .	564
13.5	Tensorflow Serving installation . . . . .	569
13.6	Exercises . . . . .	571
13.6.1	Exercise 1 . . . . .	571
13.6.2	Exercise 2 . . . . .	572
<b>14</b>	<b>Appendix</b>	<b>575</b>
14.1	Matrix multiplication . . . . .	575
14.2	Chain rule . . . . .	578
14.2.1	Univariate functions . . . . .	578
14.2.2	Multivariate functions . . . . .	581
14.2.3	Exponentially Weighted Moving Average (EWMA) . . . . .	581
14.3	Tensors . . . . .	586

14.3.1	Tensor Dot Product . . . . .	589
14.4	Convolutions . . . . .	591
14.4.1	1D Convolution & Correlation . . . . .	591
14.4.2	2D Convolution . . . . .	594
14.4.3	Image filters with convolutions . . . . .	600
14.5	Backpropagation for Recurrent Networks . . . . .	605
<b>15</b>	<b>Getting Started Exercises Solutions</b>	<b>607</b>
15.1	Exercise 1 . . . . .	607
15.2	Exercise 2 . . . . .	609
15.3	Exercise 3 . . . . .	614
15.4	Exercise 4 . . . . .	615
<b>16</b>	<b>Data Manipulation Exercises Solutions</b>	<b>619</b>
16.1	Exercise 1 . . . . .	619
16.2	Exercise 2 . . . . .	621
16.3	Exercise 3 . . . . .	626
16.4	Exercise 4 . . . . .	629
16.5	Exercise 5 . . . . .	630
<b>17</b>	<b>Machine Learning Exercises Solutions</b>	<b>633</b>
17.1	Exercise 1 . . . . .	633
17.2	Exercise 2 . . . . .	636
<b>18</b>	<b>Deep Learning Exercises Solutions</b>	<b>645</b>
18.1	Exercise 1 . . . . .	645
18.2	Exercise 2 . . . . .	650
18.3	Exercise 3 . . . . .	652
<b>19</b>	<b>Deep Learning Internals Exercises Solutions</b>	<b>655</b>
19.1	Exercise 1 . . . . .	655
19.2	Exercise 2 . . . . .	659
19.3	Exercise 3 . . . . .	661
19.4	Exercise 4 . . . . .	663

<b>20 Convolutional Neural Networks Exercises Solutions</b>	<b>665</b>
20.1 Exercise 1 . . . . .	665
20.2 Exercise 2 . . . . .	668
<b>21 Time Series and Recurrent Neural Networks Exercises Solutions</b>	<b>673</b>
21.1 Exercise 1 . . . . .	673
21.2 Exercise 2 . . . . .	676
21.3 Exercise 3 . . . . .	677
21.4 Exercise 4 . . . . .	678
<b>22 Natural Language Processing and Text Data Exercises Solutions</b>	<b>687</b>
22.1 Exercise 1 . . . . .	687
22.2 Exercise 2 . . . . .	691
<b>23 Training with GPUs Exercises Solutions</b>	<b>697</b>
23.1 Exercise 1 . . . . .	697
23.2 Exercise 2 . . . . .	700
<b>24 Performance Improvement Exercises Solutions</b>	<b>703</b>
24.1 Exercise 1 . . . . .	703
<b>25 Pretrained Models for Images Exercises Solutions</b>	<b>711</b>
25.1 Exercise 1 . . . . .	711
25.2 Exercise 2 . . . . .	713
25.3 Exercise 3 . . . . .	714
<b>26 Pretrained Embeddings for Text Exercises Solutions</b>	<b>719</b>
26.1 Exercise 1 . . . . .	719
26.2 Exercise 2 . . . . .	722
<b>27 Serving Deep Learning Models Exercises Solutions</b>	<b>729</b>
27.1 Exercise 1 . . . . .	729
27.2 Exercise 2 . . . . .	733



# O

## Introduction

### Welcome

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.

### Acknowledgements

This book would not exist without the help of many friends and colleagues who contributed in several ways. Special thanks go to Ari and Nate from Fullstack for the continuous support and useful suggestions throughout the project. Thanks to Nicolò for reading throughout the early version of the book and contributing many corrections to make it more accessible to beginners. Thanks to Carlo for helping me transform this book into a successful bootcamp. A huge thank you to François Chollet for inventing Keras and making Deep Learning accessible to the world and to the Tensorflow developers for the amazing work they are doing. Finally to Chiara, to my friends and to my family for all the emotional support through this journey: thank you, I would not have finished this without all of you!



Francesco Mosconi

## About the author

Francesco Mosconi is an experienced data scientist and entrepreneur. He is the founder/CEO and Chief Data Scientist of [Catalit](#), a data science consulting and training company based in San Francisco, CA.

With 15 years experience working with data, Francesco has been an instructor at General Assembly, The Data Incubator, Udemy and many conferences including ODSC, TDWI, PyBay and AINext.

Formerly he was co-founder and Chief Data Officer at Spire, a YC-backed company that invented the first consumer wearable device capable of continuously tracking respiration and physical activity.

Francesco started his career with a PhD in Biophysics, publishing a paper on DNA mechanics that currently has over 100 citations and then turned his data skills to business helping small and large companies grow their market through data analytics solutions.

He also started a series of workshops on Machine Learning and Deep Learning called [Dataweekends](#), training hundred of students on these topics.

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.

## 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.
2. 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:

$$x_0 + x_1 + x_2 \quad (1)$$

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 in equation 2:

$$x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + \dots \quad (2)$$

we may use the summation symbol  $\sum$  and write it like in equation 3:

$$\sum_i x_i \quad (3)$$

## 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  $\partial$  like this:

$$\frac{\partial f(x_1, x_2, \dots)}{\partial x_1} \quad (4)$$

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.

## Python

This is not a book about Python. It's a book about Deep Learning and how to build Deep Learning models using Python. Therefore, some familiarity with Python is required in order to follow along.

One of the questions we often receive is “I don't have any experience with Python, would I able to follow the course if I take a basic Python course before?”. The answer is YES, but let us help you with that.

This book focuses on data techniques and it assumes some familiarity with Python and programming languages. It is designed to speed up your learning curve in deep learning giving you enough knowledge for you to then be able to continue learning on your own.

So what are the things you can do to ramp up in Python?

Here are 2 resources to get you started: - [Learn Python the Hard Way](#): a great way to start learning without putting too much effort into it. - [Hacker Rank 30 days of code](#): more problem-solving oriented resource.

In order to follow this course easily, you should be familiar with the following Python constructs:

- special keywords: *in*, *return*, *None*
- variables and data types: *int*, *float*, *strings*
- data structures: *lists*, *dictionaries*, *sets* and *tuples*
- flow control: *for* loops, *while* loops, conditional statements (*if*, *elif*, *else*)
- *functions*
- *classes* and a bit of *object oriented programming*
- *packages* and how to *import* them

- *pythonic* constructs like *list comprehension*, *iterators* and *generators*

Once you are comfortable with these you'll be ready to take this course.

## 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 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
[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

### Environment creation

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.

TIP: when you run the `create` command you will get the error:

`CondaValueError: prefix already exists:`

If the environment already exists. In that case run the command `conda env update` instead.

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
- bz2file==0.98
- cython==0.28.*
- flask==1.0.*
- gensim==3.4.*
- h5py==2.8.*
- jupyter==1.0.*
- matplotlib==2.2.*
- numpy==1.14.*
- pandas==0.23.*
- pillow==5.2.*
- pip==10.0.*
```

```
- pytest==3.7.*  
- scikit-learn==0.19.*  
- scipy==1.1.*  
- seaborn==0.9.*  
- twisted==18.7.*  
- pip:  
  - PyHamcrest==1.9.*  
  - tensorflow==1.10.1  
  - keras==2.2.2  
  - jupyter_contrib_nbextensions==0.5.0  
  - tensorflow-serving-api==1.10.1
```

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:  
# > conda activate ztdlbook  
#  
# To deactivate an active environment, use:  
# > conda 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](#).

## Environment activation

So let's go ahead and activate the environment by typing:

```
conda 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.

## GPU enabled environment

If your machine has an NVIDIA GPU you'll want to install the GPU-enabled version of Tensorflow. In [Chapter 9](#) we cover cloud GPUs in detail and explain how to install all the required software. Assuming you have already installed the NVIDIA drivers, CUDA and CUDnn, you can create the environment for this book in a few simple steps:

1. run: `conda env create -f environment-gpu.yml` This is the exact same environment as the standard one, minus the `tensorflow-serving-api` package. The reason for this is that this package has standard Tensorflow as dependency and if we install it programmatically it will clutter our `tensorflow-gpu` installation. So go ahead and create the environment using the above config file.
2. Once the environment is created, activate it: `conda activate ztdlbook`
3. Finally install the `tensorflow-serving-api` package without dependencies: `pip install tensorflow-serving-api==1.10.1 --no-deps`

## 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!

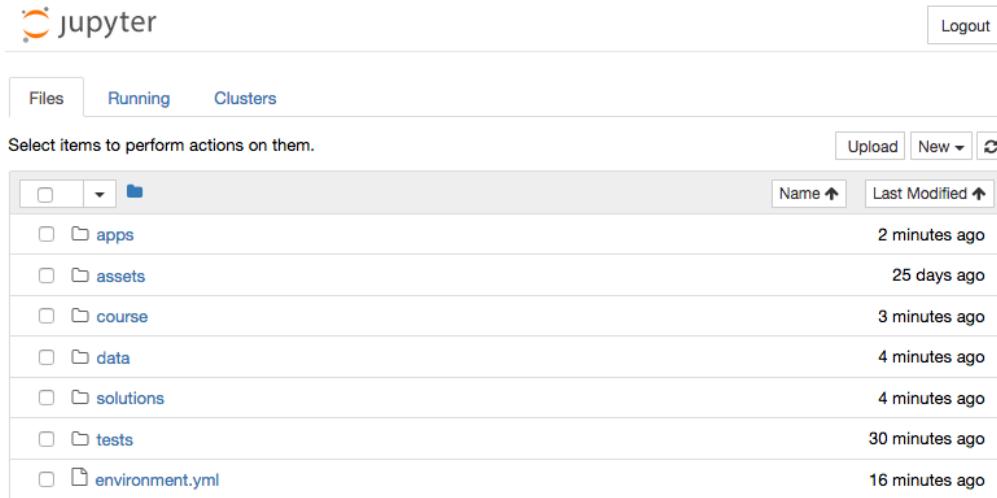
### Starting Jupyter Notebook

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 the one shown in here:

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



## Jupyter Notebook

- *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.

## Starting your first Jupyter Notebook

If you are new to Jupyter Notebook, it may feel a little disorienting at first, especially if you are used to working in an IDE. However, you’ll see that it’s actually quite easy to navigate your way around it. Let’s start from how you open the first notebook.

Click on the course folder:

and the notebooks forming the course will appear. Go ahead and click on the `00_Introduction.ipynb` notebook:

This will open a new tab where you should see the content of this chapter in the notebook. Now scroll down to this point and feel free to continue reading from the screen if you prefer.

The screenshot shows the Jupyter Notebook interface with the 'Files' tab selected. At the top, there is a navigation bar with 'Logout' and tabs for 'Files', 'Running', and 'Clusters'. Below the navigation bar, a message says 'Select items to perform actions on them.' On the right, there are buttons for 'Upload', 'New', and a refresh icon. The main area displays a list of files and folders:

	Name	Last Modified
<input type="checkbox"/>	apps	2 minutes ago
<input type="checkbox"/>	assets	25 days ago
<input checked="" 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

The course folder

The screenshot shows the Jupyter Notebook interface with the 'Files' tab selected, displaying the contents of the 'course' folder. At the top, there is a navigation bar with 'Logout' and tabs for 'Files', 'Running', 'Clusters', and 'Nbextensions'. Below the navigation bar, a message says 'Select items to perform actions on them.' On the right, there are buttons for 'Upload', 'New', and a refresh icon. The main area displays a list of files:

	Name	Last Modified	File size
<input type="checkbox"/>	..	seconds ago	
<input type="checkbox"/>	assets	3 minutes ago	
<input checked="" type="checkbox"/>	00_Introduction.ipynb	2 minutes ago	31.4 kB
<input type="checkbox"/>	01_Getting_Started.ipynb	37 minutes ago	66.2 kB
<input type="checkbox"/>	02_Data_Manipulation.ipynb	36 minutes ago	57.6 kB
<input type="checkbox"/>	03_Machine_Learning.ipynb	34 minutes ago	96.2 kB
<input type="checkbox"/>	04_Deep_Learning.ipynb	34 minutes ago	64.4 kB
<input type="checkbox"/>	05_Deep_Learning_Internal.ipynb	34 minutes ago	109 kB

Course notebook

## Jupyter Notebook cheatsheet

Let us summarize here a few very useful commands to get you started with Jupyter Notebook.

TIP: For a complete introduction to the Jupyter Notebook we encourage you to have a look at the official [documentation](#).

- **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:

### 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.

TIP: If you get an error, try the following: 1. Close this notebook. 2. Go to the terminal and **stop Jupyter Notebook** using: **CTRL+C** 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: **conda activate ztdlbook** - Windows: **activate ztdlbook** 5. Restart Jupyter Notebook. 6. Re-open the first notebook in the course folder 7. Re-run the next cell.

```
In [1]: import os  
        import sys
```

## Keyboard shortcuts

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

Jupyter shortcuts

```

env_name = 'ztdlbook'

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

```

Congrats! Your environment is correct!

## 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.

```

In [2]: python_version = '3.6'
         distribution = 'Anaconda'

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

```

Congrats! Your Python is correct!

## Jupyter

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

```
In [3]: import jupyter
        j = jupyter.__file__

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

```
Congrats! You are using 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.

```
In [4]: import pip
        import bz2file
        import cython
        import flask
        import gensim
        import h5py
```

```
import jupyter
import matplotlib
import numpy
import pandas
import PIL
import pytest
import sklearn
import scipy
import seaborn
import twisted
import hamcrest
import tensorflow
import keras
import tensorflow_serving

def check_version(pkg, version):
    actual = pkg.__version__.split('.')
    if len(actual) == 3:
        actual_major = '.'.join(actual[:2])
    elif len(actual) == 2:
        actual_major = '.'.join(actual)
    else:
        raise NotImplementedError(pkg.__name__ +
                                  "actual version :"+
                                  pkg.__version__)
    try:
        assert(actual_major == version)
    except Exception as ex:
        print("{} {}\\t=> {}".format(pkg.__name__,
                                       version,
                                       pkg.__version__))
        raise ex

check_version(cython, '0.28')
check_version(flask, '1.0')
check_version(gensim, '3.4')
check_version(h5py, '2.8')
check_version(matplotlib, '2.2')
check_version(numpy, '1.14')
check_version(pandas, '0.23')
check_version(PIL, '5.2')
check_version(pip, '10.0')
check_version(pytest, '3.7')
check_version(sklearn, '0.19')
check_version(scipy, '1.1')
check_version(seaborn, '0.9')
check_version(twisted, '18.7')
check_version(hamcrest, '1.9')
```

```
check_version(tensorflow, '1.10')
check_version(keras, '2.2')

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

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

# 1

## Getting Started

### 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

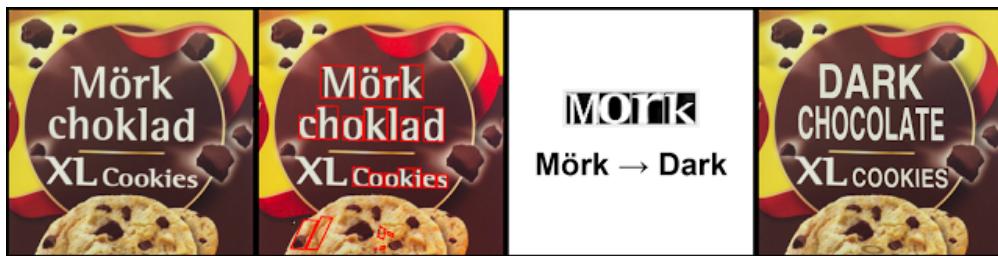
Deep Learning may be applied for predictive purposes. Algorithms can be applied to times series of a certain value over time, in order to forecast a future trend. For example the energy consumption of a region, the temperature over an area, the price of a stock, and so on. Again, Neural Networks can be used for predicting demographics and election results or even earthquakes.



Object Detection

### 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.



Machine Translation

### 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.



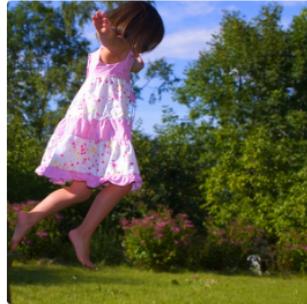
"man in black shirt is playing guitar."



"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."

### Image Caption Generation

### 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.

## 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:

```
In [1]: import numpy as np
```

TIP: If you get an error message similar to the following one, don't worry.

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

Python error messages may not seem very easy to navigate, but it is actually quite simple. To understand the error, we suggest reading from the bottom up. Usually the last line is the most informative. Here it says: `ImportError`, so it looks like it didn't find the module called `numpy`. This could be due to many reasons, but it probably indicates that either you didn't install `numpy` or you didn't activate the conda environment. Please refer back to the installation section and make sure you have activated the environment before starting `jupyter notebook`.

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

```
In [2]: 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:

```
In [3]: a
```

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

Note that the type of `a` is `numpy.ndarray`. The documentation for this type is available [here](#).

In [4]: `type(a)`

Out[4]: `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:

```
In [5]: 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:

In [6]: b

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

In [7]: c

```
Out[7]: 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]]])
```

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:

In [8]: a.shape

```
Out[8]: (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 `verylongclumsynenamevariable` 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.

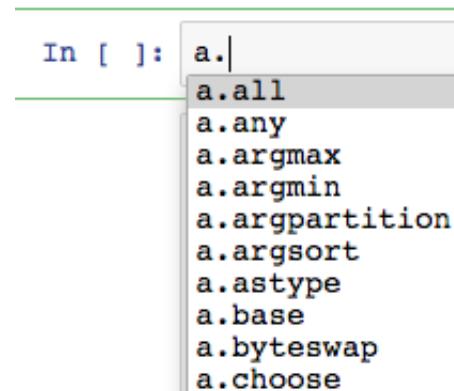
and then hit tab. You will notice that a small window pops up with all the methods and attributes available for the object a. This looks like:

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

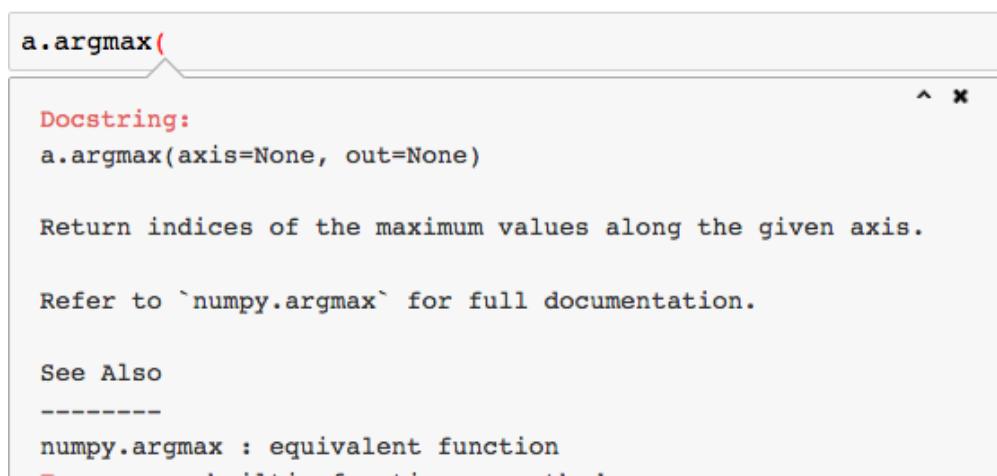
**Trick 3: Documentation pop-up** Let's go ahead and select `a.argmax` from the tab menu by hitting enter. This is a method, and we can quickly find how it works. Let's open a **round parenthesis** `a.argmax()` and let's hit SHIFT+TAB+TAB (that is TAB two times in a row while holding down SHIFT). This will open a pop-up window with the documentation of the argmax function.

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:



Methods pop-up pressing DOT+TAB



Access documentation with SHIFT+SHIFT+TAB

```
In [9]: b.shape
```

```
Out[9]: (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  $2 \times 4$  matrix.

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

```
In [10]: c.shape
```

```
Out[10]: (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:

```
In [11]: a[0]
```

```
Out[11]: 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`.

```
In [12]: # 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:

```
In [13]: c[2, 0]
```

```
Out[13]: array([0, 4, 5])
```

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

```
In [14]: b[:, 0]
```

```
Out[14]: 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:

```
In [15]: a[0:2]
```

```
Out[15]: array([1, 3])
```

and we can select the upper left  $2 \times 2$  sub-matrix in b as:

```
In [16]: b[:1, :1]
```

```
Out[16]: array([[8]])
```

Try selecting the elements requested in the next few lines.

Select the second and third elements of a:

```
In [17]: # uncomment and complete the next line
    # assert ( your code here == np.array([3, 2]))
```

Select the elements from 1 to the end in a:

```
In [18]: # 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:

```
In [19]: # 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:

```
In [20]: a[0:-1:2]
```

```
Out[20]: array([1, 2])
```

or, simply:

```
In [21]: a[::-2]
```

```
Out[21]: 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])
```

```
In [22]: 3 * a
```

```
Out[22]: array([ 3,  9,  6, 12])
```

```
In [23]: a + a
```

```
Out[23]: array([2, 6, 4, 8])
```

```
In [24]: a * a
```

```
Out[24]: array([ 1,  9,  4, 16])
```

```
In [25]: a / a
```

```
Out[25]: array([1., 1., 1., 1.])
```

```
In [26]: a - a
```

```
Out[26]: array([0, 0, 0, 0])
```

```
In [27]: a + b
```

```
Out[27]: array([[ 9,  8,  8,  5],
                [ 5,  6,  2, 11],
                [ 2,  6,  4, 13]])
```

```
In [28]: a * b
```

```
Out[28]: array([[ 8, 15, 12,  4],
   [ 4,  9,  0, 28],
   [ 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`.

```
In [29]: import matplotlib.pyplot as plt
```

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

Let's set a few common parameters that define how plots will look like. Their names are self-explanatory. In future chapters we'll bundle these into a configuration file.

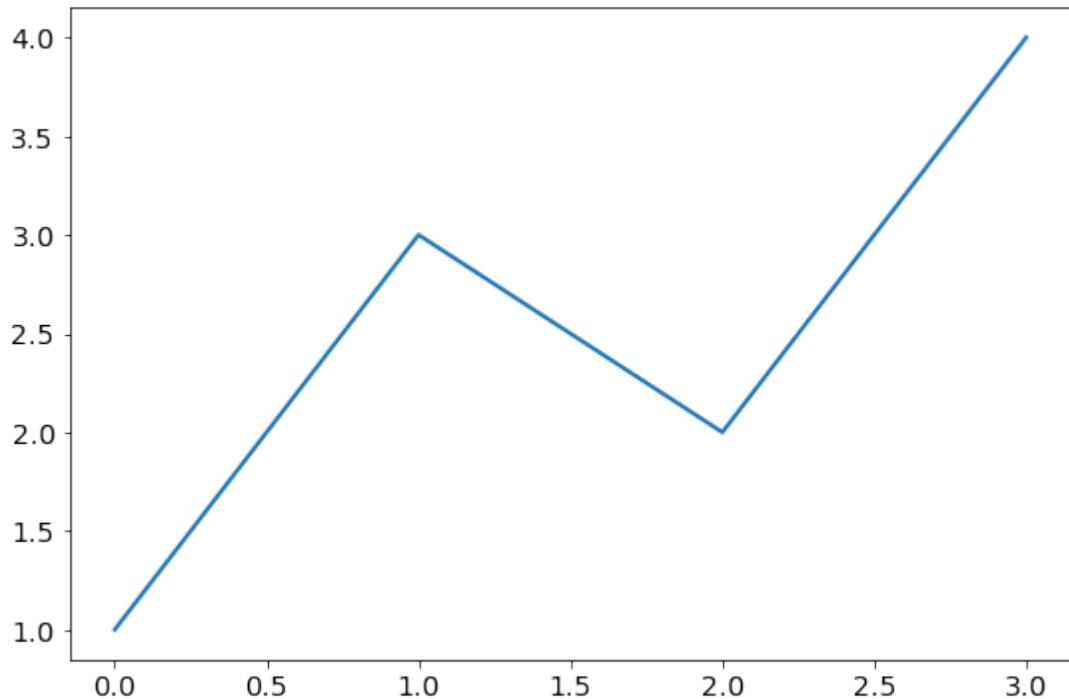
```
In [30]: from matplotlib.pyplot import rcParams
```

```
rcParams['font.size'] = 14
rcParams['lines.linewidth'] = 2
rcParams['figure.figsize'] = (9, 6)
rcParams['axes.titlepad'] = 14
rcParams['savefig.pad_inches'] = 0.2
```

## Plot

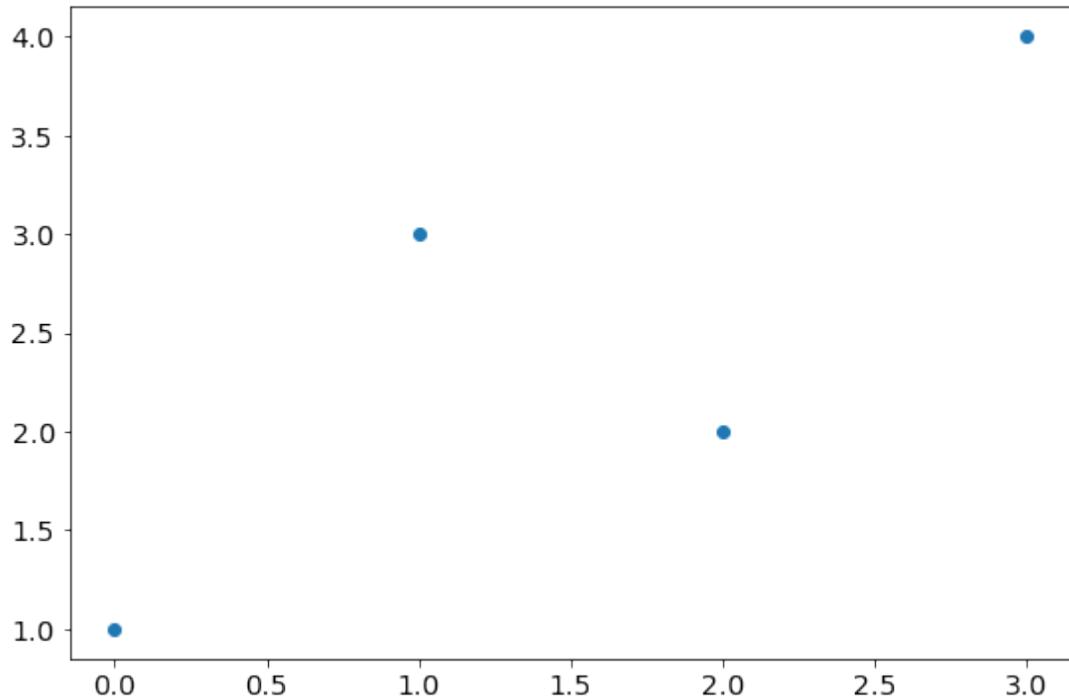
To plot some data with a line plot, we can just call the `plot` function on that data. We can try plotting our `a` vector from above like so:

```
In [31]: plt.plot(a);
```



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

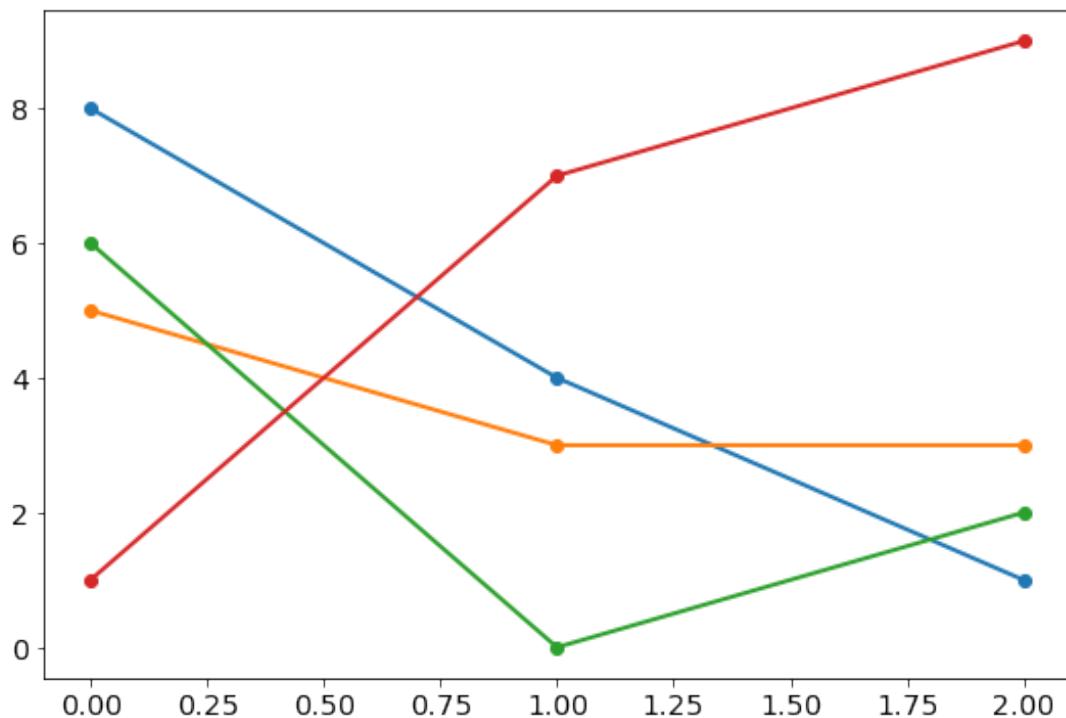
```
In [32]: plt.plot(a, 'o');
```



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.

```
In [33]: plt.plot(b, 'o-');
```



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:

In [34]: b

Out[34]: array([[8, 5, 6, 1],  
[4, 3, 0, 7],  
[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:

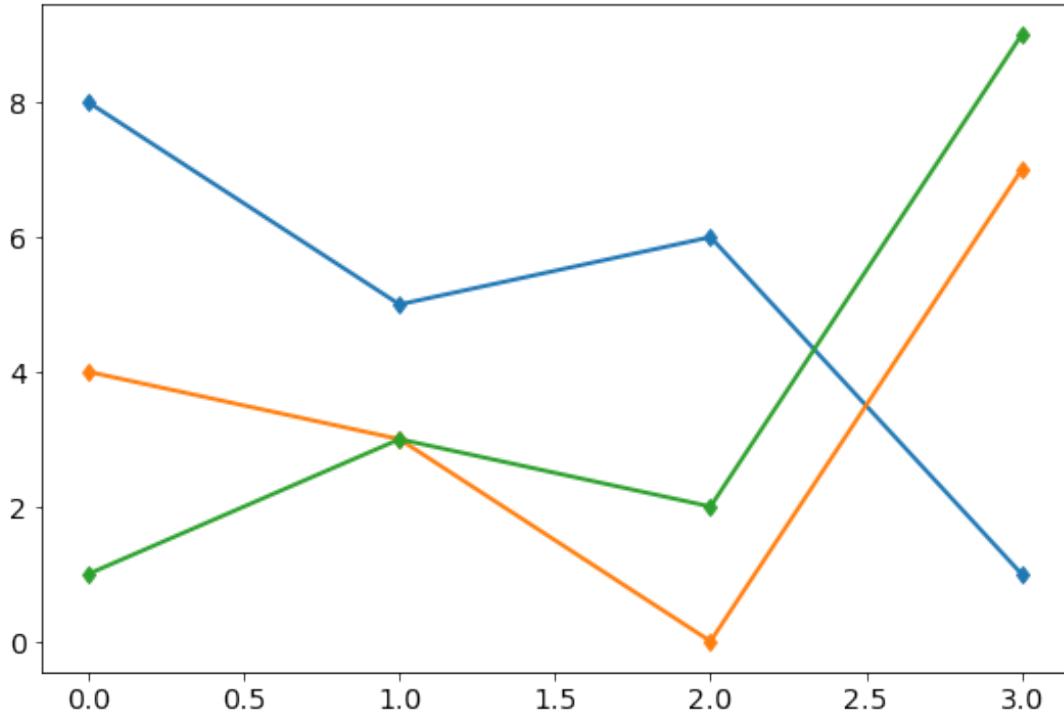
In [35]: b.transpose()

Out[35]: array([[8, 4, 1],  
[5, 3, 3],

```
[6, 0, 2],
[1, 7, 9])
```

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

```
In [36]: plt.plot(b.transpose(), 'd-');
```



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):

```
In [37]: 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 )
```

```

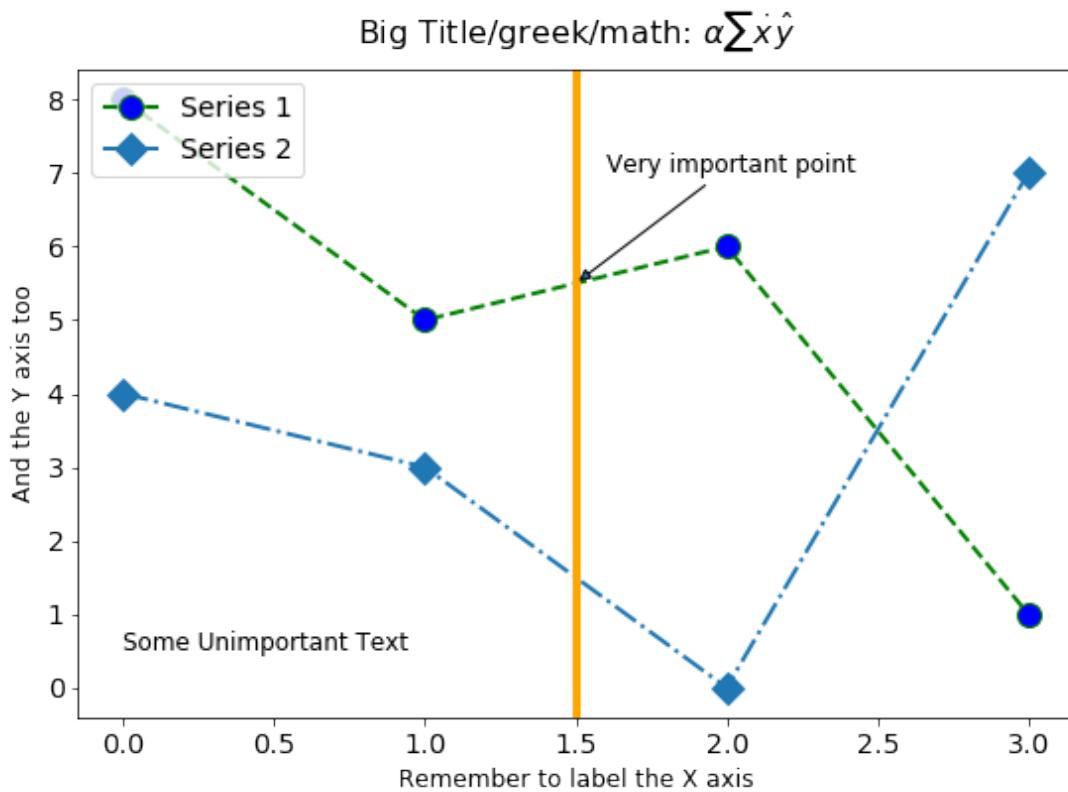
plt.xlabel('Remember to label the X axis', fontsize=12)
plt.ylabel('And the Y axis too', fontsize=12)

t = r'Big Title/greek/math: $\alpha \sum \dot{x} \hat{y}$'
plt.title(t, 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'], loc=2);

```



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

```
In [38]: from sklearn.datasets 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  $\geq 2$  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  $\geq 1$ . 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`:

```
In [39]: X
```

```
Out[39]: 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]])
```

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

```
In [40]: y[:10]
```

```
Out[40]: array([1, 1, 0, 1, 1, 1, 0, 0, 0, 1])
```

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

```
In [41]: X.shape
```

```
Out[41]: (1000, 2)
```

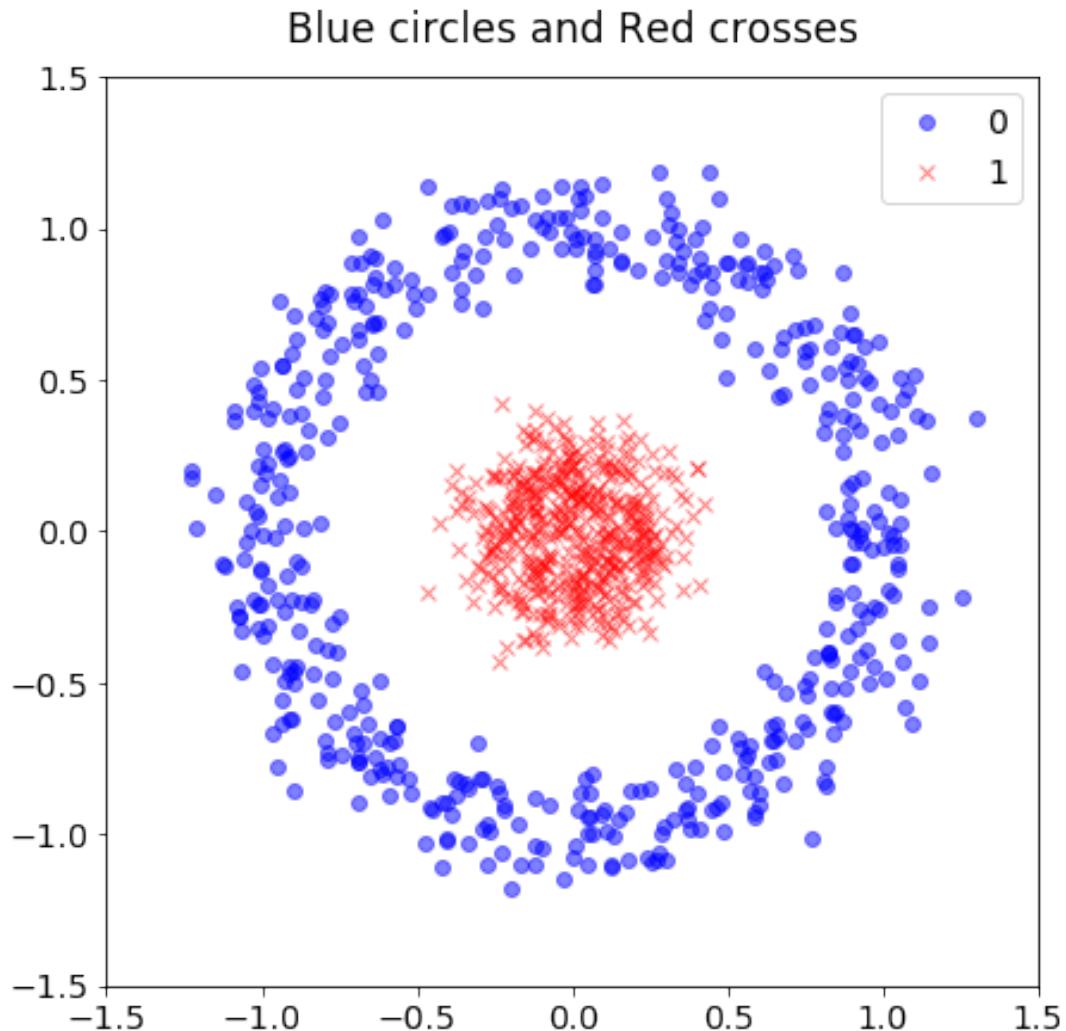
```
In [42]: y.shape
```

```
Out[42]: (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:

```
In [43]: plt.figure(figsize=(7, 7))
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");
```



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 are 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 time throughout the book and we'll have plenty of opportunities to familiarize ourselves with them.

Let's start by importing a few libraries:

```
In [44]: from keras.models import Sequential  
        from keras.layers import Dense  
        from keras.optimizers import SGD
```

Using TensorFlow backend.

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.

```
In [45]: 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.

```
In [46]: 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.

```
In [47]: model.compile(optimizer=SGD(lr=0.5),
                     loss='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.

```
In [48]: model.fit(X, y, epochs=20)
```

```
Epoch 1/20
1000/1000 [=====] - 1s 1ms/step - loss: 0.6749 - acc: 0.6580
Epoch 2/20
1000/1000 [=====] - 0s 59us/step - loss: 0.5820 - acc: 0.8280
Epoch 3/20
1000/1000 [=====] - 0s 58us/step - loss: 0.4815 - acc: 0.8490
Epoch 4/20
1000/1000 [=====] - 0s 59us/step - loss: 0.4157 - acc: 0.8640
Epoch 5/20
1000/1000 [=====] - 0s 59us/step - loss: 0.3728 - acc: 0.8700
Epoch 6/20
1000/1000 [=====] - 0s 60us/step - loss: 0.3407 - acc: 0.8700
Epoch 7/20
1000/1000 [=====] - 0s 59us/step - loss: 0.2860 - acc: 0.8940
Epoch 8/20
```

```
1000/1000 [=====] - 0s 59us/step - loss: 0.2060 - acc: 0.9410
Epoch 9/20
1000/1000 [=====] - 0s 59us/step - loss: 0.1443 - acc: 0.9960
Epoch 10/20
1000/1000 [=====] - 0s 59us/step - loss: 0.1072 - acc: 0.9990
Epoch 11/20
1000/1000 [=====] - 0s 58us/step - loss: 0.0846 - acc: 1.0000
Epoch 12/20
1000/1000 [=====] - 0s 58us/step - loss: 0.0706 - acc: 1.0000
Epoch 13/20
1000/1000 [=====] - 0s 59us/step - loss: 0.0604 - acc: 1.0000
Epoch 14/20
1000/1000 [=====] - 0s 59us/step - loss: 0.0528 - acc: 1.0000
Epoch 15/20
1000/1000 [=====] - 0s 59us/step - loss: 0.0471 - acc: 1.0000
Epoch 16/20
1000/1000 [=====] - 0s 59us/step - loss: 0.0425 - acc: 1.0000
Epoch 17/20
1000/1000 [=====] - 0s 59us/step - loss: 0.0388 - acc: 1.0000
Epoch 18/20
1000/1000 [=====] - 0s 58us/step - loss: 0.0358 - acc: 1.0000
Epoch 19/20
1000/1000 [=====] - 0s 59us/step - loss: 0.0330 - acc: 1.0000
Epoch 20/20
1000/1000 [=====] - 0s 58us/step - loss: 0.0308 - acc: 1.0000
```

Out[48]: <keras.callbacks.History at 0x7ff03af678d0>

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.

```
In [49]: hticks = np.linspace(-1.5, 1.5, 101)
       vticks = np.linspace(-1.5, 1.5, 101)
```

```
In [50]: hticks[:10]
```

```
Out[50]: array([-1.5 , -1.47, -1.44, -1.41, -1.38, -1.35, -1.32, -1.29, -1.26,
       -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.

```
In [51]: aa, bb = np.meshgrid(hticks, vticks)
```

```
In [52]: aa.shape
```

```
Out[52]: (101, 101)
```

```
In [53]: aa
```

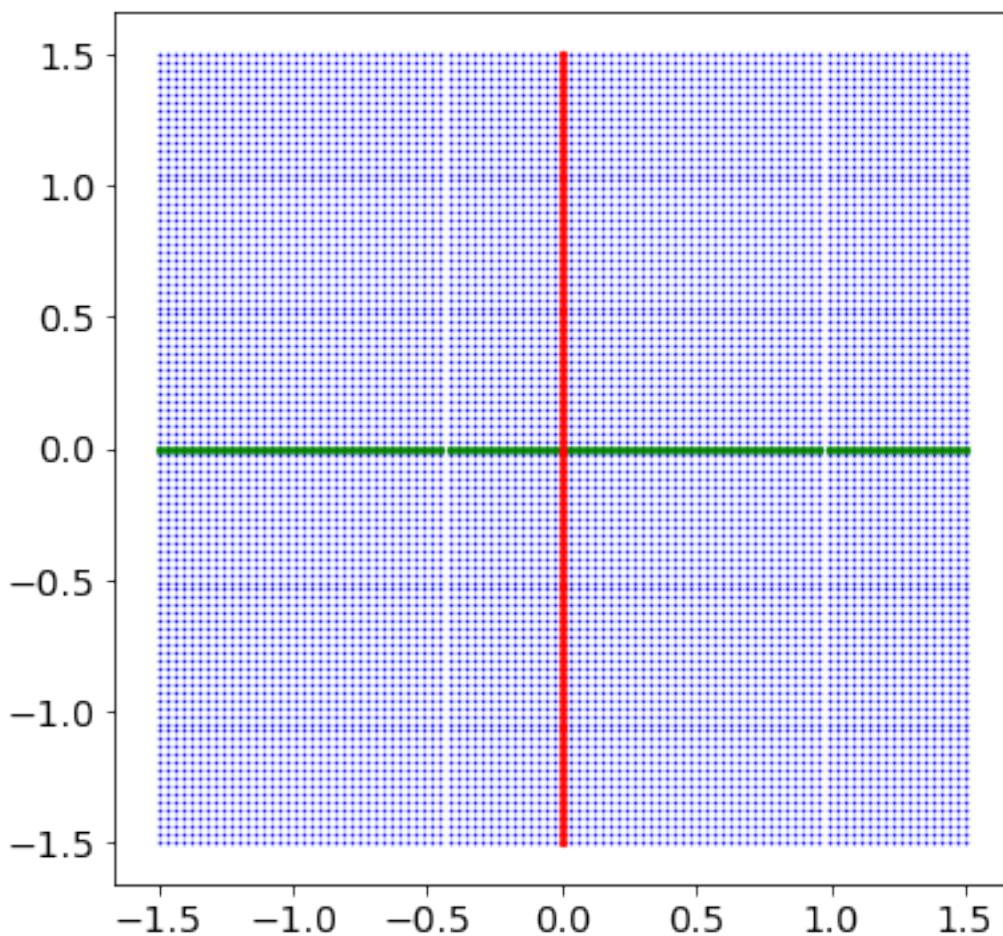
```
Out[53]: array([[[-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
   [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
   [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
   ...,
   [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
   [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ],
   [-1.5 , -1.47, -1.44, ..., 1.44, 1.47, 1.5 ]])
```

```
In [54]: bb
```

```
Out[54]: 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 ]])
```

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

```
In [55]: plt.figure(figsize=(6,6))
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');
```



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.

```
In [56]: ab = np.c_[aa.ravel(), bb.ravel()]
```

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

```
In [57]: ab.shape
```

```
Out[57]: (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.

```
In [58]: c = model.predict(ab)
```

```
In [59]: c
```

```
Out[59]: array([[1.5898737e-04],  
                 [1.2135810e-04],  
                 [9.3694063e-05],  
                 ...,  
                 [3.0899048e-02],  
                 [3.0890310e-02],  
                 [3.0880447e-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:

```
In [60]: c.min()
```

```
Out[60]: 9.601998e-06
```

```
In [61]: c.max()
```

```
Out[61]: 0.9903734
```

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

```
In [62]: c.shape
```

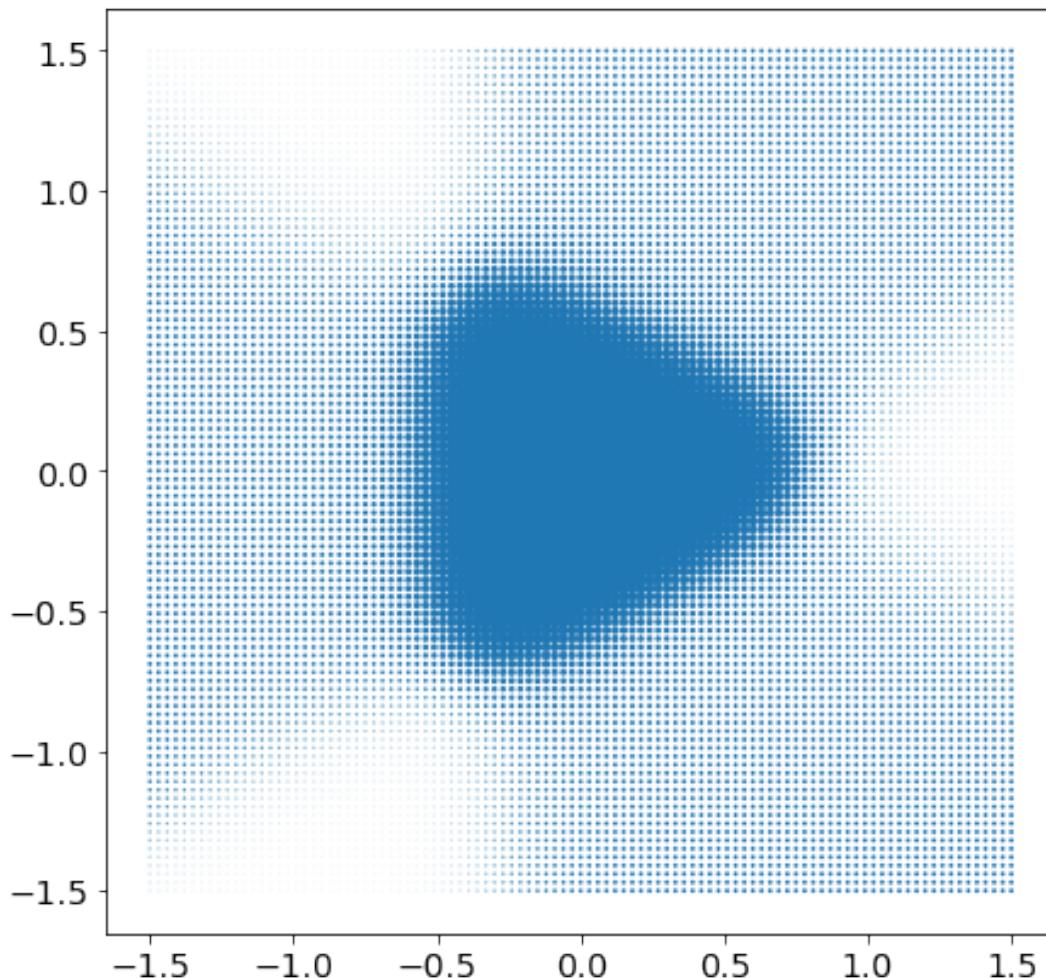
```
Out[62]: (10201, 1)
```

```
In [63]: cc = c.reshape(aa.shape)  
cc.shape
```

Out [63]: (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

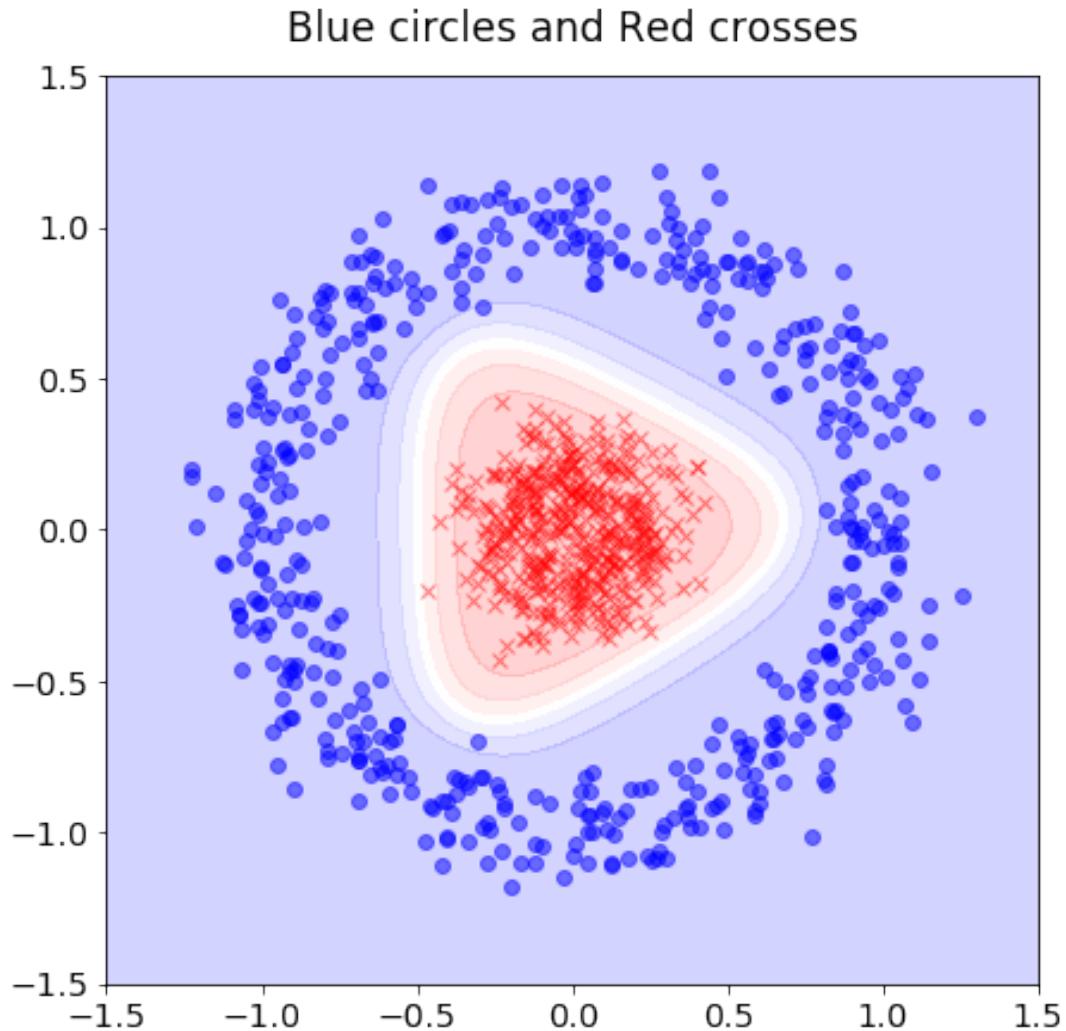
```
In [64]: plt.figure(figsize=(7, 7))
plt.scatter(aa, bb, s=20*cc);
```



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:

```
In [65]: plt.figure(figsize=(7, 7))
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");
```



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.

## Exercises

### 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

# 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 *mytabular* 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 illustrates the components of tabular data. At the top, a dark grey box labeled "Features" has two arrows pointing downwards to the left and right respectively. The left arrow points to the second column of a table, which contains attributes like Age, Gender, Annual Salary, etc. The right arrow points to the last column of the table, which is labeled "Labels". The table itself has four rows, each representing a "Record OR Data Point". The first row contains a logo icon. The second and third rows are highlighted with red backgrounds. The fourth row is also highlighted with red and contains the text "Record OR Data Point". The columns are labeled from left to right: Age, Gender, Annual Salary, Months in residence, Months in job, Current Debt, Paid off credit, and 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	18	M	\$15,000	3	1	\$10,000	No
Client 4	Record OR Data Point						?
						Labels	\$15,000

Common terms for tabular data

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.

```
In [1]: 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.

Before we do anything else, let's also set a couple of common options that will help us with contain the size of the tables displayed. We configure pandas to show at most 13 rows of data in a dataframe and at most 11 columns. Bigger dataframes will be truncated with ellipses.

```
In [2]: pd.set_option("display.max_rows", 13)
pd.set_option("display.max_columns", 11)
pd.set_option("display.latex.repr", True)
```

Notice here that the `display.latex.repr` is only to `True` set for the PDF version of the book, while it's set to `False` for the other versions. Starting from the next chapter we'll group all the configurations in a single script. Let's now load the data from the `titanic-train.csv` file:

```
In [3]: 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:

```
In [4]: df.head()
```

Out [4] :

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	female	38.0	1	0	PC 17599 STON/O2. 3101282	71.2833	C85	C
2	3	1	3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	26.0	0	0	313803	79.250	NaN	S
3	4	1	1		female	35.0	1	0		53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

`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.

In [5]: `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
SibSp           891 non-null int64
Parch           891 non-null int64
Ticket          891 non-null object
Fare            891 non-null float64
Cabin           204 non-null object
Embarked        889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

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

In [6]: `df.describe()`

Out [6] :

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

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]`

In [7] : `df . iloc [3]`

Out[7] :

	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
Fare	53.1
Cabin	C123
Embarked	S

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

In [8] : `df . loc [0:4, 'Ticket']`

Out[8] :

	Ticket
0	A/5 21171
1	PC 17599
2	STON/O2. 3101282
3	113803
4	373450

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

In [9] : `df ['Ticket'] . head ()`

Out[9] :

Ticket	
0	A/5 21171
1	PC 17599
2	STON/O2. 3101282
3	113803
4	373450

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

```
In [10]: df[['Embarked', 'Ticket']].head()
```

Out[10] :

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

```
In [11]: df[df['Age'] > 70]
```

Out[11] :

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
96	97	0	1 Goldschmidt, Mr. George B	male	71.0	0	0	PC 17754	34.6542	A5	C
116	117	0	3 Connors, Mr. Patrick	male	70.5	0	0	370369	7.7500	NaN	Q
493	494	0	1 Artagaveytia, Mr. Ramon	male	71.0	0	0	PC 17609	49.5042	NaN	C
630	631	1	1 Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	27042	30.0000	A23	S
851	852	0	3 Svensson, Mr. Johan	male	74.0	0	0	347060	7.7750	NaN	S

To understand what this does, let's break it down. `df['Age'] > 70` returns a boolean Series of values that are True when the Age is greater than 70 (and False otherwise). The lenght of this series is the same as that of the whole DataFrame, as you can check by running:

```
In [12]: len(df['Age'] > 70)
```

Out[12] : 891

Passing this series to the [] operator, selects only the rows for which the boolean series is True. In other words, Pandas matches the index of the DataFrame with the index of the Series and selects only the rows for which the condition is True.

We can obtain the same result using the query operator.

In [13]: `df.query("Age > 70")`

Out[13] :

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
96	97	0	1 Goldschmidt, Mr. George B	male	71.0	0	0	PC 17754	34.6542	A5	C
116	117	0	3 Connors, Mr. Patrick	male	70.5	0	0	370369	7.7500	NaN	Q
493	494	0	1 Artagaveitia, Mr. Ramon	male	71.0	0	0	PC 17609	49.5042	NaN	C
630	631	1	1 Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	27042	30.0000	A23	S
851	852	0	3 Svensson, Mr. Johan	male	74.0	0	0	347060	7.7750	NaN	S

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.

In [14]: `df[(df['Age'] == 11) & (df['SibSp'] == 5)]`

Out[14] :

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
59	60	0	3 Goodwin, Master. William Frederick	male	11.0	5	2	CA 2144	46.9000	NaN	S

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

In [15]: `df[(df.Age == 11) | (df.SibSp == 5)]`

Out[15] :

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
59	60	0	3 Goodwin, Master. William Frederick	male	11.0	5	2	CA 2144	46.9000	NaN	S
71	72	0	3 Goodwin, Miss. Lillian Amy	female	16.0	5	2	CA 2144	46.9000	NaN	S
386	387	0	3 Goodwin, Master. Sidney Leonard	male	1.0	5	2	CA 2144	46.9000	NaN	S
480	481	0	3 Goodwin, Master. Harold Victor	male	9.0	5	2	CA 2144	46.9000	NaN	S
542	543	0	3 Andersson, Miss. Sigrid Elisabeth	female	11.0	4	2	347082	31.2750	NaN	S
683	684	0	3 Goodwin, Mr. Charles Edward	male	14.0	5	2	CA 2144	46.9000	NaN	S
731	732	0	3 Hassan, Mr. Houssein G N	male	11.0	0	0	2699	18.7875	NaN	C
802	803	1	1 Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0000	B96 B98	S

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

In [16]: `df.query('(Age == 11) | (SibSp == 5)')`

**Out [16] :**

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
59	60	0	3	Goodwin, Master. William Frederick	male	11.0	5	2	CA 2144	46.9000	NaN	S
71	72	0	3	Goodwin, Miss. Lillian Amy	female	16.0	5	2	CA 2144	46.9000	NaN	S
386	387	0	3	Goodwin, Master. Sidney Leonard	male	1.0	5	2	CA 2144	46.9000	NaN	S
480	481	0	3	Goodwin, Master. Harold Victor	male	9.0	5	2	CA 2144	46.9000	NaN	S
542	543	0	3	Andersson, Miss. Sigrid Elisabeth	female	11.0	4	2	347082	31.2750	NaN	S
683	684	0	3	Goodwin, Mr. Charles Edward	male	14.0	5	2	CA 2144	46.9000	NaN	S
731	732	0	3	Hassan, Mr. Houssein G N	male	11.0	0	0	2699	18.7875	NaN	C
802	803	1	1	Carter, Master. William Thornton II	male	11.0	1	2	113760	120.0000	B96 B98	S

## 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.

**In [17] :** `df['Embarked'].unique()`

**Out [17] :** `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`.

**In [18] :** `df.sort_values('Age', ascending = False).head()`

**Out [18] :**

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
630	631	1	1	Barkworth, Mr. Algernon Henry Wilson	male	80.0	0	0	27042	30.0000	A23	S
851	852	0	3	Svensson, Mr. Johan	male	74.0	0	0	347060	7.7750	NaN	S
493	494	0	1	Artagaveytia, Mr. Ramon	male	71.0	0	0	PC 17609	49.5042	NaN	C
96	97	0	1	Goldschmidt, Mr. George B	male	71.0	0	0	PC 17754	34.6542	A5	C
116	117	0	3	Connors, Mr. Patrick	male	70.5	0	0	370369	7.7500	NaN	Q

## 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.

In [19]: df['Survived'].value\_counts()

Out[19] :

Survived
0 549
1 342

or how many people were travelling in each class.

In [20]: df['Pclass'].value\_counts()

Out[20] :

Pclass
3 491
1 216
2 184

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:

In [21]: df.groupby(['Pclass', 'Survived'])['PassengerId'].count()

Out[21] :

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

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:

```
In [22]: df['Age'].min()
```

```
Out[22]: 0.42
```

while the oldest was eighty years old:

```
In [23]: df['Age'].max()
```

```
Out[23]: 80.0
```

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

```
In [24]: df['Age'].mean()
```

```
Out[24]: 29.69911764705882
```

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

```
In [25]: df['Age'].median()
```

```
Out[25]: 28.0
```

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

```
In [26]: mean_age_by_surv = df.groupby('Survived')['Age'].mean()  
mean_age_by_surv
```

```
Out[26]:
```

	Age
Survived	
0	30.626179
1	28.343690

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.

```
In [27]: std_age_by_survived = df.groupby('Survived')['Age'].std()
        std_age_by_survived
```

Out [27] :

Survived	Age	
	0	1
0	14.172110	
1	14.950952	

## 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:

```
In [28]: df1 = mean_age_by_surv.round(0).reset_index()
        df1
```

Out [28] :

Survived	Age
0	31.0
1	28.0

```
In [29]: df2 = std_age_by_survived.round(0).reset_index()
        df2
```

Out [29] :

Survived	Age
0	14.0
1	15.0

```
In [30]: df3 = pd.merge(df1, df2, on='Survived')
        df3
```

Out [30] :

Survived	Age_x	Age_y
0	31.0	14.0
1	28.0	15.0

```
In [31]: df3.columns = ['Survived',
                      'Average Age',
                      'Age Standard Deviation']
df3
```

Out[31] :

	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:

```
In [32]: df.pivot_table(index='Pclass',
                      columns='Survived',
                      values='PassengerId',
                      aggfunc='count')
```

Out[32] :

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'.

```
In [33]: df['IsFemale'] = df['Sex'] == 'female'
```

```
In [34]: corr_w_surv = df.corr()['Survived'].sort_values()
corr_w_surv
```

Out [34] :

	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

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. Let's import matplotlib:

```
In [35]: import matplotlib.pyplot as plt
```

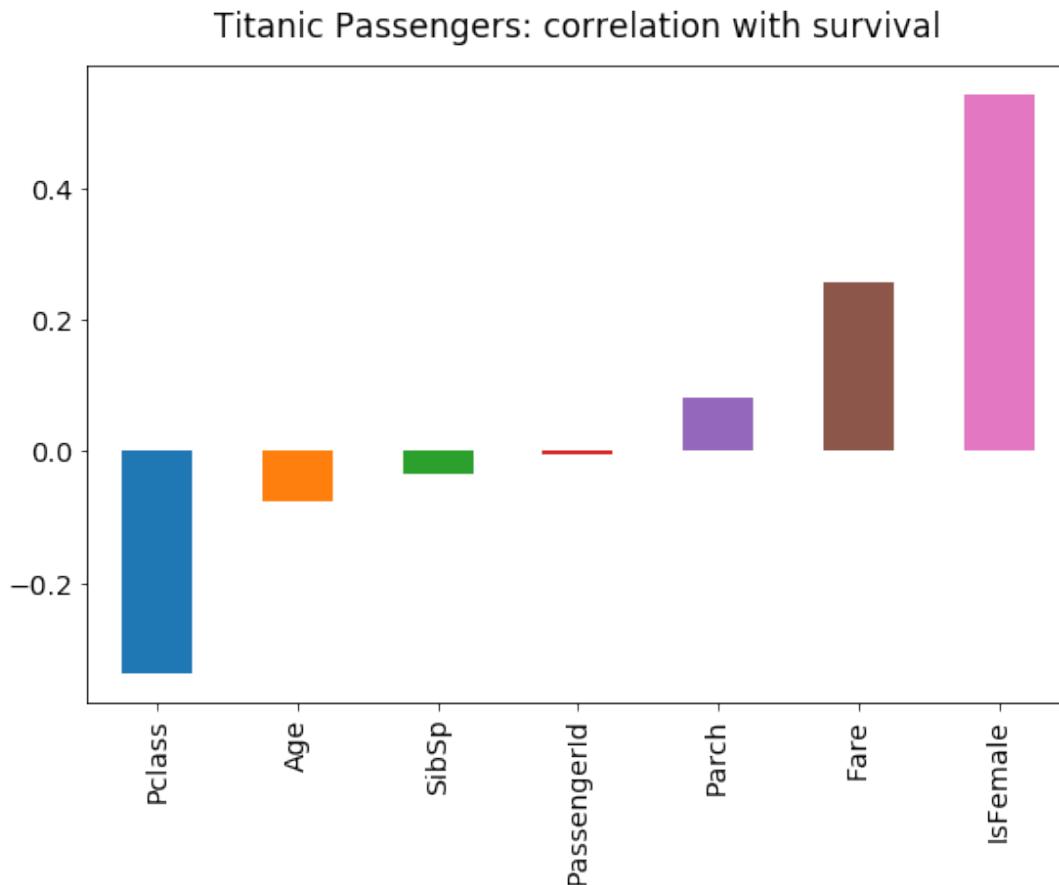
And let's set the configuration of plots first:

```
In [36]: from matplotlib.pyplot import rcParams

rcParams['font.size'] = 14
rcParams['lines.linewidth'] = 2
rcParams['figure.figsize'] = (9, 6)
rcParams['axes.titlepad'] = 14
rcParams['savefig.pad_inches'] = 0.2
```

Now let's use pandas to plot the `corr_w_surv` datafram. Notice that we will exclude the last row, which is `Survived` itself:

```
In [37]: title = 'Titanic Passengers: correlation with survival'
corr_w_surv.iloc[:-1].plot(kind='bar', title=title);
```



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.

```
In [38]: import numpy as np
```

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 (data 3). - A sequence with somewhat intermediate noise, following a sinusoidal oscillatory pattern (data 4).

```
In [39]: 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. First we aggregate the data using `np.vstack` and we transpose it:

```
In [40]: data = np.vstack([data1, data2, data3, data4])
        data = data.transpose()
```

Then we create a data frame with the appropriate column names:

```
In [41]: cols = ['data1', 'data2', 'data3', 'data4']

        df = pd.DataFrame(data, columns=cols)
        df.head()
```

Out[41]:

	data1	data2	data3	data4
0	0.106252	0.625998	2.397470	2.927218
1	-0.035474	0.733440	2.959889	3.058139
2	-0.022673	0.817278	2.183994	3.077296
3	-0.103543	0.896346	2.764160	2.792608
4	0.041501	0.723697	2.356034	2.885100

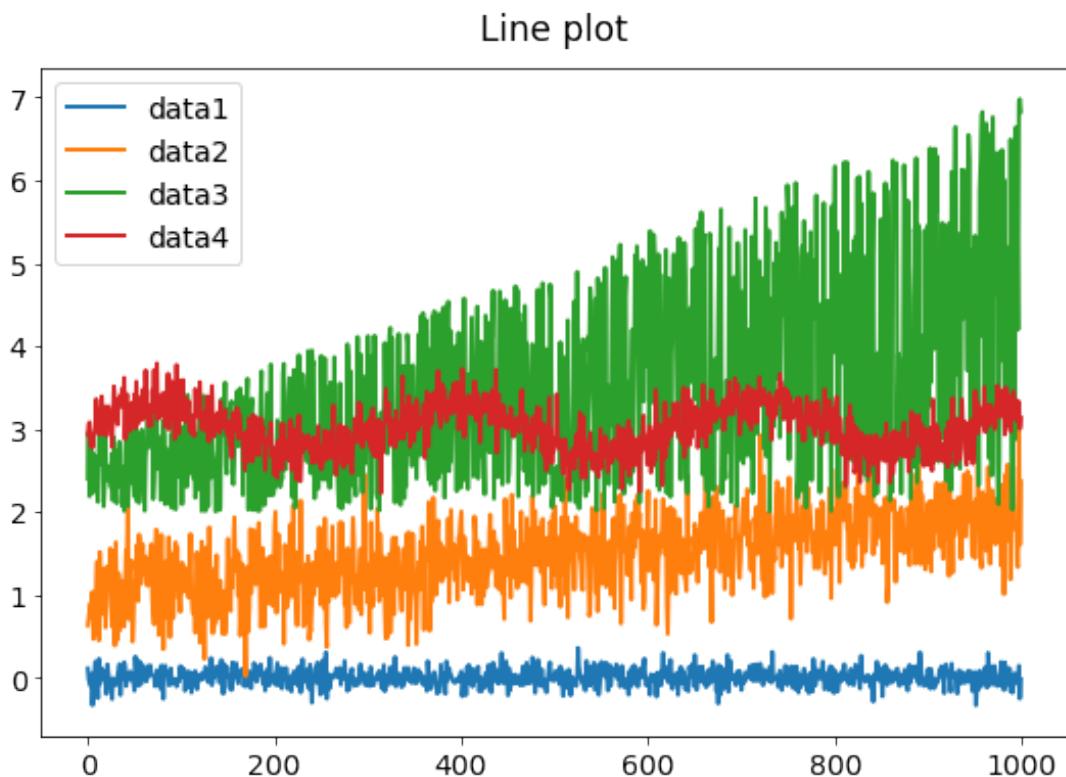
Even when we've been given a description of these four data sets, it's really hard to understand what's going on by simply looking at the table of numbers. Instead, let's look at this data visually.

## 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).

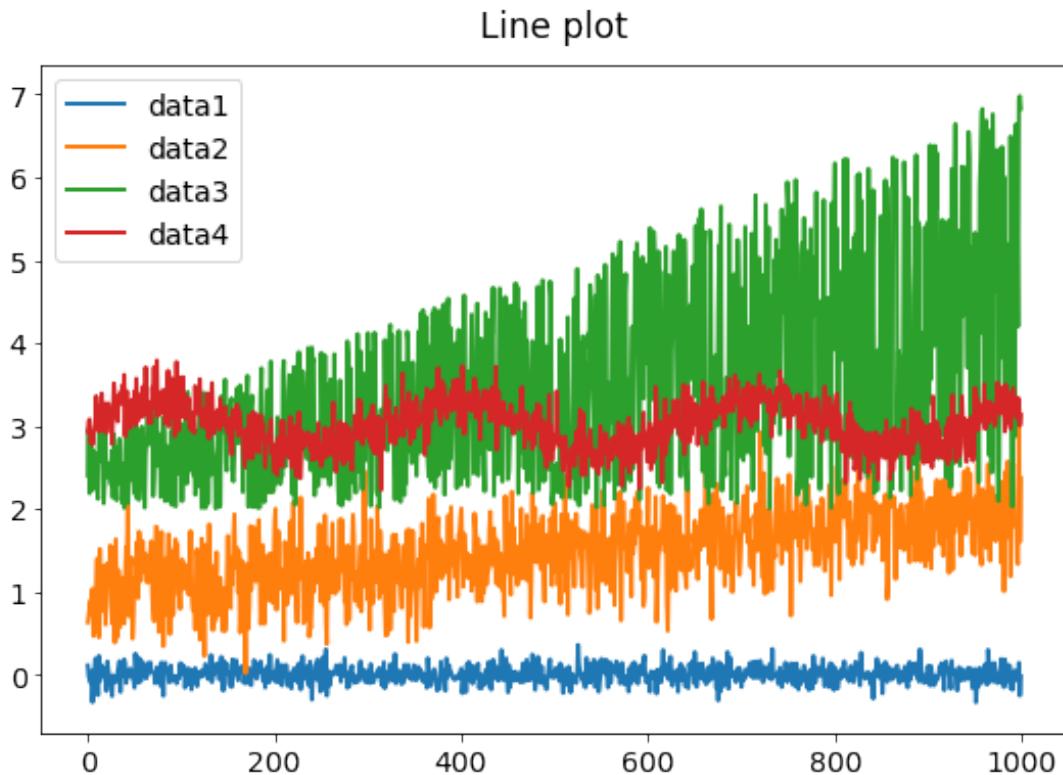
A line plot represents the values of data in sequential order, and makes it easy to spot trends like growth over time or seasonal patterns.

```
In [42]: 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:

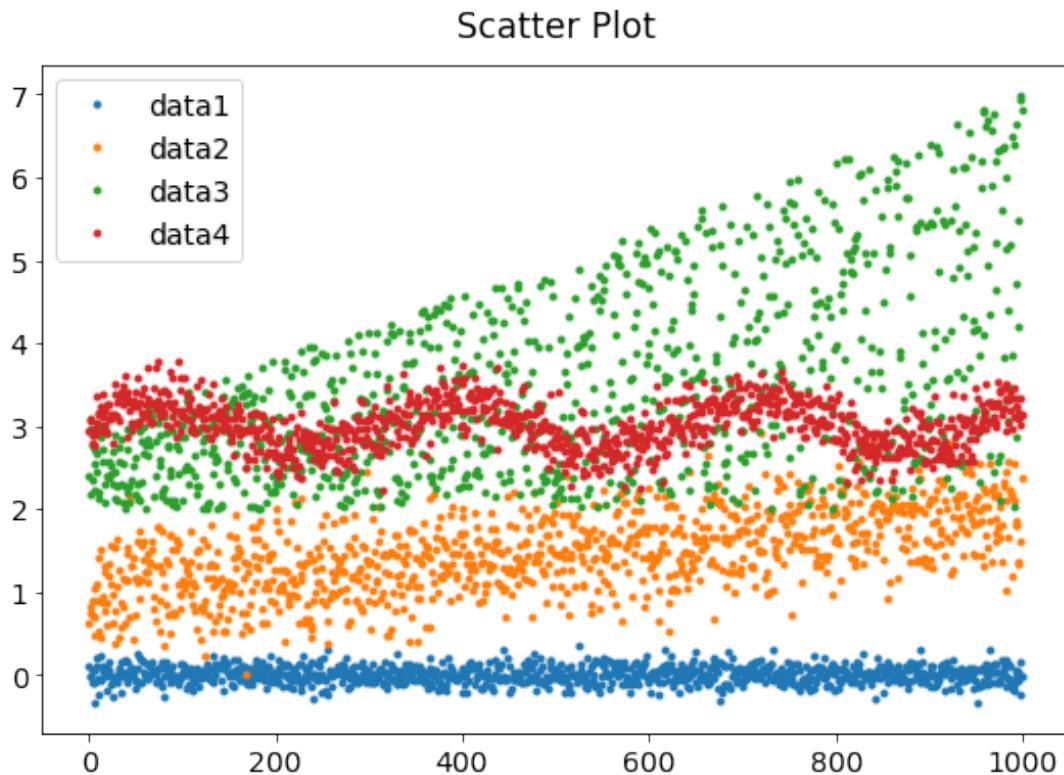
```
In [43]: plt.plot(df)
    plt.title('Line plot')
    plt.legend(['data1', 'data2', 'data3', 'data4']);
```



## Scatter plot

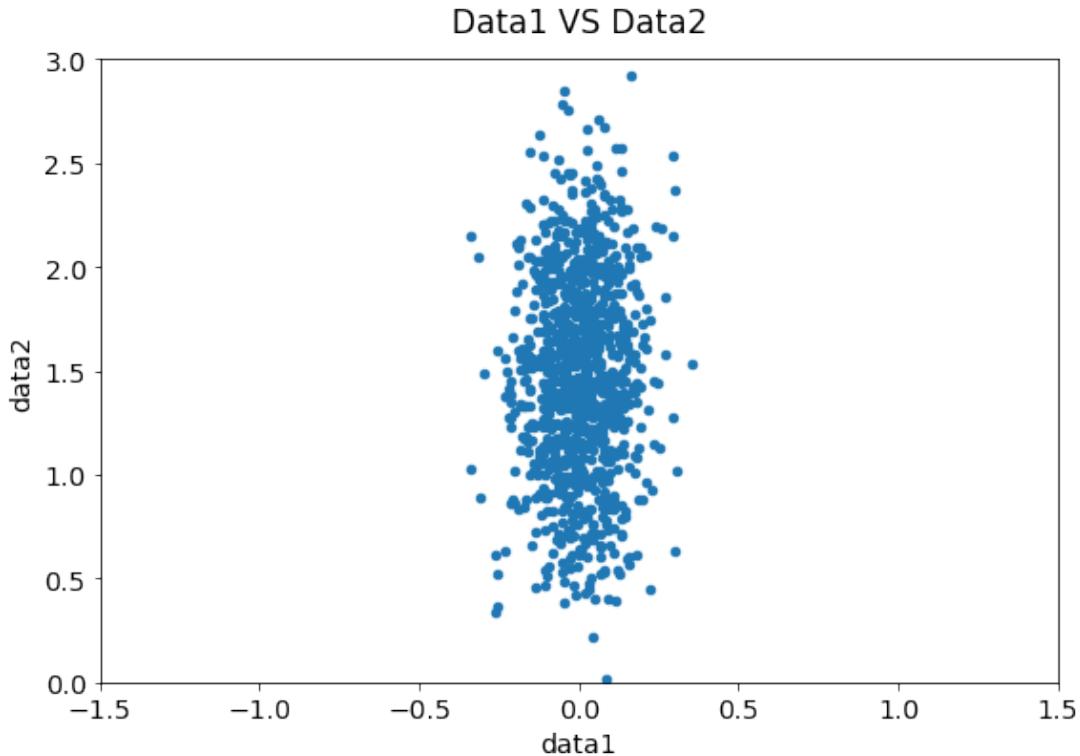
If data is not ordered, and we are looking for correlations between variables, a `scatter` plot is a better choice. We can simply change the style of the line plot if we want to plot data in order:

```
In [44]: df.plot(style='.', title='Scatter Plot');
```



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

```
In [45]: df.plot(kind='scatter', x='data1', y='data2',
                 xlim=(-1.5, 1.5), ylim=(0, 3),
                 title='Data1 VS Data2');
```



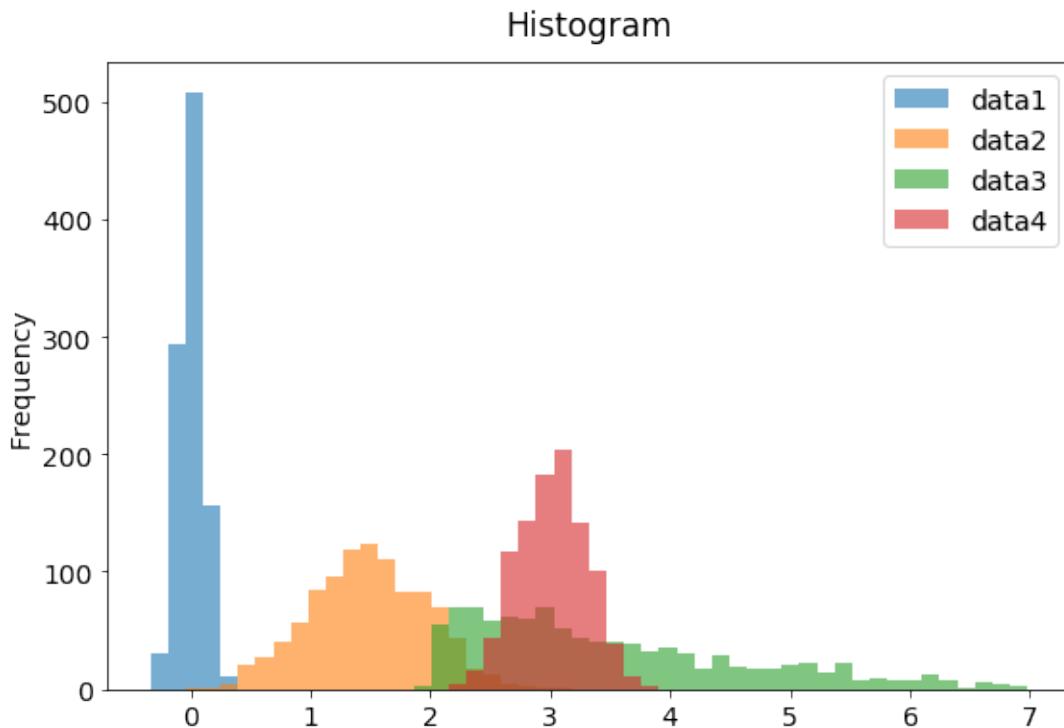
In the above plot, we see that there is no correlation between `data1` and `data2` (which may be obvious because `data1` is a flat random noise).

## 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.

```
In [46]: df.plot(kind='hist',
               bins=50,
               title='Histogram',
               alpha=0.6);
```

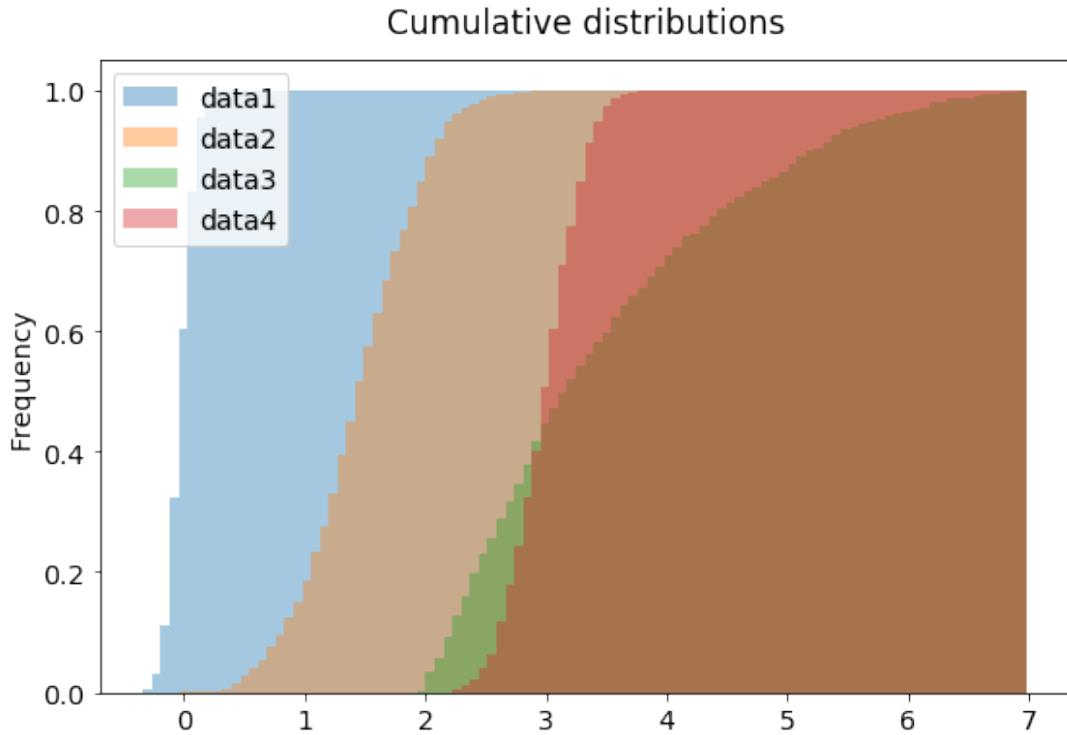


Note that we lost all the temporal information contained in our data, for example the oscillations in data4 are not visible any longer, all we see is a quite large bell-like distribution, where the sinusoidal oscillations have been summed up in the histogram.

## 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?

```
In [47]: df.plot(kind='hist',
                 bins=100,
                 title='Cumulative distributions',
                 density=True,
                 cumulative=True,
                 alpha=0.4);
```



Try to answer these questions:

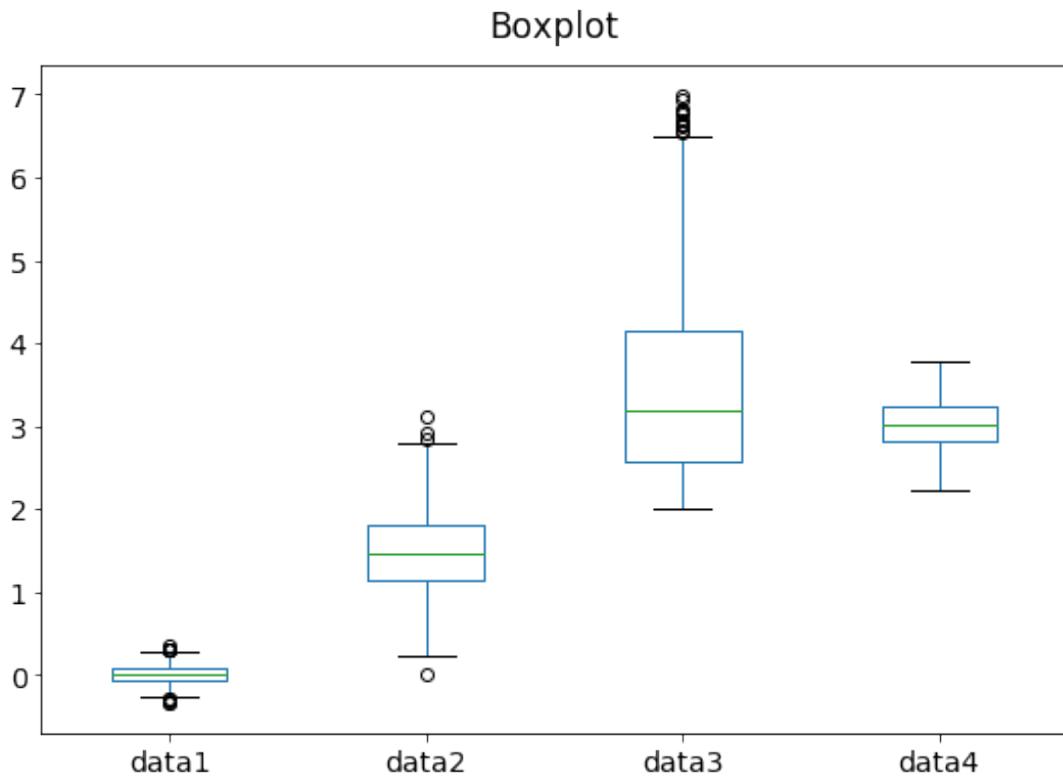
1. how much of data1 falls below 2?
2. how much of data2 falls below 1.5?

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.

```
In [48]: df.plot(kind='box',
                 title='Boxplot');
```



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  $[Q_1 - 1.5 \times IQR, Q_3 + 1.5 \times IQR]$ , where  $Q_1$  is the first quartile,  $Q_3$  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:

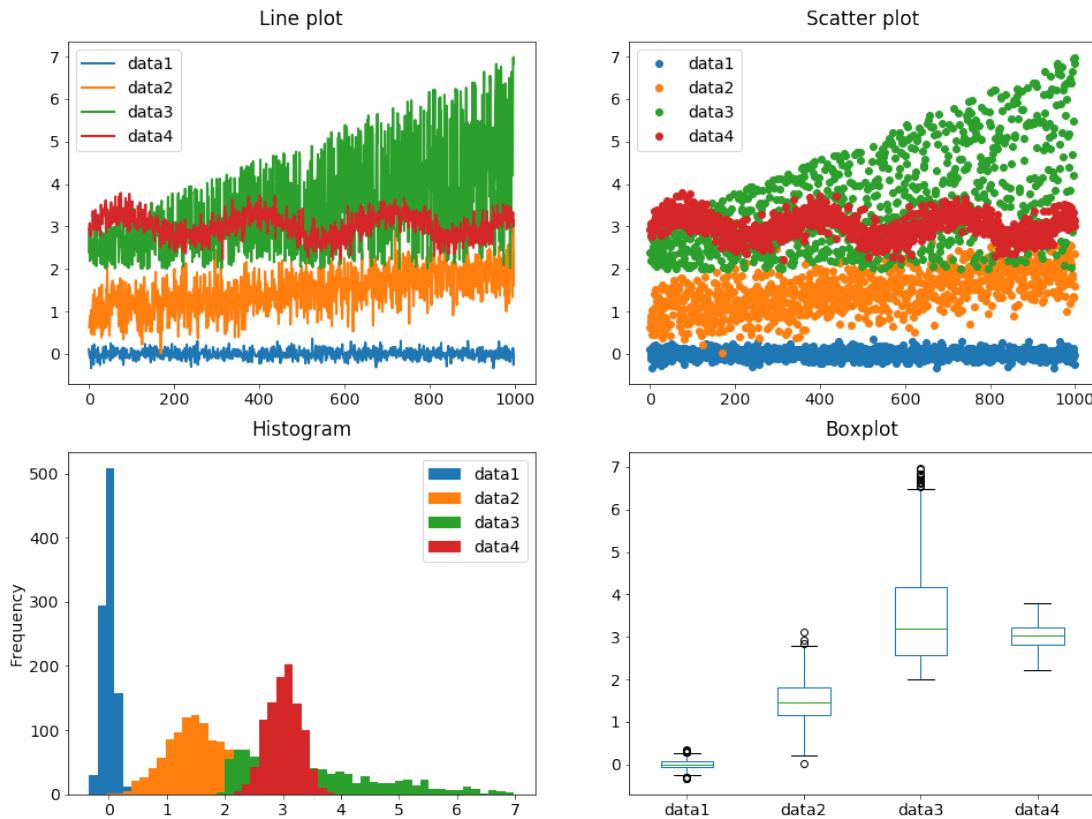
```
In [49]: fig, ax = plt.subplots(2, 2, figsize=(16,12))

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');
```



## 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:

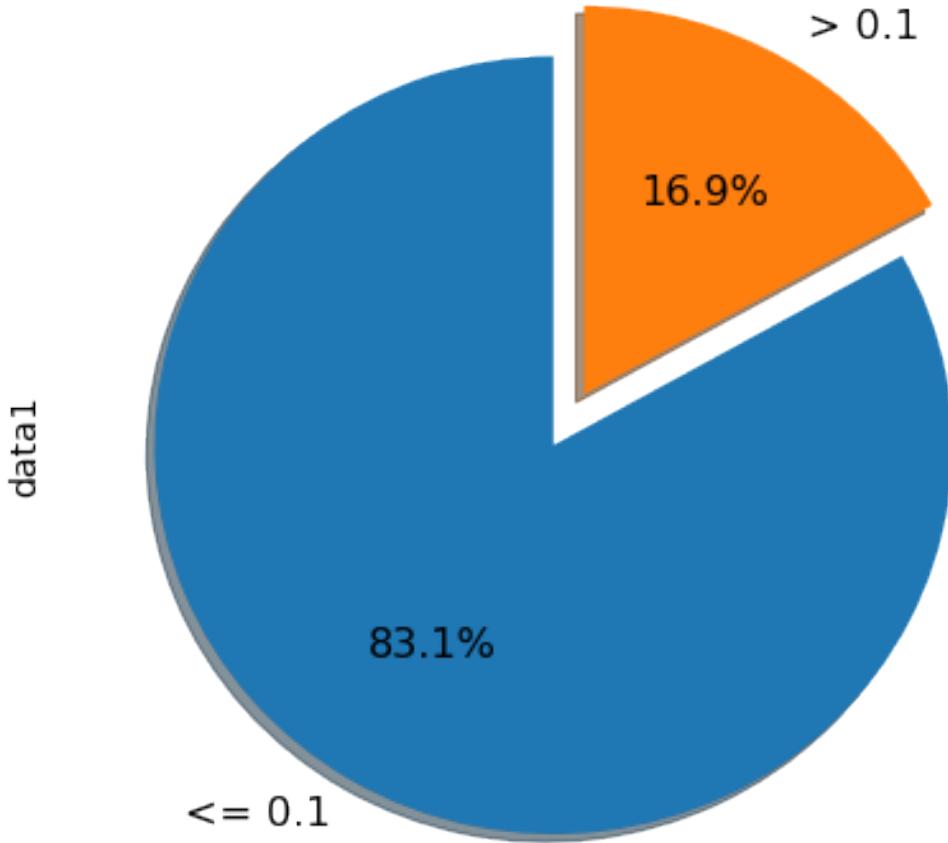
```
In [50]: gt01 = df['data1'] > 0.1
piecounts = gt01.value_counts()
piecounts
```

Out[50] :

	data1
False	831
True	169

```
In [51]: piecounts.plot(kind='pie',
figsize=(7, 7),
```

```
explode=[0, 0.15],  
labels=['<= 0.1', '> 0.1'],  
autopct='%.1f%%',  
shadow=True,  
startangle=90,  
fontsize=16);
```



## Hexbin plot

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

```
In [52]: dat1 = np.random.normal((0, 0), 2, size=(1000, 2))  
dat2 = np.random.normal((9, 9), 3, size=(2000, 2))
```

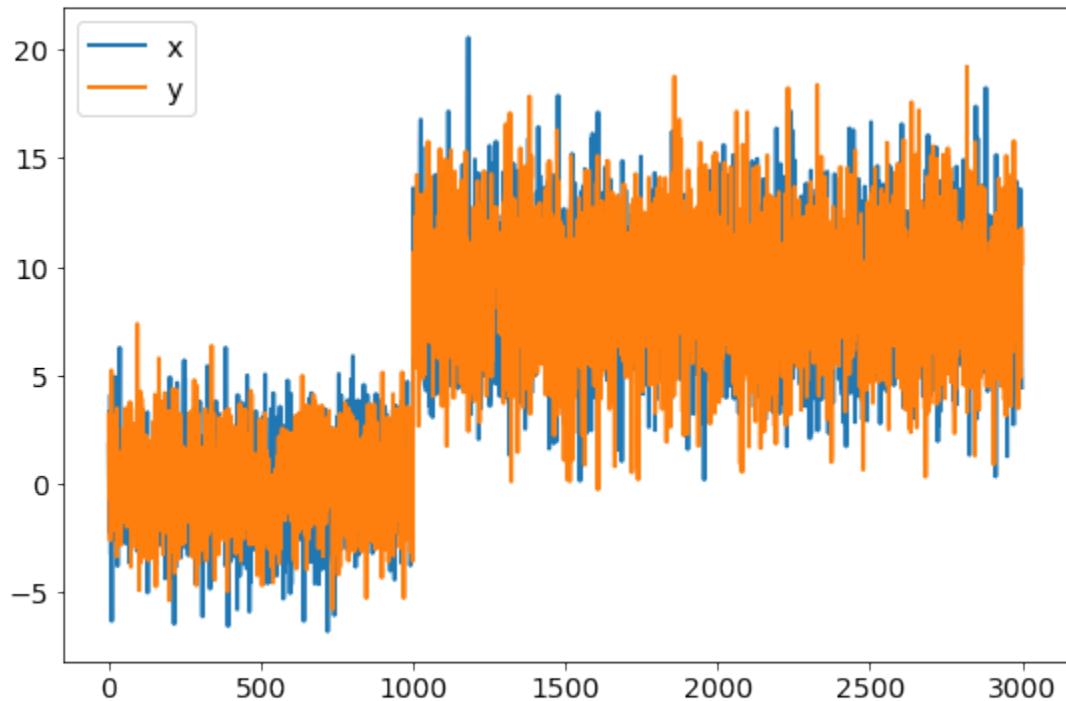
```
data = np.vstack([dat1, dat2])  
df = pd.DataFrame(data, columns=['x', 'y'])
```

In [53]: df.head()

Out[53] :

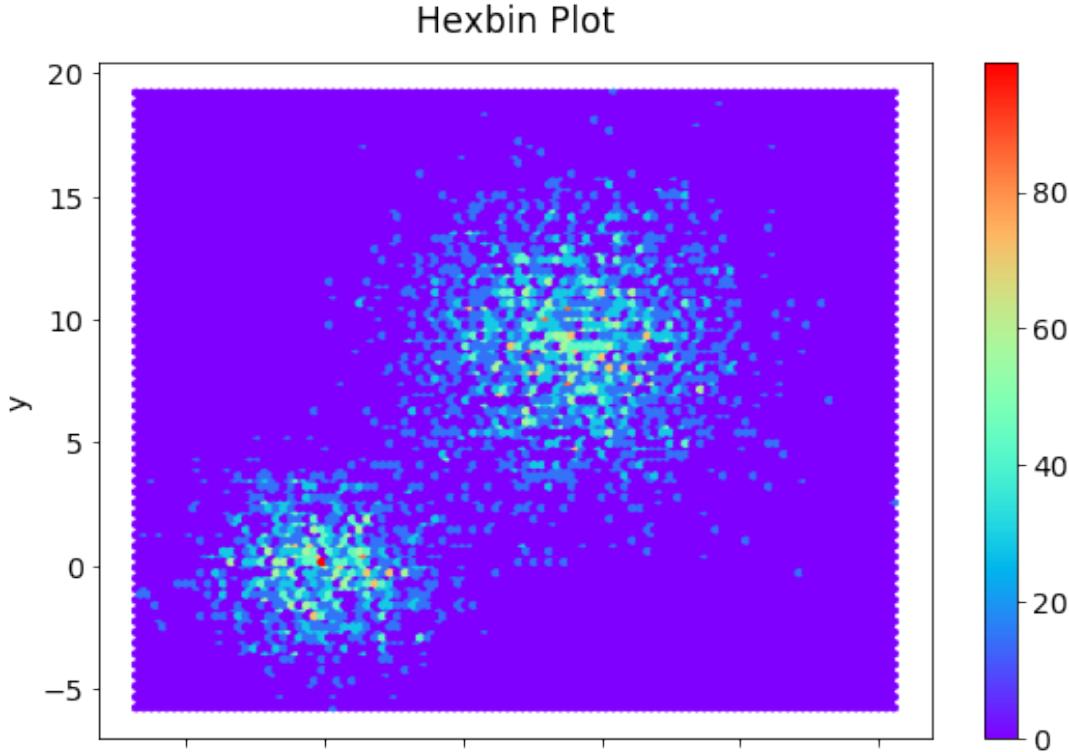
	x	y
0	1.795765	3.212868
1	-0.380623	-0.613050
2	3.183915	0.179148
3	3.315577	-0.039096
4	-2.252500	-0.930392

In [54]: df.plot();



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.

```
In [55]: df.plot(kind='hexbin', x='x', y='y', bins=100,
               cmap='rainbow', title='Hexbin Plot');
```



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.

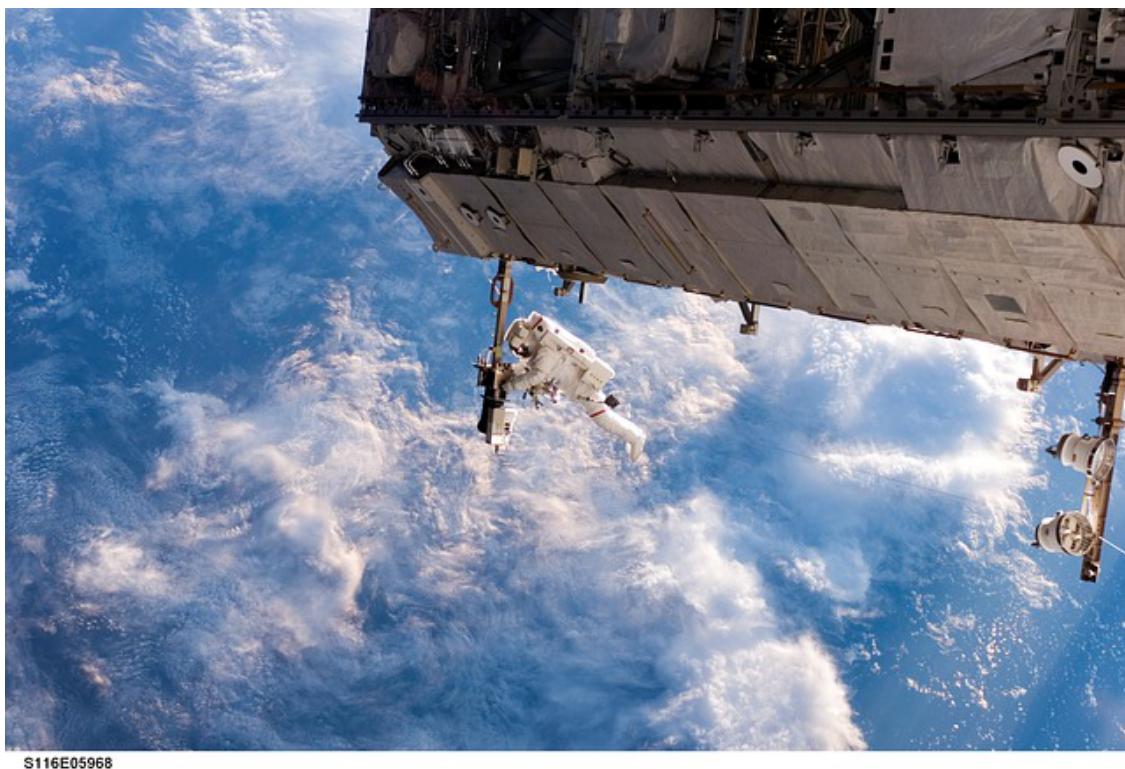
## Images

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

```
In [56]: from PIL import Image
```

```
In [57]: img = Image.open('../data/iss.jpg')
img
```

Out [57] :



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.

```
In [58]: imgarray = np.asarray(img)
```

```
In [59]: imgarray.shape
```

```
Out[59]: (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.

```
In [60]: imgarray.ravel().shape
```

```
Out[60]: (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.

```
In [61]: from scipy.io import wavfile
```

```
In [62]: rate, snd = wavfile.read(filename='..../data/sms.wav')
```

We can play the audio file in the notebook:

```
In [63]: from IPython.display import Audio
```

```
In [64]: Audio(data=snd, rate=rate)
```

```
Out[64]: <IPython.lib.display.Audio object>
```

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

```
In [65]: len(snd)
```

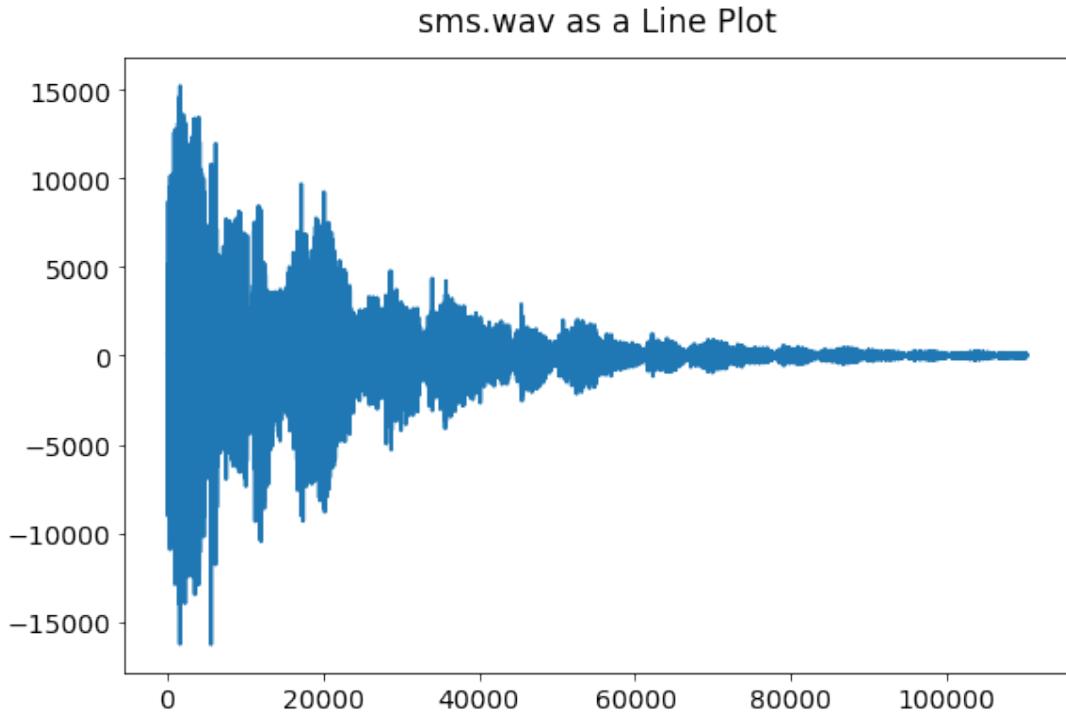
```
Out[65]: 110250
```

```
In [66]: snd
```

```
Out[66]: array([70, 14, 27, ..., 58, 68, 59], dtype=int16)
```

We can use matplotlib to plot the sound like this:

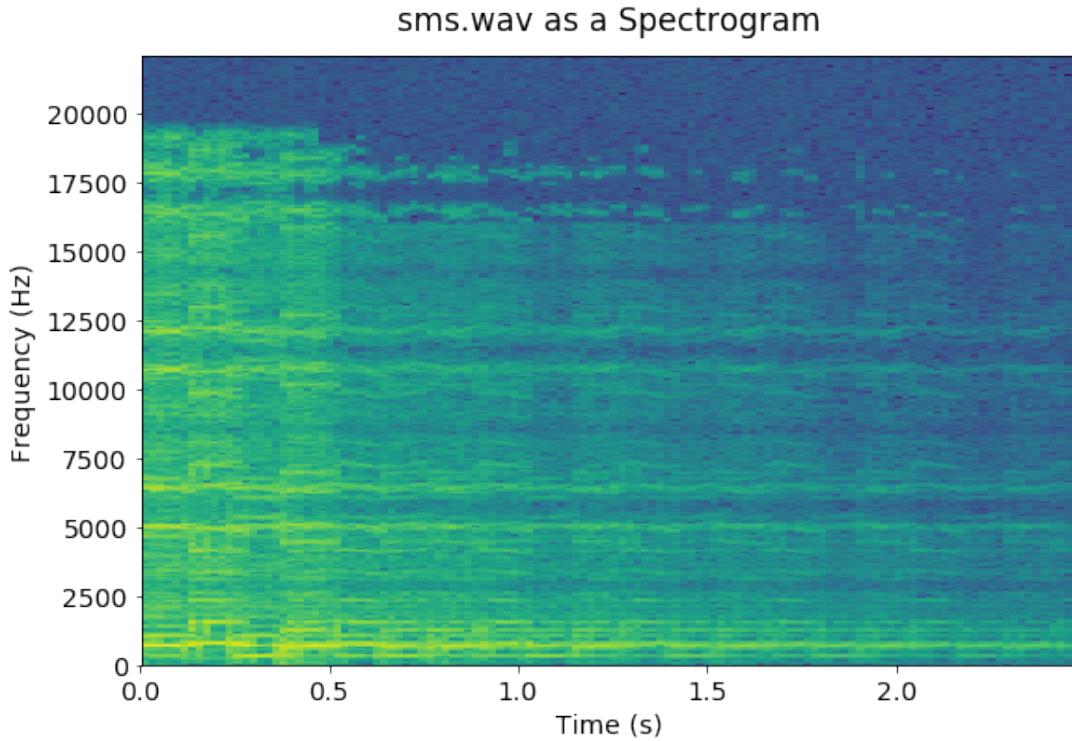
```
In [67]: plt.plot(snd)
plt.title('sms.wav as a Line Plot');
```



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 task like music recognition or speech-to-text.

```
In [68]: _ = plt.specgram(snd, NFFT=1024, Fs=44100)
plt.ylabel('Frequency (Hz)')
plt.xlabel('Time (s)')
plt.title('sms.wav as a Spectrogram');
```



## 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`

# 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 use 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

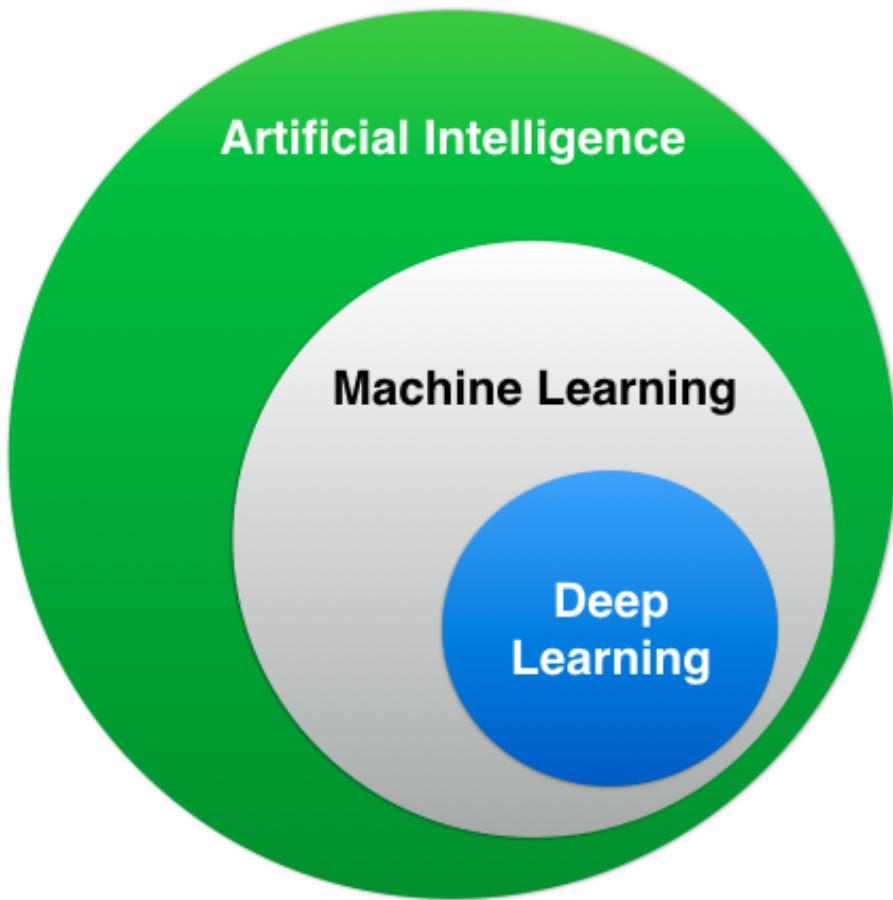
Machine Learning is a branch of Artificial Intelligence that develops computer programs that are able to learn patterns and rules from data. Although its origins can be traced back to the early days of modern computer science, only in the last decade has Machine Learning become a **fundamental tool for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

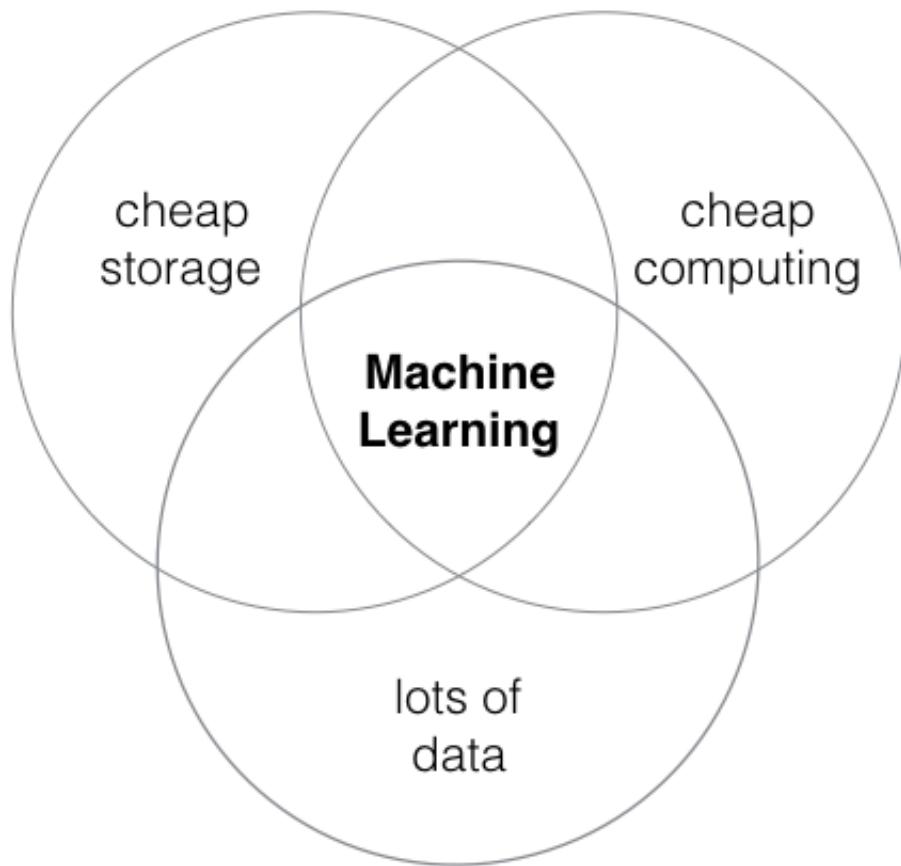
This revolution has largely been possible thanks to the combination of **3 factors**:

- 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



Deep Learning is a branch of Artificial Intelligence



3 Enablers of the Machine Learning revolution

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

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

Different types of learning

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 millions 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.

**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?

The movie is great!



The movie is about AI.



The movie is terrible!



Sentiment of a text sentence

## Configuration File

As promised in earlier chapter, from this chapter onwards we'll bundle common packages and configurations in a single config file that we load at the beginning of the chapter. Let's go ahead and load it:

```
In [1]: with open('common.py') as fin:  
    msg = fin.read()
```

Let's take a look at its content:

```
In [2]: print(msg)
```

```
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
  
pd.set_option("display.max_rows", 13)  
pd.set_option('display.max_columns', 11)  
pd.set_option("display.latex.repr", True)
```

To execute this file we use the `exec` function:

```
In [3]: exec(msg)
```

We have now loaded `pyplot`, `pandas` and `numpy` and we have set a few configuration parameters. Let's also load the configuration for `matplotlib`. For some reason this must be executed in a separate cell or it won't work:

```
In [4]: with open('matplotlibconf.py') as fin:
    exec(fin.read())
```

We are ready to roll!

## 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:

```
In [5]: df = pd.read_csv('../data/weight-height.csv')
```

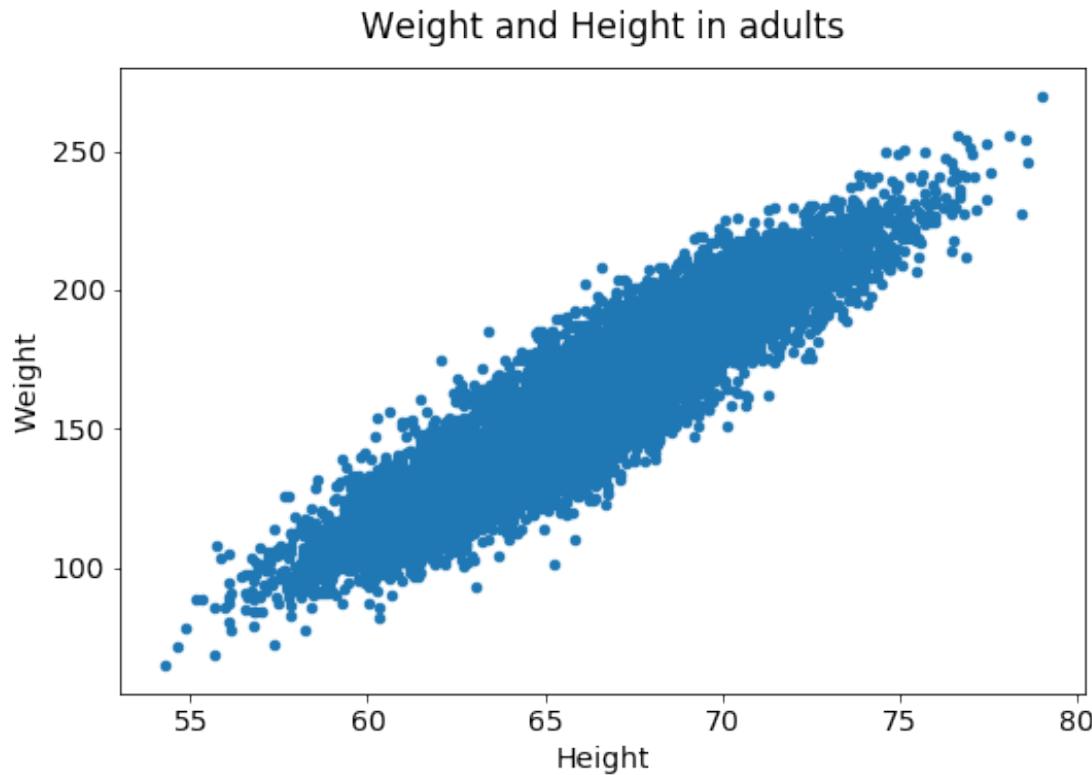
```
In [6]: df.head()
```

Out[6] :

	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

```
In [7]: def plot_humans():
    df.plot(kind='scatter',
            x='Height',
            y='Weight',
            title='Weight and Height in adults')

plot_humans()
```

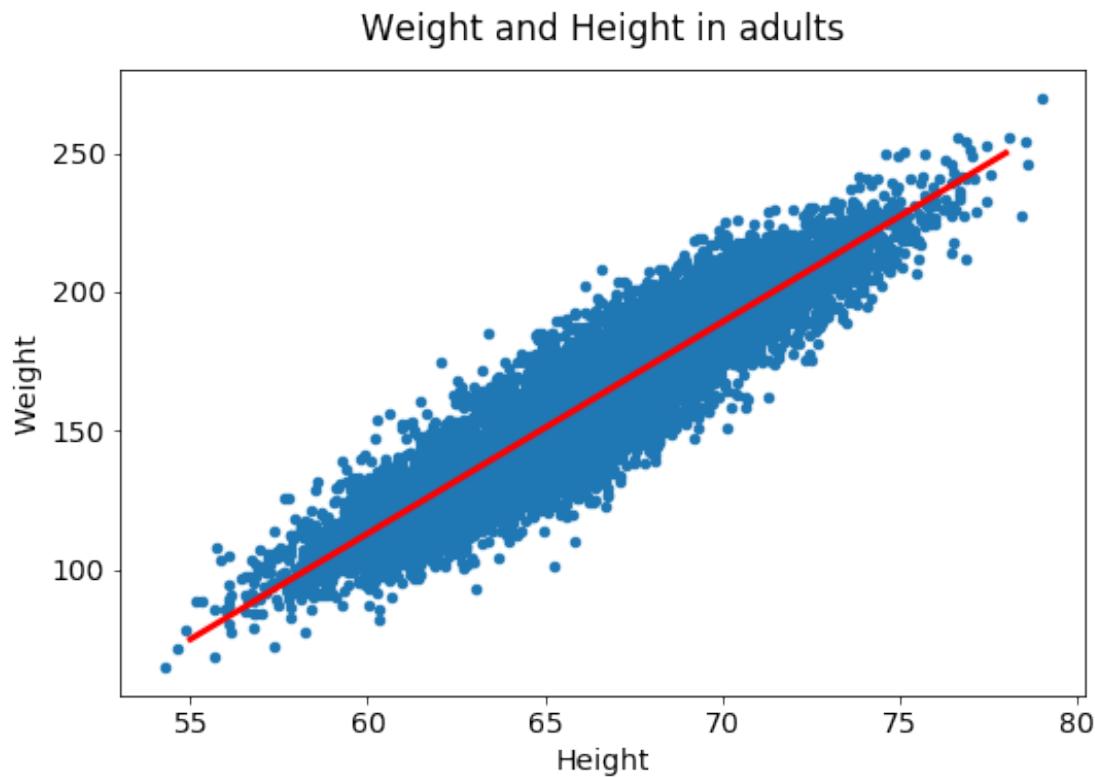


Can we tell if there is a pattern in how the dots are laid out on the graph or do they seem completely randomly spread? Our visual cortex, a great pattern recognizer, tells us that there is a pattern: dots are roughly spread around a straight diagonal line. This line seems to indicate the obvious: taller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

```
In [8]: plot_humans()
```

```
# Here we plot 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);
```



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:

$$\hat{y} = b + Xw \quad (3.1)$$

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  $w = 1$ , 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  $\epsilon$ :

$$y = b + Xw + \epsilon = \hat{y} + \epsilon \quad (3.2)$$

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:

```
In [9]: 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):

```
In [10]: x = np.linspace(55, 80, 100)
x
```

```
Out[10]: 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,
   66.36363636, 66.61616162, 66.86868687, 67.12121212, 67.37373737,
   67.62626263, 67.87878788, 68.13131313, 68.38383838, 68.63636364,
   68.88888889, 69.14141414, 69.39393939, 69.64646465, 69.8989899 ,
   70.15151515, 70.4040404 , 70.65656566, 70.90909091, 71.16161616,
```

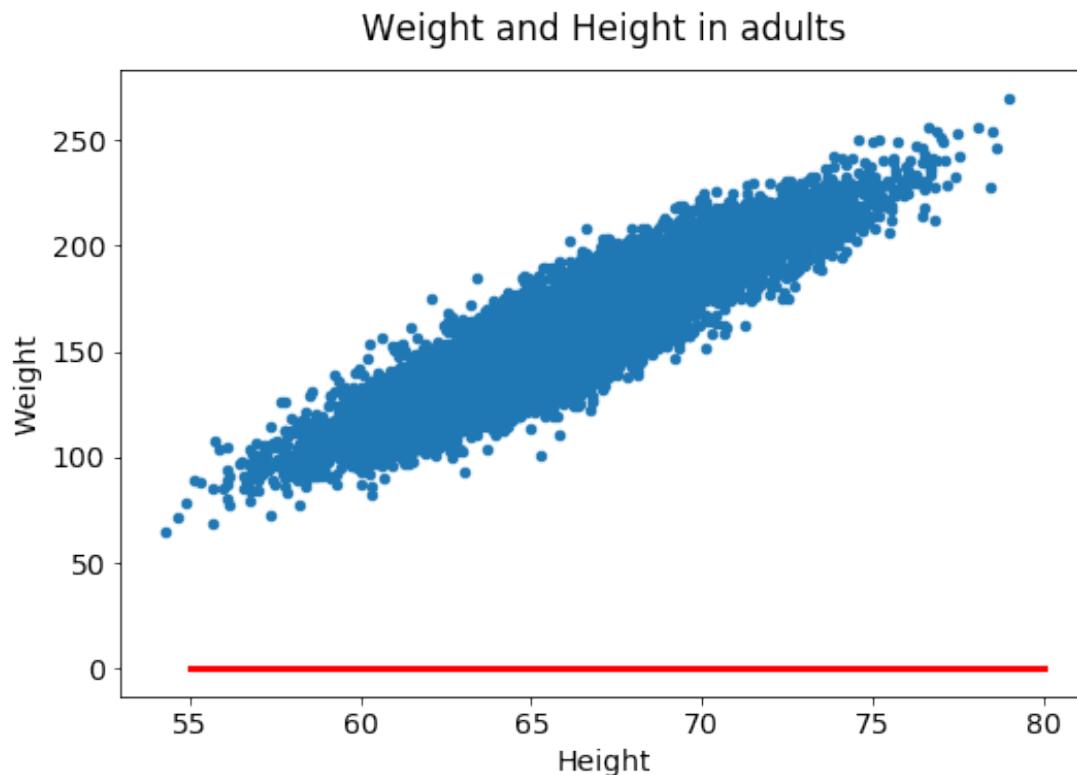
71.41414141, 71.66666667, 71.91919192, 72.17171717, 72.42424242,  
 72.67676768, 72.92929293, 73.18181818, 73.43434343, 73.68686869,  
 73.93939394, 74.19191919, 74.44444444, 74.6969697, 74.94949495,  
 75.2020202, 75.45454545, 75.70707071, 75.95959596, 76.21212121,  
 76.46464646, 76.71717172, 76.96969697, 77.22222222, 77.47474747,  
 77.72727273, 77.97979798, 78.23232323, 78.48484848, 78.73737374,  
 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:

```
In [11]: yhat = line(x, w=0, b=0)
        yhat
```

```
In [12]: plot_humans()
```

```
plt.plot(x, yhat, color='red', linewidth=3);
```

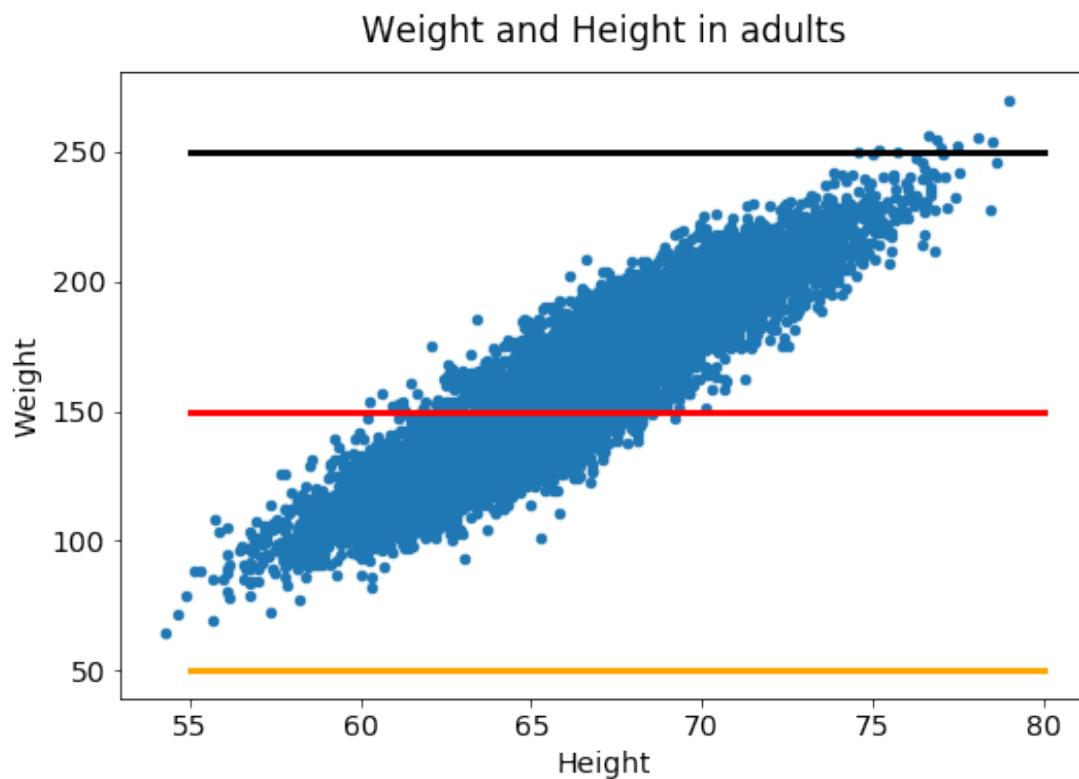


So we've drawn a horizontal line as our model. This is not really a good model for our data! It would be a good model if everyone in our population was floating in space and therefore measured o weight regardless of their height. Fun, but not accurate for our chart! See how far the line is from our data.

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).

```
In [13]: 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='black', linewidth=3);
```

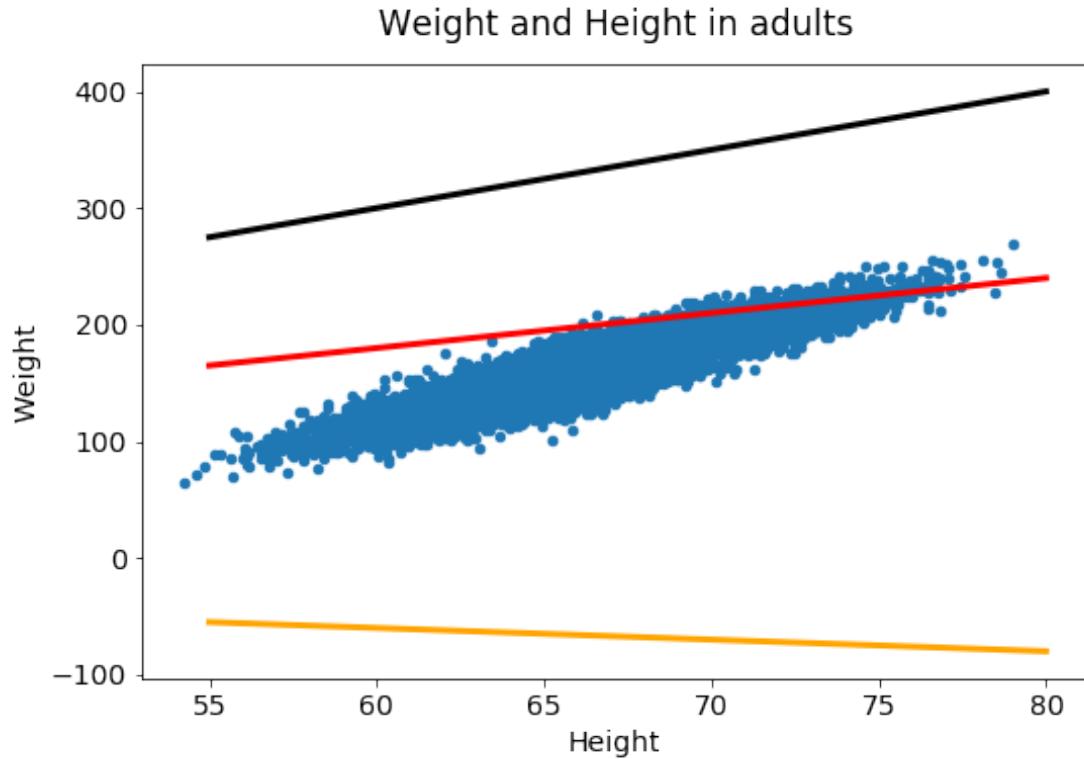


This would be a good model only if we had a broken scale that always returned a fixed value, regardless of who steps on it. Also not accurate.

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**.

```
In [14]: plot_humans()
```

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



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

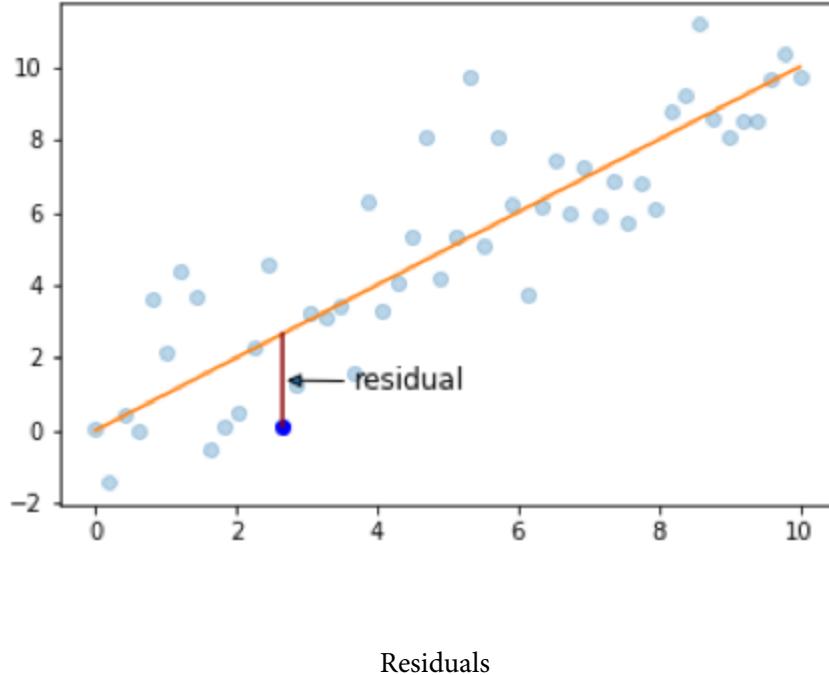
In order to find the best possible linear model to describe our data, we need to define a criterion to evaluate the “goodness” of a particular model.

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:

$$\epsilon_i = y_i - \hat{y}_i \quad (3.3)$$

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:

$$\text{Total Error} = \sum_i |\epsilon_i| = \sum_i |y_i - \hat{y}_i| \quad (3.4)$$

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:

$$\text{MSE} = \frac{1}{N} \sum_i (y_i - \hat{y}_i)^2 \quad (3.5)$$

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:

```
In [15]: 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  $X$ :

```
In [16]: X = df[['Height']].values
X
```

```
Out[16]: array([[73.84701702],
                 [68.78190405],
                 [74.11010539],
                 ...,
                 [63.86799221],
                 [69.03424313],
                 [61.94424588]])
```

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

```
In [17]: X.shape
```

```
Out[17]: (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:

```
In [18]: y_true = df['Weight'].values
y_true
```

```
Out[18]: array([241.89356318, 162.31047252, 212.74085556, ..., 128.47531878,
   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  $X$ :

```
In [19]: y_pred = line(X)
y_pred
```

```
Out[19]: array([[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:

```
In [20]: mean_squared_error(y_true, y_pred.ravel())
```

```
Out[20]: 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:

```
In [21]: y_pred = line(X, w=2)
mean_squared_error(y_true, y_pred.ravel())
```

```
Out[21]: 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  $b$ :

```
In [22]: y_pred = line(X, w=2, b=20)
    mean_squared_error(y_true, y_pred.ravel())
```

Out[22]: 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  $b$ .

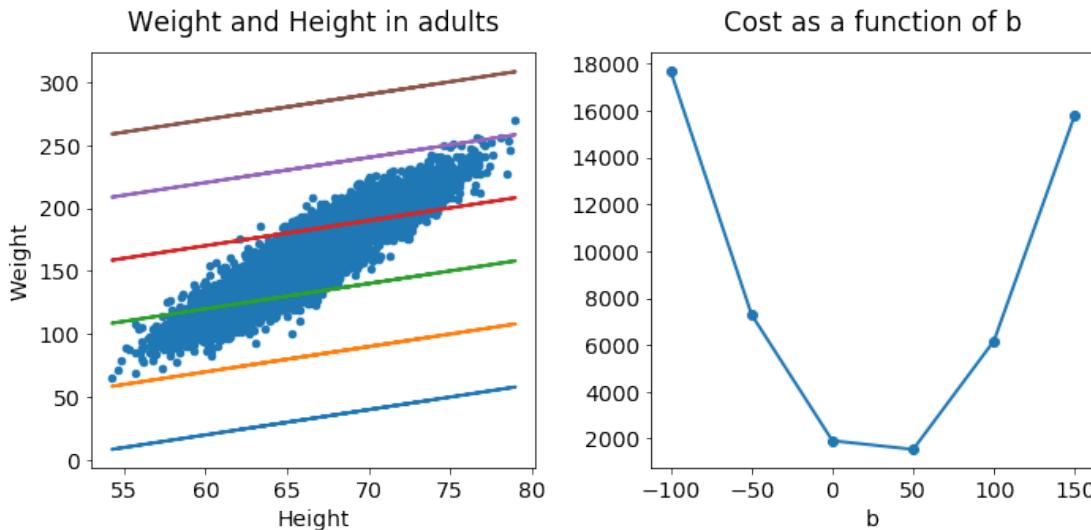
```
In [23]: plt.figure(figsize=(10, 5))

# 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 = [] # append the values of the cost here
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')
plt.tight_layout();
```



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.

```
In [24]: from keras.models import Sequential
         from keras.layers import Dense
         from keras.optimizers import Adam, SGD
```

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.

```
In [25]: 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:

```
In [26]: model.add(Dense(1, input_shape=(1,)))
```

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

```
In [27]: 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.

```
In [28]: 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 ( $X$ ,  $y_{\text{true}}$ ) dataset 40 times.

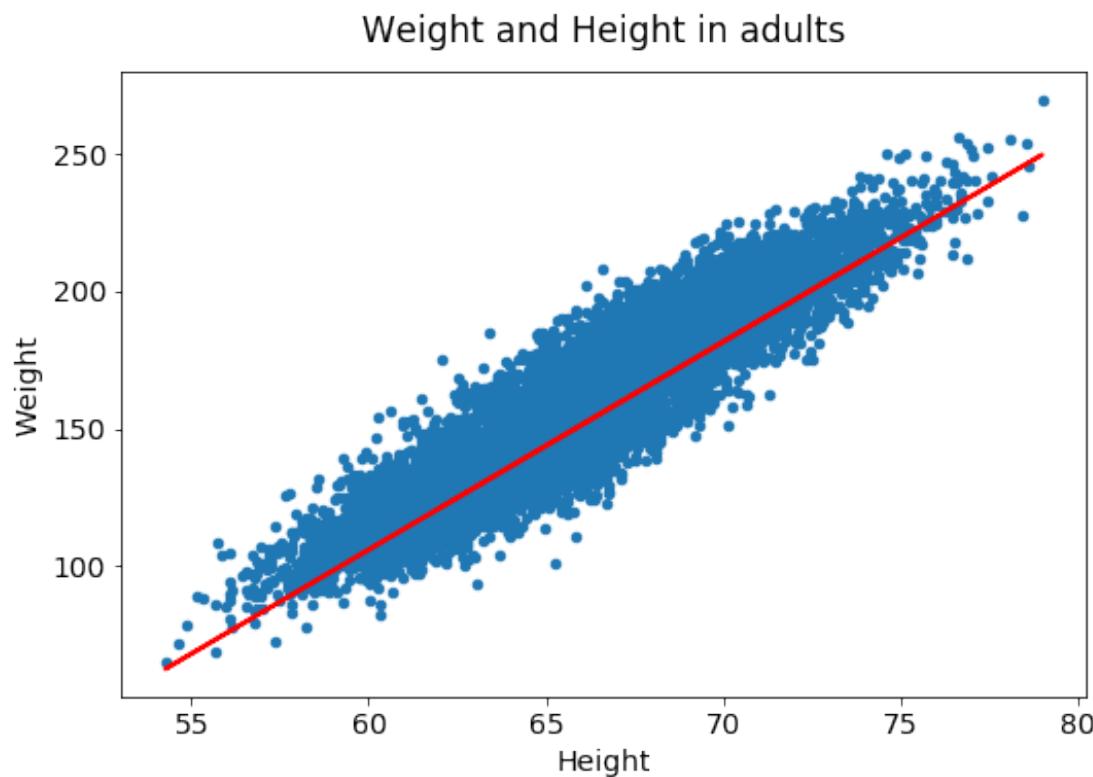
```
In [29]: model.fit(X, y_true, epochs=40, verbose=0)
```

```
Out[29]: <keras.callbacks.History at 0x7ff0f1e16d68>
```

Let's see how well the model fits our data. We will store our predictions in a variable called  $y_{\text{pred}}$  and plot them over the data:

```
In [30]: y_pred = model.predict(X)
```

```
In [31]: df.plot(kind='scatter',
                 x='Height',
                 y='Weight',
                 title='Weight and Height in adults')
plt.plot(X, y_pred, color='red');
```



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  $w$ .

```
In [32]: W, B = model.get_weights()
```

```
In [33]: W
```

```
Out[33]: array([[7.5778575]], dtype=float32)
```

```
In [34]: B
```

```
Out[34]: array([-348.83707], 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.

```
In [35]: w = W[0, 0]
```

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

```
In [36]: 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:

$$0 = Xw + b \quad (3.6)$$

and we can shuffle things around to obtain:

$$X = \frac{-b}{w} \quad (3.7)$$

Let's calculate it:

In [37] : -b/w

Out [37] : 46.033733

So this model only makes sense for people who are at least about 45 inches tall. If you are shorter than 46 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:

In [38] : `from sklearn.metrics import r2_score`

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

```
In [39]: r = r2_score(y_true, y_pred)
        print("The R2 score is {:.3f}".format(r))
```

The R2 score is 0.802

TIP: In the last command we introduced a way to define [Python format](#), to make numbers more readable. In particular, we specified the format `{:0.3f}`. 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.

```
In [40]: from sklearn.model_selection import train_test_split
```

```
In [41]: X_train, X_test, y_train, y_test = \
        train_test_split(X, y_true, test_size=0.2)
```

```
In [42]: len(X_train)
```

Out[42]: 8000

In [43]: `len(X_test)`

Out[43]: 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):

In [44]: `model.fit(X_train, y_train, epochs=50, verbose=0)`

Out[44]: <keras.callbacks.History at 0x7fefac180b70>

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.

In [45]: `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.

In [46]: `from sklearn.metrics import mean_squared_error as mse`

In [47]: `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): 152.5
Mean Squared Error (Test set): 164.9
```

```
In [48]: 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): 0.851
R2 score (Test set): 0.843
```

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:

```
In [49]: df = pd.read_csv('../data/user_visit_duration.csv')
```

```
In [50]: df.head()
```

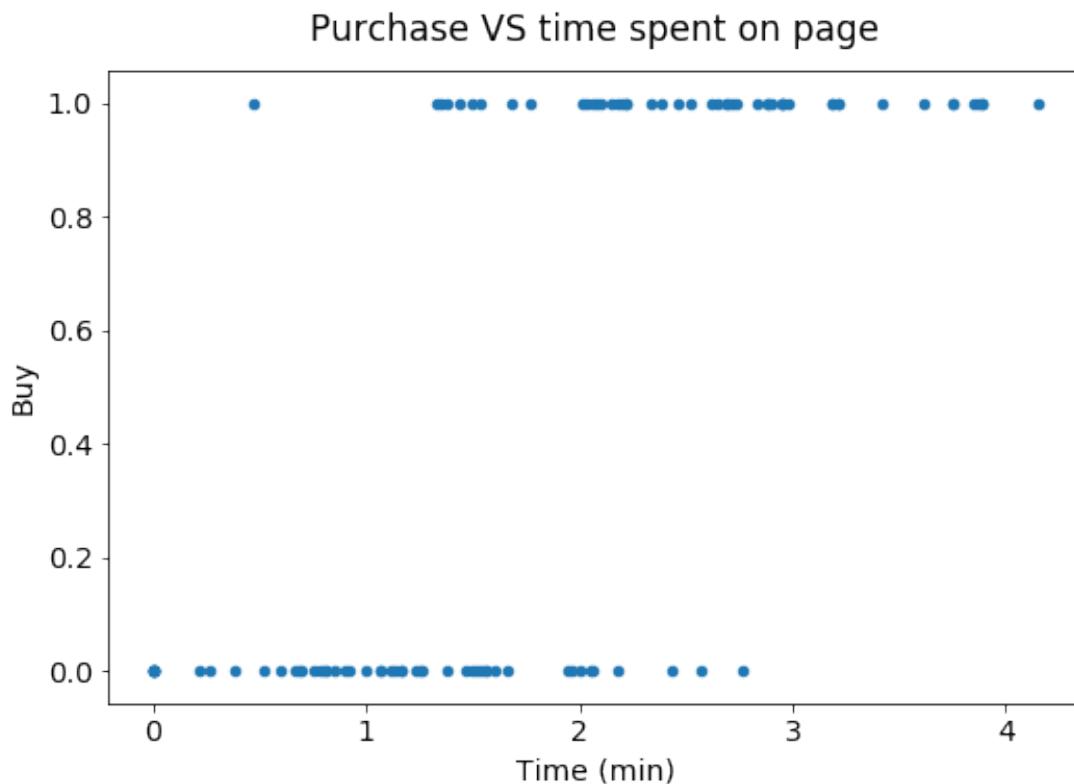
```
Out[50] :
```

	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:

```
In [51]: df.plot(kind='scatter', x='Time (min)', y='Buy',  
title='Purchase VS time spent on page');
```



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.

```
In [52]: 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:

```
In [53]: model.set_weights([[1.0]], [0.])
```

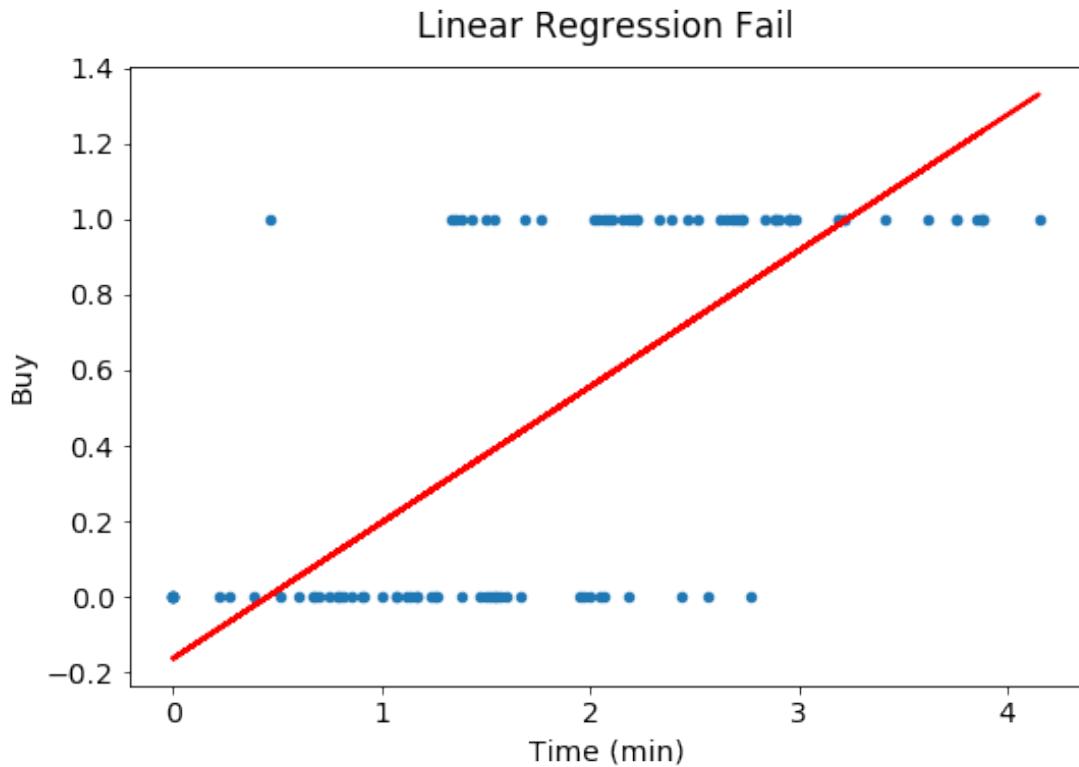
Then we fit the model on  $X$  and  $y$  for 200 epochs, suppressing the output with `verbose=0`:

```
In [54]: model.fit(X, y, epochs=200, verbose=0);
```

Let's see what the predictions look like:

```
In [55]: 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');
```



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.

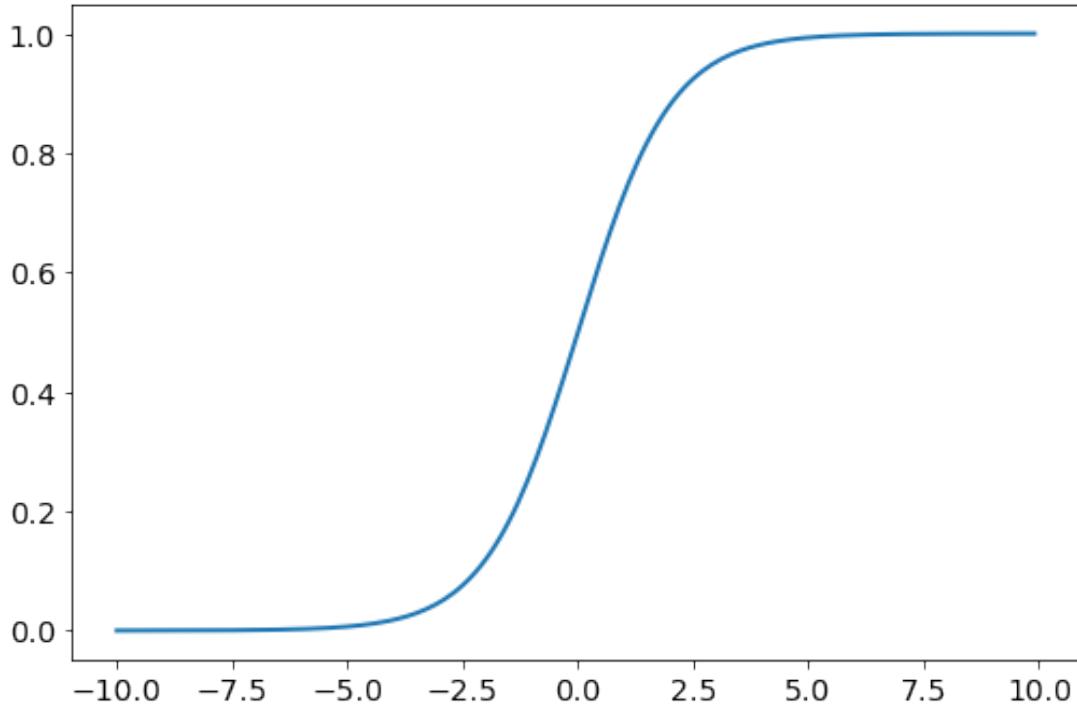
Let us first start by defining a nonlinear hypothesis. We need a nonlinear function that will map all of the real axis into the interval  $[0, 1]$ . There are many such functions and we will see a few in the next chapters. A simple, smooth and well-behaved function is the **Sigmoid function**:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (3.8)$$

which looks like this:

```
In [56]: def sigmoid(z):
    return 1.0/(1.0 + np.exp(-z))

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



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:

$$\text{Buy} = \frac{1}{1 + e^{-(\text{Time } w+b)}} \quad (3.9)$$

or

$$\hat{y} = \sigma(Xw + b) \quad (3.10)$$

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:

```
In [57]: x = np.linspace(-10, 10, 100)

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

plt.subplot(121)

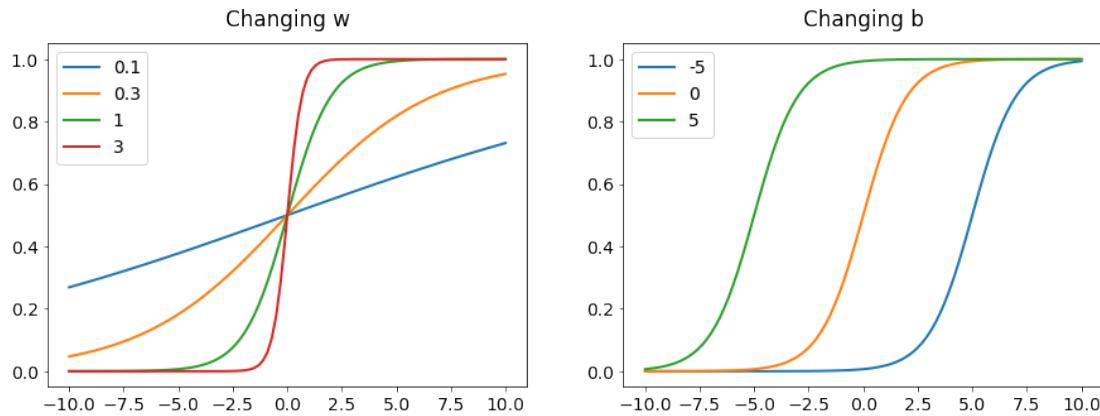
ws = [0.1, 0.3, 1, 3]
for w in ws:
    plt.plot(x, sigmoid(line(x, w=w)))

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');
```



## 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:

$$c_i = -y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i) \quad (3.11)$$

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

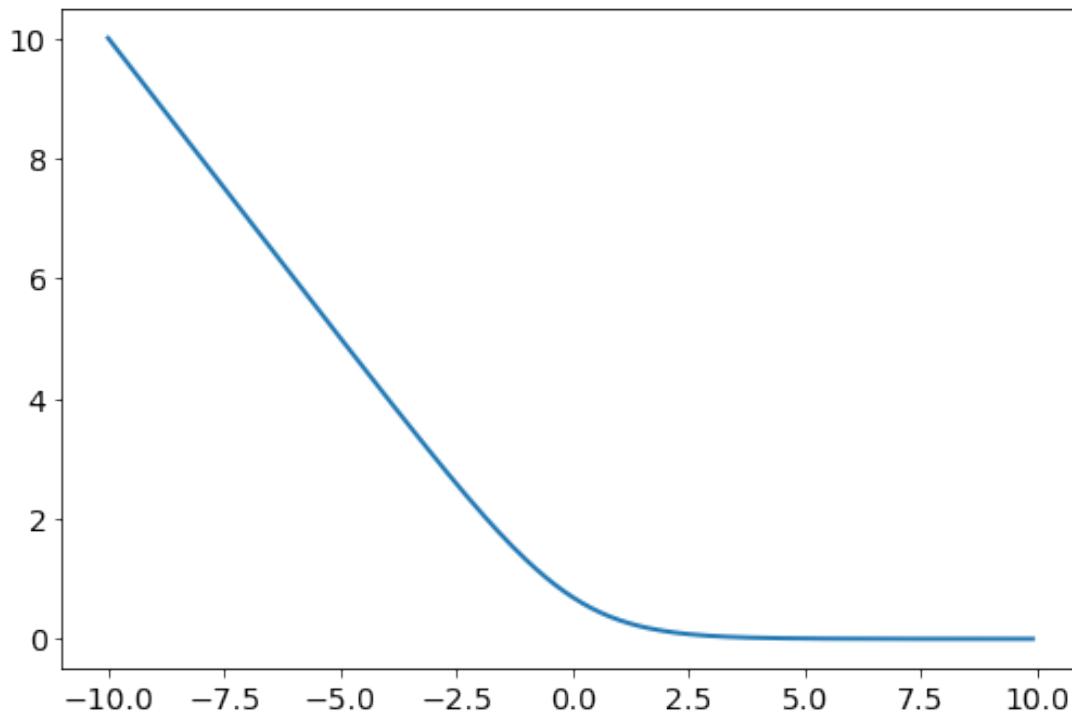
$$c_i = \begin{cases} -\log(\hat{y}_i) & \text{for } y_i = 1 \\ -\log(1 - \hat{y}_i) & \text{for } y_i = 0 \end{cases} \quad (3.12)$$

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:

$$\log(\sigma(z)) = -\log(1) + \log(1 + e^{-z}) = \log(1 + e^{-z}) \quad (3.13)$$

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

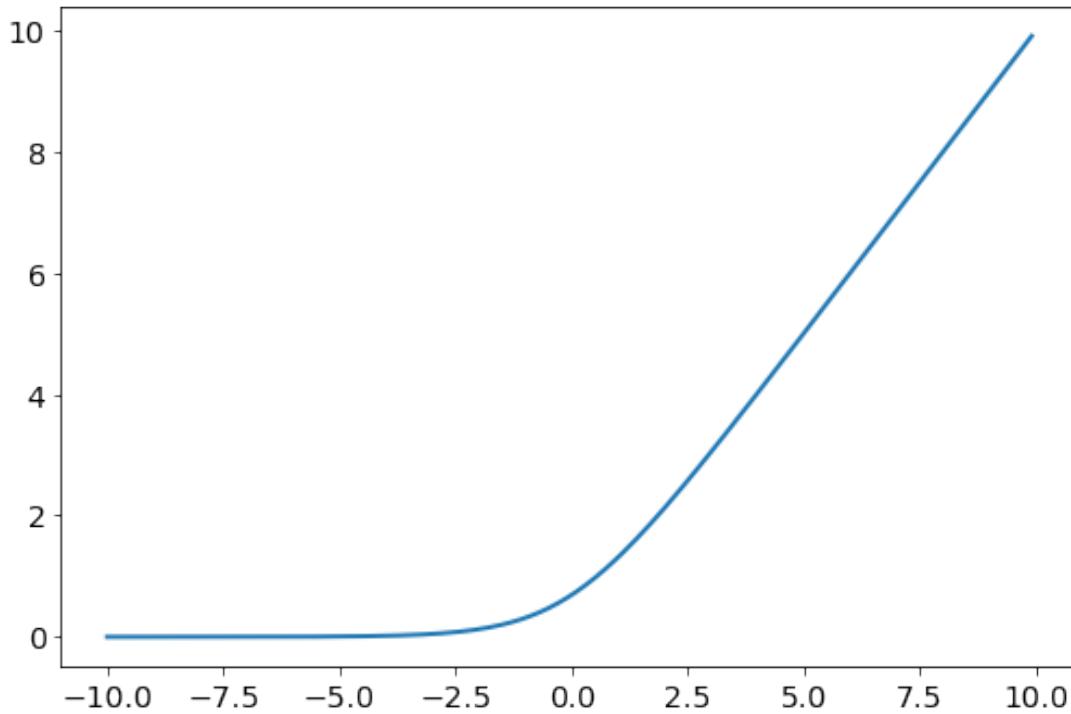
```
In [58]: plt.plot(z, -np.log(sigmoid(z)));
```



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.

```
In [59]: plt.plot(z, -np.log(1 - sigmoid(z)));
```



Now that we have defined the cost for a single point, we can define the average cost as:

$$c = \frac{1}{N} \sum_i c_i \quad (3.14)$$

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:

```
In [60]: model = Sequential()
```

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

We can add the activation as a layer:

```
In [61]: from keras.layers import Activation
```

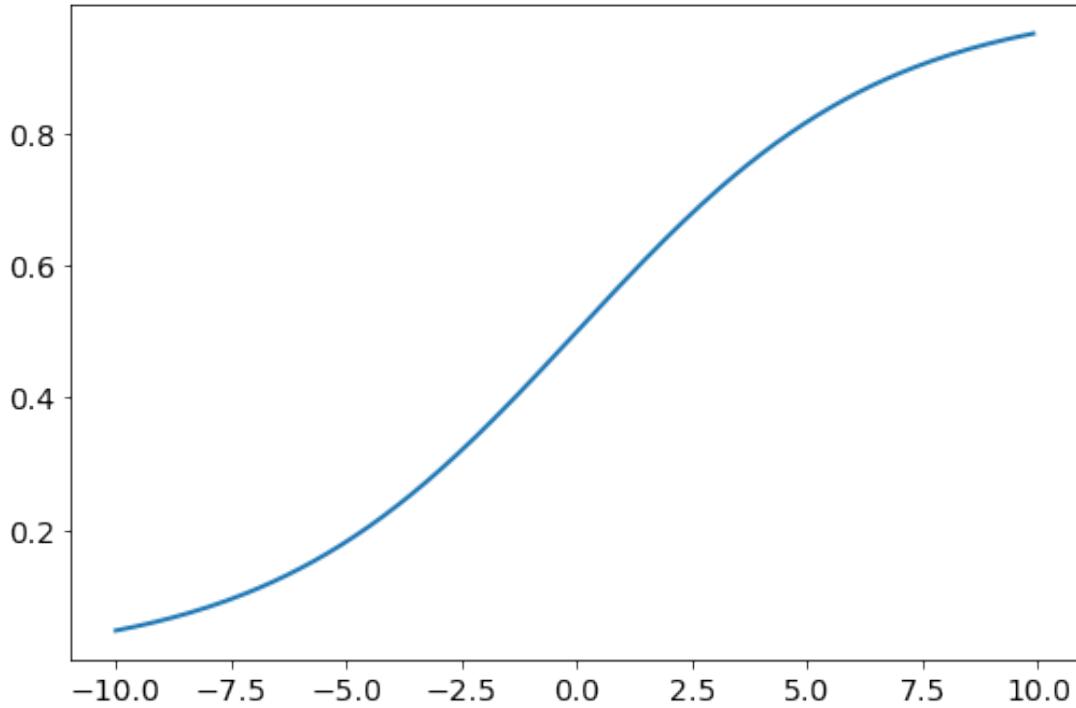
```
In [62]: model.add(Activation('sigmoid'))
```

```
In [63]: 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  
Non-trainable params: 0  
-----
```

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:

```
In [64]: plt.plot(z, model.predict(z));
```



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:

```
In [65]: model.compile(optimizer=SGD(lr=0.5),
                      loss='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.

```
In [66]: model.fit(X, y, epochs=25);

Epoch 1/25
100/100 [=====] - 0s 1ms/step - loss: 0.6364 - acc: 0.5600
Epoch 2/25
100/100 [=====] - 0s 75us/step - loss: 0.5843 - acc: 0.6400
Epoch 3/25
100/100 [=====] - 0s 77us/step - loss: 0.5547 - acc: 0.7000
Epoch 4/25
100/100 [=====] - 0s 76us/step - loss: 0.5450 - acc: 0.6900
Epoch 5/25
100/100 [=====] - 0s 76us/step - loss: 0.5059 - acc: 0.8000
Epoch 6/25
100/100 [=====] - 0s 74us/step - loss: 0.4946 - acc: 0.8100
Epoch 7/25
100/100 [=====] - 0s 79us/step - loss: 0.4847 - acc: 0.7900
Epoch 8/25
100/100 [=====] - 0s 74us/step - loss: 0.4809 - acc: 0.8100
Epoch 9/25
100/100 [=====] - 0s 79us/step - loss: 0.4655 - acc: 0.8200
Epoch 10/25
100/100 [=====] - 0s 77us/step - loss: 0.4522 - acc: 0.8300
Epoch 11/25
100/100 [=====] - 0s 75us/step - loss: 0.4454 - acc: 0.8200
Epoch 12/25
100/100 [=====] - 0s 73us/step - loss: 0.4475 - acc: 0.8000
Epoch 13/25
100/100 [=====] - 0s 75us/step - loss: 0.4290 - acc: 0.8300
Epoch 14/25
100/100 [=====] - 0s 75us/step - loss: 0.4240 - acc: 0.8300
Epoch 15/25
```

```

100/100 [=====] - 0s 75us/step - loss: 0.4215 - acc: 0.8100
Epoch 16/25
100/100 [=====] - 0s 76us/step - loss: 0.4177 - acc: 0.8300
Epoch 17/25
100/100 [=====] - 0s 75us/step - loss: 0.4444 - acc: 0.8100
Epoch 18/25
100/100 [=====] - 0s 72us/step - loss: 0.4188 - acc: 0.8200
Epoch 19/25
100/100 [=====] - 0s 75us/step - loss: 0.4111 - acc: 0.8200
Epoch 20/25
100/100 [=====] - 0s 79us/step - loss: 0.4126 - acc: 0.8100
Epoch 21/25
100/100 [=====] - 0s 76us/step - loss: 0.4263 - acc: 0.8300
Epoch 22/25
100/100 [=====] - 0s 74us/step - loss: 0.4067 - acc: 0.8400
Epoch 23/25
100/100 [=====] - 0s 77us/step - loss: 0.3979 - acc: 0.8200
Epoch 24/25
100/100 [=====] - 0s 76us/step - loss: 0.4009 - acc: 0.8300
Epoch 25/25
100/100 [=====] - 0s 75us/step - loss: 0.4018 - acc: 0.8400

```

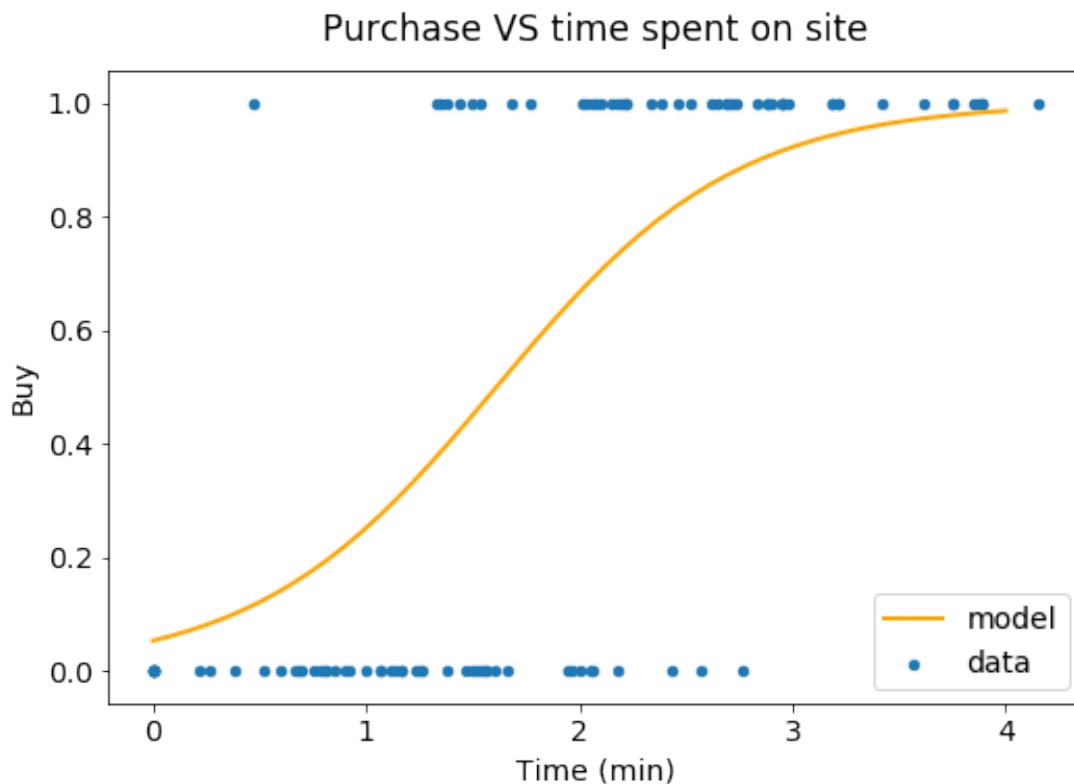
The model seems to have converged because the loss does not seem to improve in the last epochs. Let's see what the predictions look like:

```

In [67]: ax = df.plot(kind='scatter', x='Time (min)', y='Buy',
                     title='Purchase VS time spent on site')

temp = np.linspace(0, 4)
ax.plot(temp, model.predict(temp), color='orange')
plt.legend(['model', 'data']);

```



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.

```
In [68]: y_pred = model.predict(X)
```

```
In [69]: 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`:

```
In [70]: from sklearn.metrics import accuracy_score
```

```
In [71]: acc = accuracy_score(y, y_class_pred)
print("Accuracy score: {:.3f}".format(acc))
```

```
Accuracy score: 0.830
```

## 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:

```
In [72]: 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.

```
In [73]: params = model.get_weights()
```

```
In [74]: params
```

```
Out[74]: [array([[1.7809732]]), array([-2.8685334]), dtype=float32]
```

```
In [75]: params = [np.zeros(w.shape) for w in params]
```

```
In [76]: params
```

```
Out[76]: [array([[0.]]), array([0.])]
```

```
In [77]: model.set_weights(params)
```

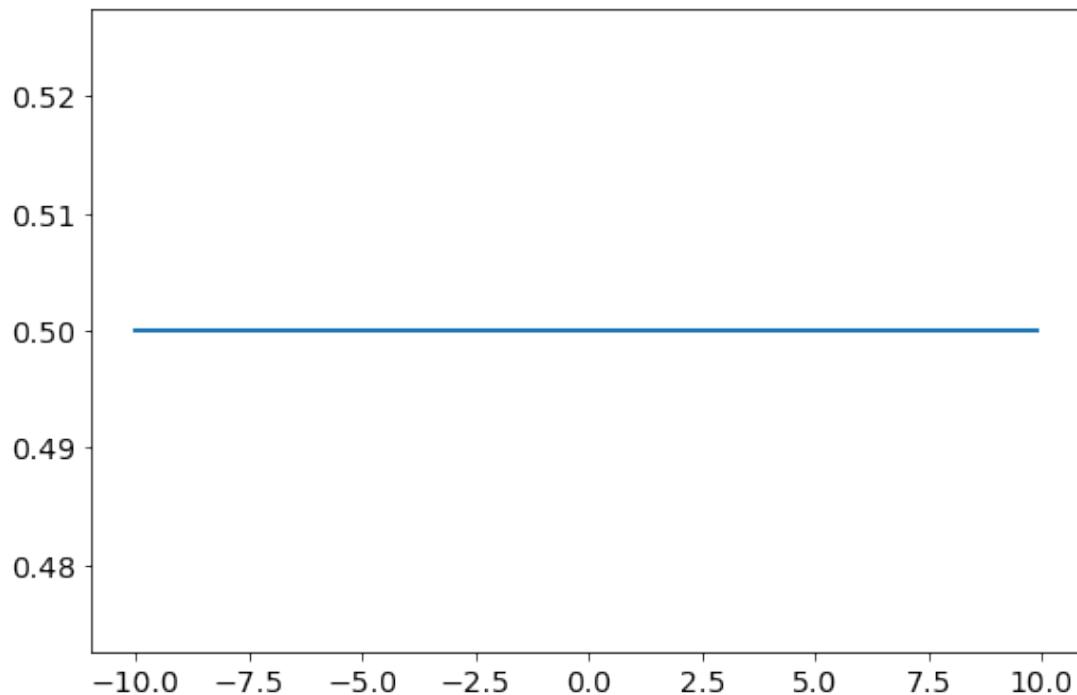
Let's check that the model is now predicting garbage:

```
In [78]: 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:

```
In [79]: plt.plot(z, model.predict(z));
```



Let's re-train it on the training data only

```
In [80]: model.fit(X_train, y_train, epochs=25, verbose=0);
```

And let's check the accuracy score on training and test sets:

```
In [81]: y_pred_train_class = model.predict(X_train) > 0.5
acc = accuracy_score(y_train, y_pred_train_class)
print("Train accuracy score {:.3f}".format(acc))

y_pred_test_class = model.predict(X_test) > 0.5
acc = accuracy_score(y_test, y_pred_test_class)
print("Test accuracy score {:.3f}".format(acc))
```

```
Train accuracy score 0.825
Test accuracy score 0.850
```

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:

$$X \cdot w = x_{j0}w_0 + x_{j1}w_1 + x_{j2}w_2 + \dots = \sum_i x_{ji}w_i \text{ for each data point } j \quad (3.15)$$

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.

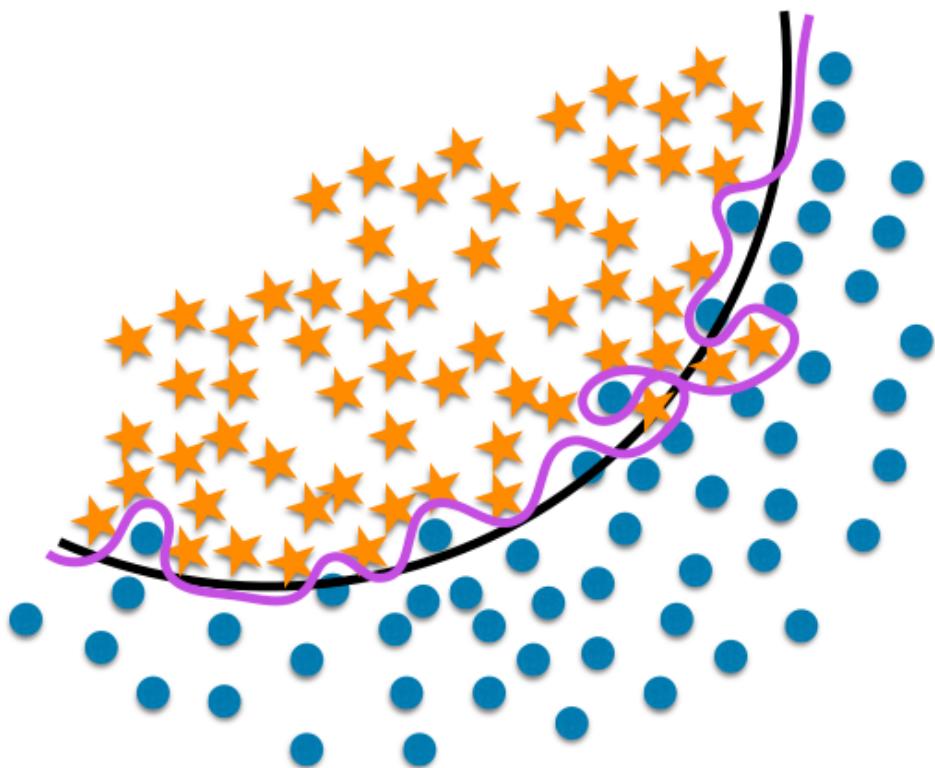
It is going to be very hard to overfit with a simple model with only one parameter, but as the number of parameters increases, the likelihood of overfitting increases as well. We'll need to watch out for our model *overfitting* the dataset.

### 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.



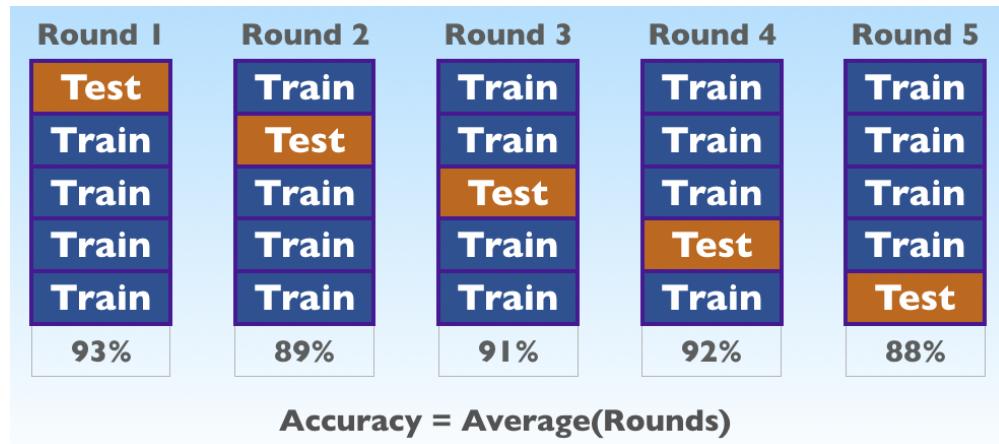
Overfitting in a classification problem

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**.



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`.

```
In [82]: from keras.wrappers.scikit_learn import KerasClassifier
```

```
In [83]: def build_logistic_regr():
    model = Sequential()
    model.add(Dense(1, input_dim=1,
                   activation='sigmoid'))

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

```
In [84]: model = KerasClassifier(build_fn=build_logistic_regr,
                               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:

```
In [85]: from sklearn.model_selection import cross_val_score
        from sklearn.model_selection import KFold
```

```
In [86]: cv = KFold(3, shuffle=True)
        scores = cross_val_score(model, X, y, cv=cv)
```

```
In [87]: scores
```

```
Out[87]: array([0.73529412, 0.87878788, 0.78787879])
```

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:

```
In [88]: m = scores.mean()
        s = scores.std()
        print("Cross Validation accuracy:",
              "{:0.4f} ± {:0.4f}".format(m, s))
```

```
Cross Validation accuracy: 0.8007 ± 0.0593
```

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:

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:

$$\text{Acc} = \frac{(\text{TP} + \text{TN})}{\text{All}} \quad (3.16)$$

On the other hand our model could be wrong in two ways.

	<i>Test Negative</i>	<i>Test Positive</i>
<i>Condition Negative</i>	<b>TRUE NEGATIVE</b>	<b>FALSE POSITIVE (Type I error)</b>
<i>Condition Positive</i>	<b>FALSE NEGATIVE (Type II error)</b>	<b>TRUE POSITIVE</b>

Confusion Matrix

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:

```
In [89]: from sklearn.metrics import confusion_matrix
```

We define a short helper function to add column and row labels for nice display:

```
In [90]: def pretty_cm(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
```

```
In [91]: pretty_cm(y, y_class_pred, ['Not Buy', 'Buy'])
```

Out[91]:

	Predicted Not Buy	Predicted Buy
Not Buy	40	10
Buy	7	43

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:

$$\text{Precision} = \frac{(\text{TP})}{\text{TP} + \text{FP}} \quad (3.17)$$

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:

$$\text{Recall} = \frac{(\text{TP})}{\text{TP} + \text{FN}} \quad (3.18)$$

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:

$$F_1 = 2 \frac{PR}{P + R} \quad (3.19)$$

$F_1$  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  $F_1$  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:

```
In [92]: from sklearn.metrics import precision_score, recall_score, f1_score
```

```
In [93]: precision = precision_score(y, y_class_pred)
print("Precision: \t{:0.3f}".format(precision))

recall = recall_score(y, y_class_pred)
print("Recall: \t{:0.3f}".format(recall))

f1 = f1_score(y, y_class_pred)
print("F1 Score: \t{:0.3f}".format(f1))
```

```
Precision:      0.811
Recall:         0.860
F1 Score:       0.835
```

Scikit-Learn offers a handy `classification_report` function that combines all these:

```
In [94]: from sklearn.metrics import classification_report
```

```
In [95]: print(classification_report(y, y_class_pred))
```

	precision	recall	f1-score	support
0	0.85	0.80	0.82	50
1	0.81	0.86	0.83	50

avg / total	0.83	0.83	0.83	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

Multi-class Confusion Matrix

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:

```
In [96]: df = pd.read_csv('../data/weight-height.csv')
df.head()
```

Out [96] :

	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:

In [97] : `pd.get_dummies(df['Gender'], prefix='Gender').head()`

Out [97] :

	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):

In [98] : `df.head()`

Out [98] :

	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

```
In [99]: df['Height (feet)'] = df['Height']/12.0
          df['Weight (100 lbs)'] = df['Weight']/100.0
```

```
In [100]: df.describe().round(2)
```

Out [100] :

	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:

```
In [101]: 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)
```

Out [101] :

	Height	Weight	Height (feet)	Weight (100 lbs)	Weight_mms	Height_mms
count	10000.00	10000.00	10000.00	10000.00	10000.00	10000.00
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.

In [102] : `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)
```

Out[102] :

	Height	Weight	Height (feet)	Weight (100 lbs)	Weight_mms	Height_mms	Weight_ss	Height_ss
count	10000.00	10000.00	10000.00	10000.00	10000.00	10000.00	10000.00	10000.00
mean	66.37	161.44	5.53	1.61	0.47	0.49	0.00	0.00
std	3.85	32.11	0.32	0.32	0.16	0.16	1.00	1.00
min	54.26	64.70	4.52	0.65	0.00	0.00	-3.01	-3.15
25%	63.51	135.82	5.29	1.36	0.35	0.37	-0.80	-0.74
50%	66.32	161.21	5.53	1.61	0.47	0.49	-0.01	-0.01
75%	69.17	187.17	5.76	1.87	0.60	0.60	0.80	0.73
max	79.00	269.99	6.58	2.70	1.00	1.00	3.38	3.28

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:

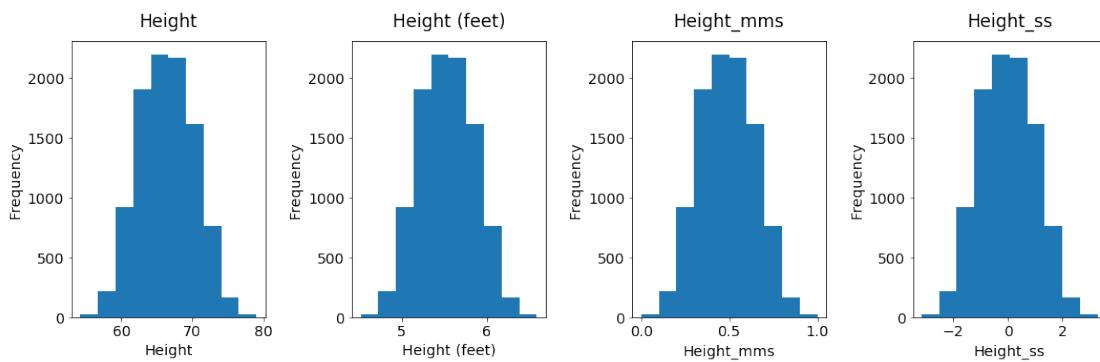
In [103] : `plt.figure(figsize=(15, 5))`

```

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)

plt.tight_layout();

```



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](#).

# 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**.

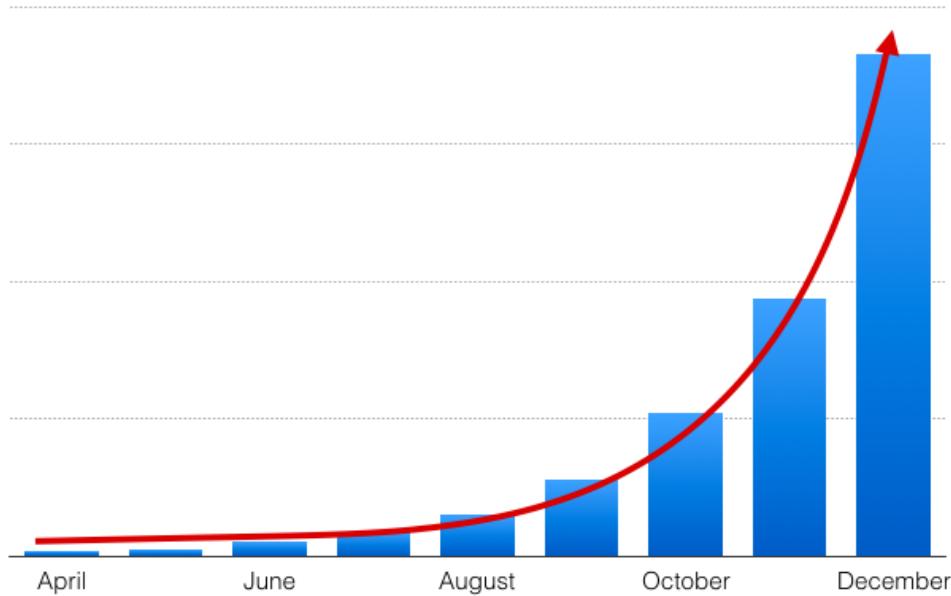
While these techniques are very powerful, they also have some limitations.

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.

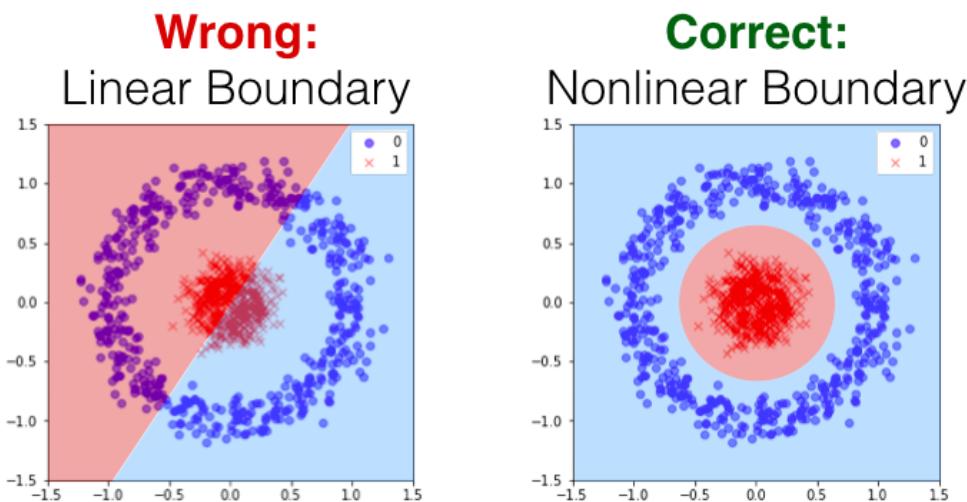
Similarly, logistic regression is incapable of separating classes that cannot be pulled apart by a flat boundary (a line in 2D, a plane in 3D, a hyperplane if we have more than 3 features). This happens all the time, and you may hear the term “not linearly separable” to describe two classes that cannot 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

Number of active users



Active Users Versus Time



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.

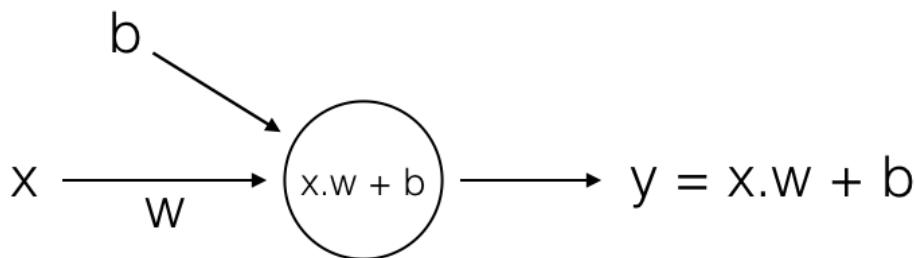
### Linear regression

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:

$$y = x \cdot w + b \quad (4.1)$$

and we can represent its operation as an artificial Neural Network like this:

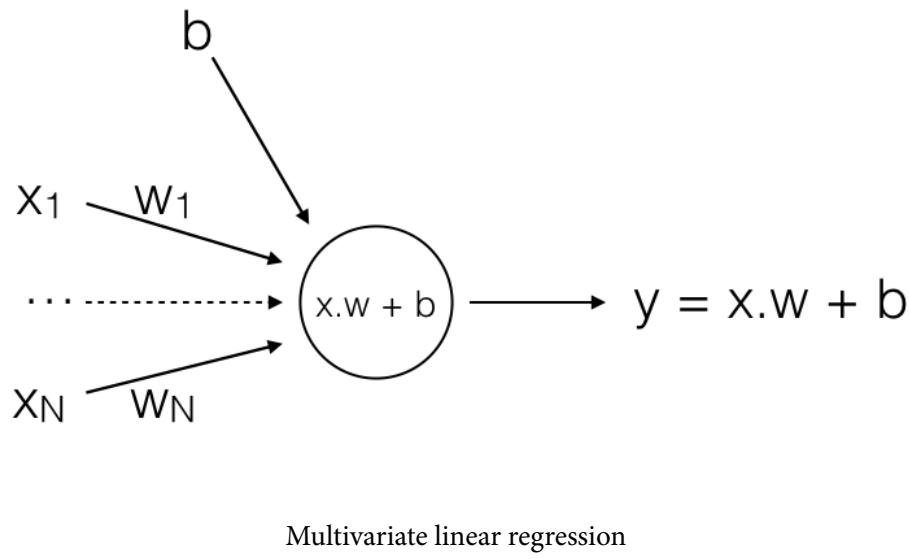


Linear regression as a neural net

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:

$$y = X \cdot w + b \quad (4.2)$$

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:

$$y = x_1 w_1 + \dots + x_N w_N + b = X \cdot w + b \quad (4.3)$$

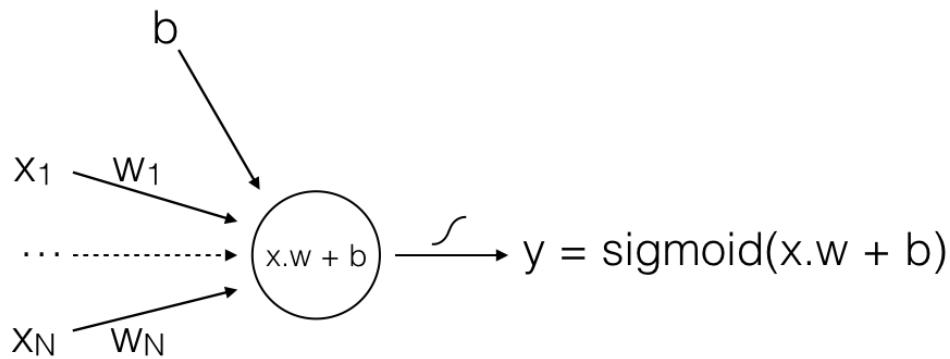
This is great! We can now visually represent linear regression with as many inputs as we like.

## 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.



Neural Network for Logistic Regression

TIP: if you need a refresher about the sigmoid you can check [Chapter 3](#) as well as this [nice article on Wikipedia](#).

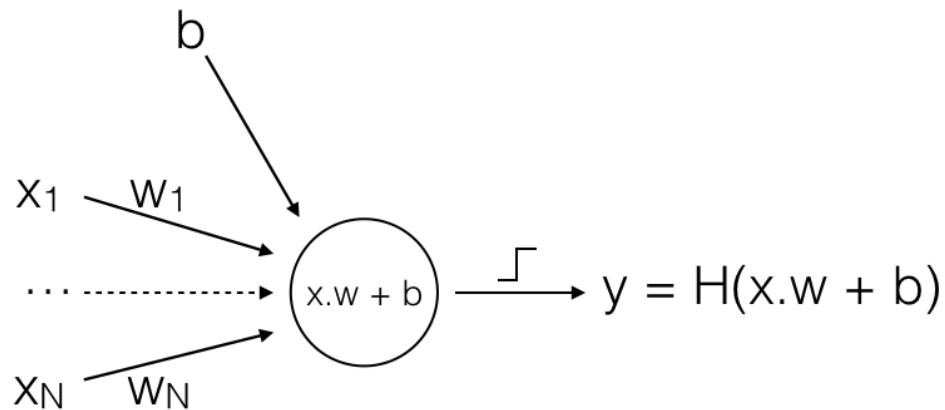
## Perceptron

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**.

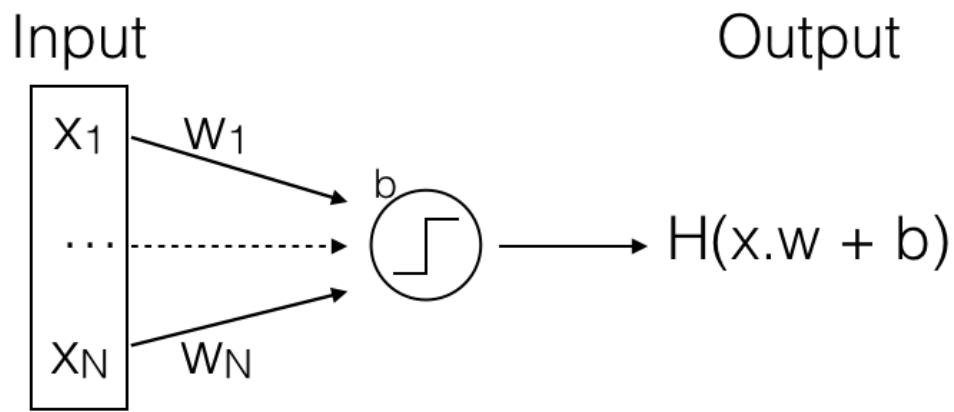
The Perceptron is also a binary classifier, but instead of using a smooth *sigmoid* activation function, it uses the *step function*:

$$y = \begin{cases} 1 & \text{if } w.x + b > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.4)$$

We could even simplify our diagram notation without losing information by including the bias and the activation symbols in the node itself, like this:



Perceptron



A more compact notation

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.

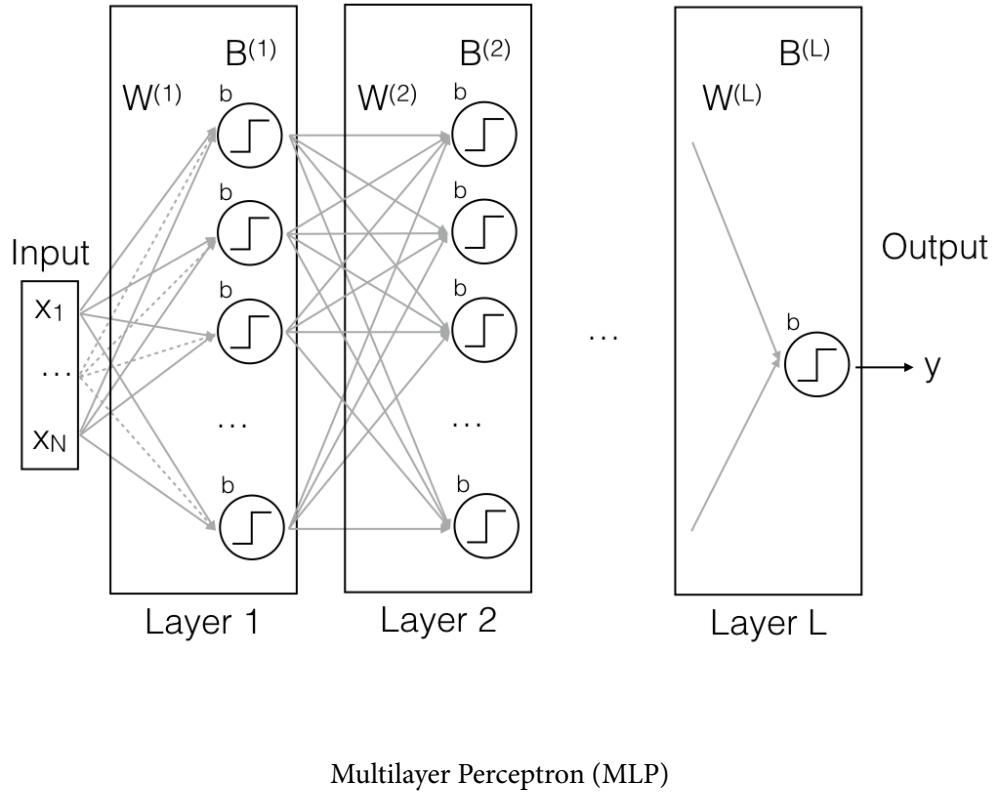
The weighted sum of the input plus the bias is sometimes also called a *linear combination* of the input features because it only involves sums and multiplications by parameters (no strange functions like exponentials, cosines etc.).

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:

$$z_k^{(1)} = x_1 w_{1k}^{(1)} + x_2 w_{2k}^{(1)} + \dots + b_k^{(1)} \text{ for every node } k \text{ in the first layer.} \quad (4.5)$$

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.

$$Z^{(1)} = X \cdot W^{(1)} + B^{(1)} = \sum_j x_j w_{jk}^{(1)} + b_k^{(1)} \quad (4.6)$$

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:

$$A^{(1)} = H(Z^{(1)}) \quad (4.7)$$

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:

$$A^{(2)} = H(A^{(1)} \cdot W^{(2)} + B^{(2)}) \quad (4.8)$$

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 nonlinear 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. First let's load our common files:

```
In [1]: with open('common.py') as fin:
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:
    exec(fin.read())
```

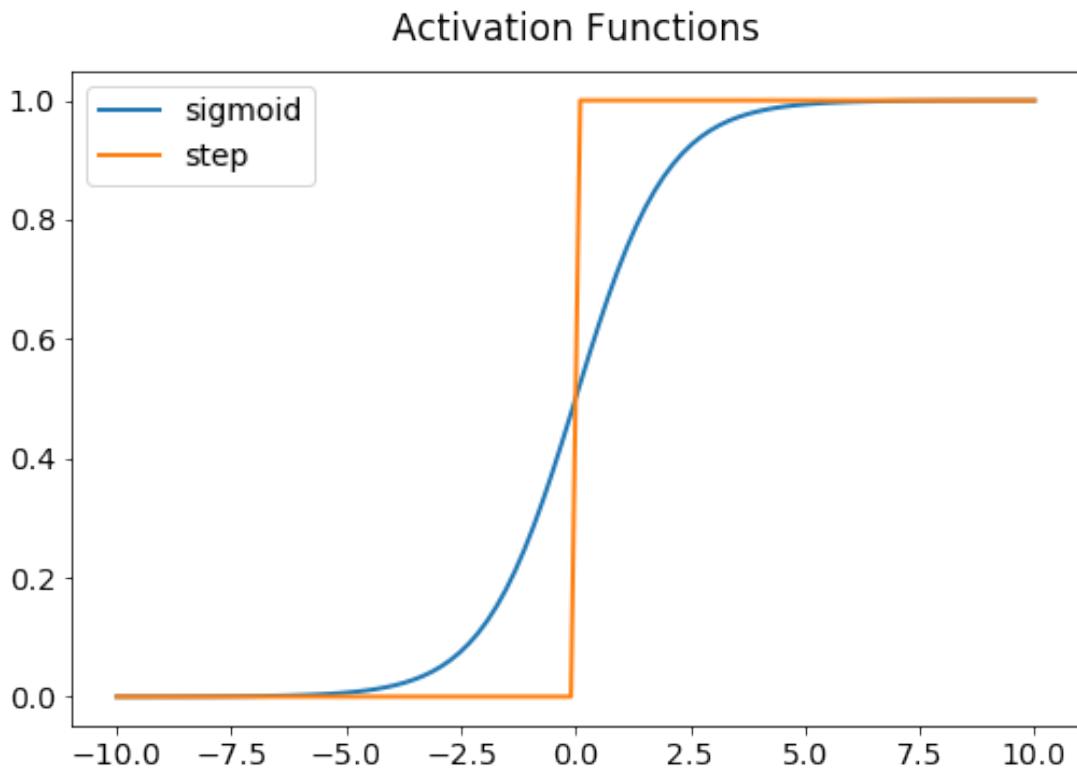
Sigmoid and Step functions are easy to define using numpy (using their mathematical formulas):

```
In [3]: 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:

```
In [4]: x = np.linspace(-10, 10, 100)
plt.plot(x, sigmoid(x))
plt.plot(x, step(x))
plt.legend(['sigmoid', 'step'])
plt.title('Activation Functions');
```



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

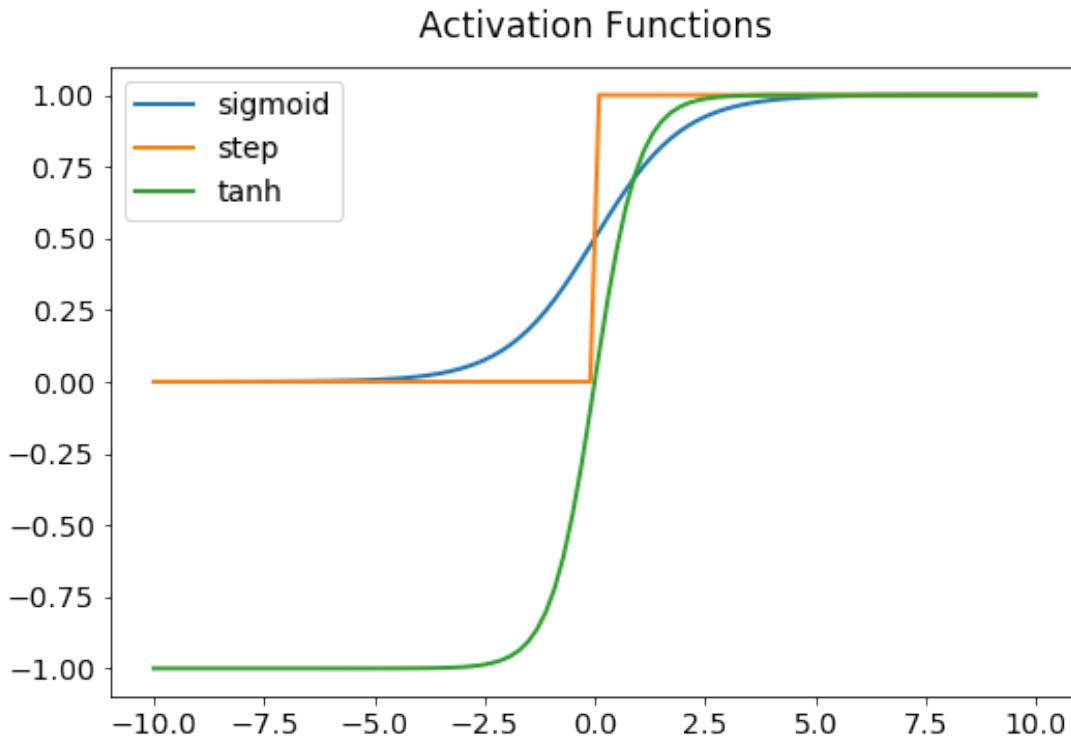
The [hyperbolic tangent](#) has a very similar shape to the sigmoid, but it is bounded and smoothly varying between  $[-1, +1]$  instead of  $[0, 1]$ , and is defined as:

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (4.9)$$

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:

```
In [5]: 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'])
plt.title('Activation Functions');
```



## ReLU

The [rectified linear unit](#) function or simply *rectifier* is defined as:

$$y = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.10)$$

or simply:

$$y = \max(0, x) \quad (4.11)$$

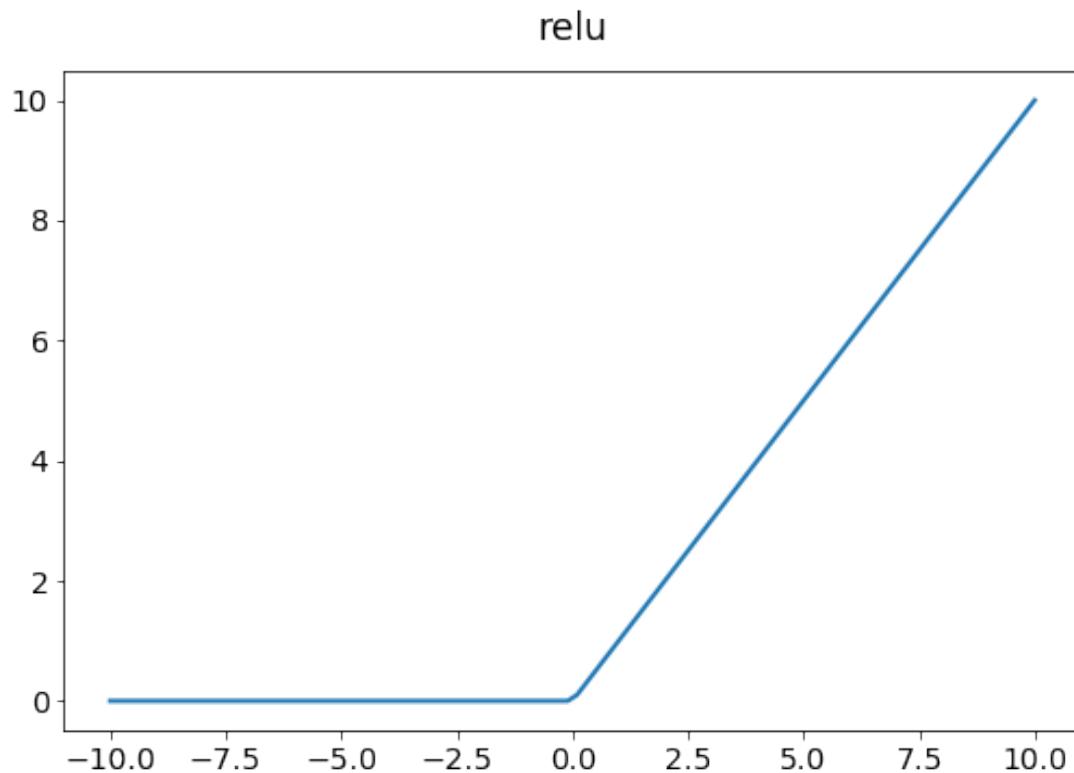
Originally motivated from biology, it has been shown to be very effective and it is probably the most

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.

```
In [6]: def relu(x):  
    cond = x > 0  
    return cond * x
```

```
In [7]: x = np.linspace(-10, 10, 100)  
plt.plot(x, relu(x))  
plt.title('relu');
```



## Softplus

The `Softplus` function is a smooth approximation of the ReLU:

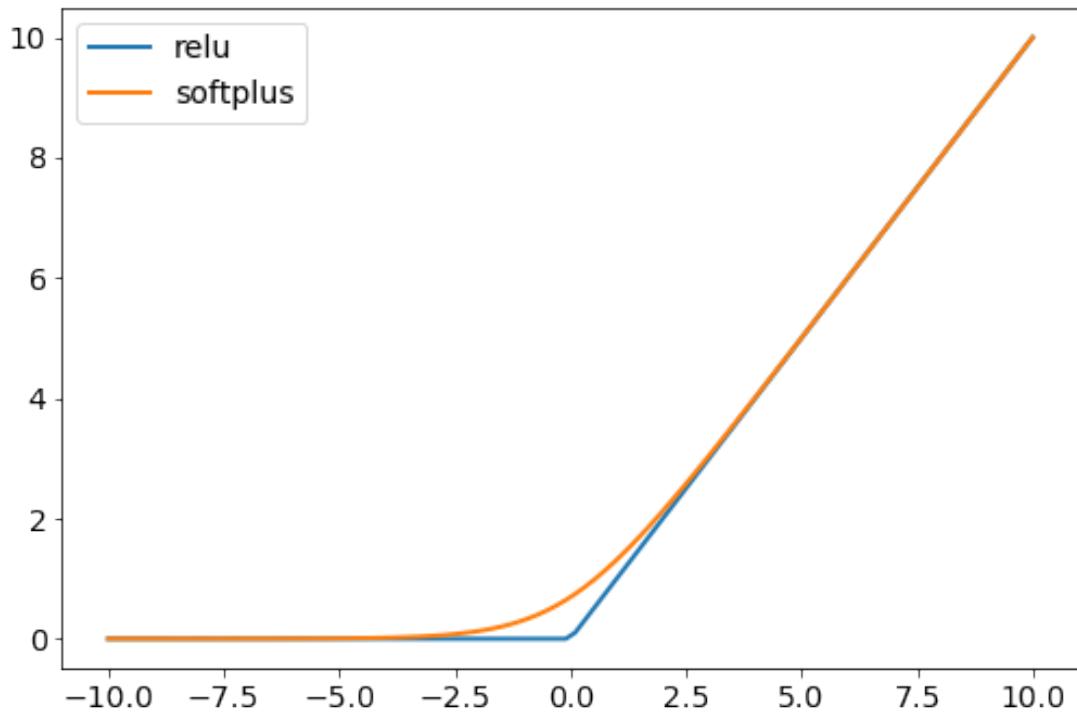
$$y = \log(1 + e^x) \quad (4.12)$$

We mention it for completeness, though it's rarely used in practice.

```
In [8]: def softplus(x):
    return np.log1p(np.exp(x))
```

```
In [9]: x = np.linspace(-10, 10, 100)
plt.plot(x, relu(x))
plt.plot(x, softplus(x))
plt.legend(['relu', 'softplus'])
plt.title('ReLU and Softplus');
```

ReLU and Softplus



## SeLU

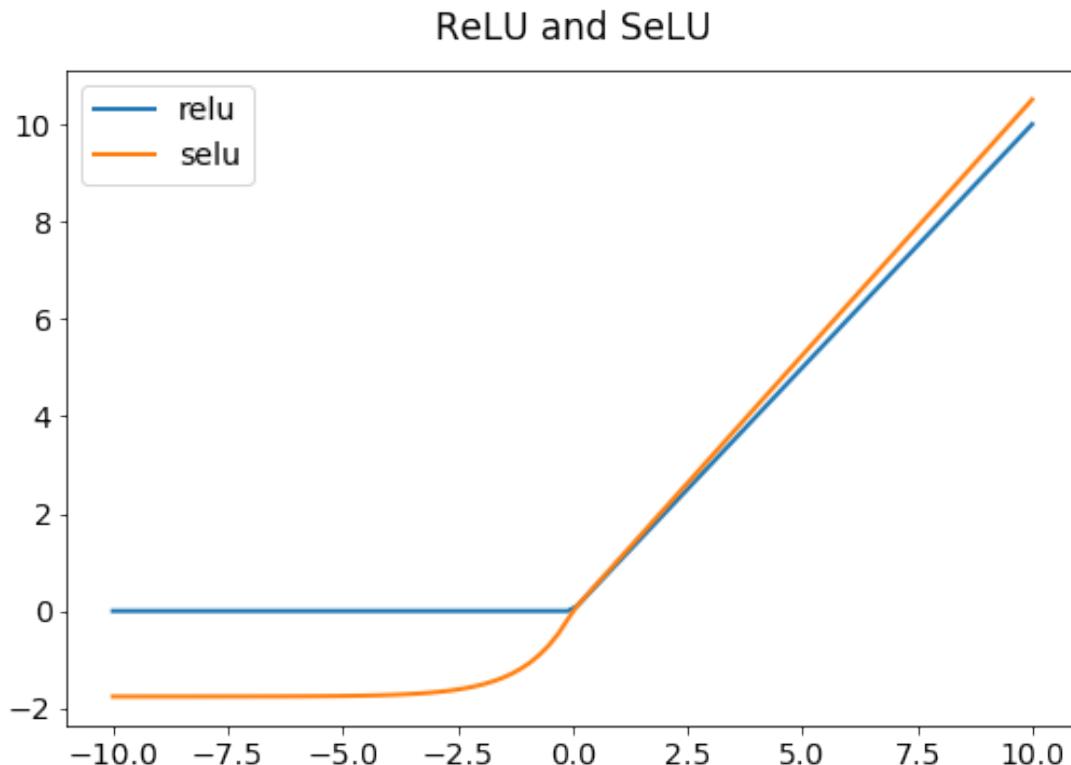
Finally, the SeLU activation function is a very recent development (see paper published in June 2017). The name stands for **scaled exponential linear unit** and it's implemented as:

$$y = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases} \quad (4.13)$$

On the positive axis it behaves like the rectified linear unit (ReLU), scaled by a factor  $\lambda$ . 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!

```
In [10]: def selu(x):
    alpha = 1.6732632423543772848170429916717
    scale = 1.0507009873554804934193349852946
    res = scale * np.where(x>0.0,
                           x,
                           alpha * (np.exp(x) - 1))
    return res
```

```
In [11]: x = np.linspace(-10, 10, 100)
plt.plot(x, relu(x))
plt.plot(x, selu(x))
plt.legend(['relu', 'selu'])
plt.title('ReLU and SeLU');
```



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. First we import the `make_moons` function from Scikit Learn:

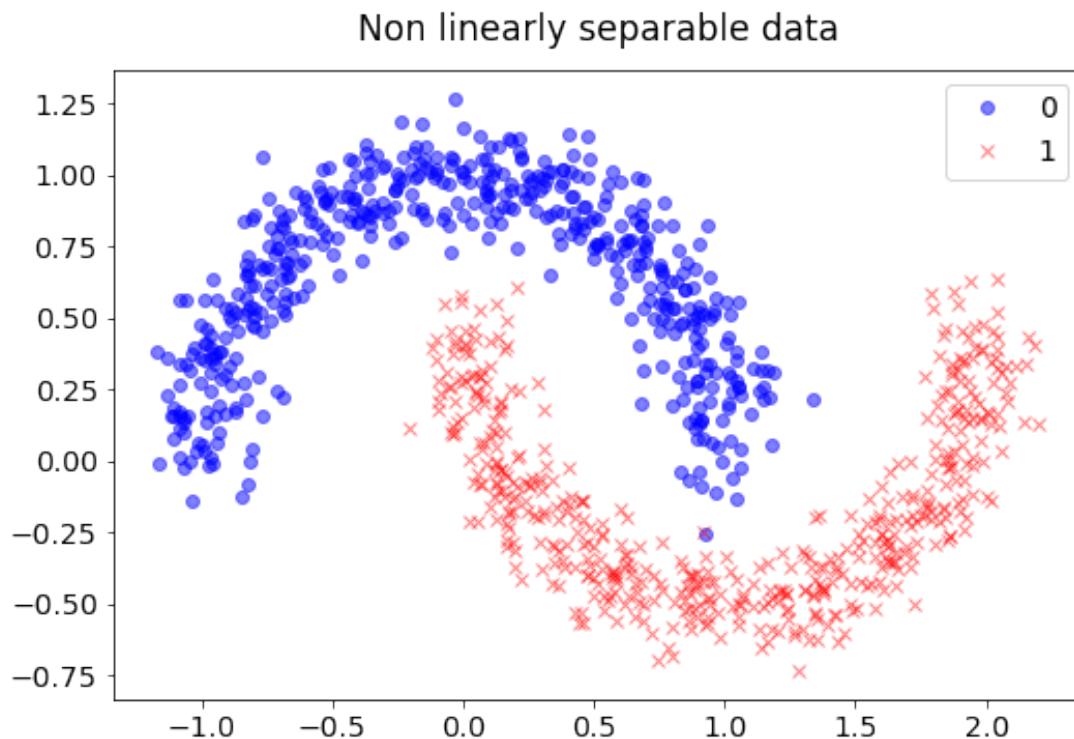
```
In [12]: from sklearn.datasets import make_moons
```

And then we use it to generate a synthetic dataset with 1000 points and 2 classes:

```
In [13]: X, y = make_moons(n_samples=1000,  
                           noise=0.1,  
                           random_state=0)
```

Let's plot this dataset and see what it looks like:

```
In [14]: 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'])  
  
plt.title('Non linearly separable data');
```



```
In [15]: X.shape
```

```
Out[15]: (1000, 2)
```

We split the data into training and test sets:

```
In [16]: from sklearn.model_selection import train_test_split
```

```
In [17]: 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:

```
In [18]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.optimizers import SGD, Adam
```

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.

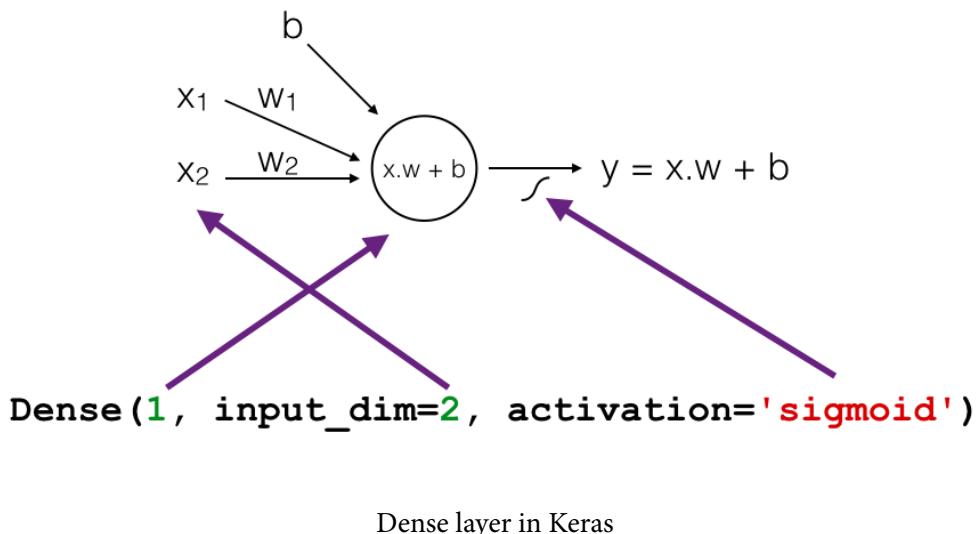
Let's start by building a [Logistic Regression model](#) like we did in the previous chapter. We will create it using the `Sequential` API, which is the simpler way to build models in Keras. We add a single `Dense` layer with 2 inputs, a single node and a `sigmoid` activation function:

```
In [19]: model = Sequential()
```

Now we'll add a single `Dense` layer with 2 inputs and we'll use the `sigmoid` activation function here.

```
In [20]: 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):

```
In [21]: 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 Multiclass 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.

```
In [22]: model.fit(X_train, y_train, epochs=200, verbose=0);
```

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:

```
In [23]: results = model.evaluate(X_test, y_test)
```

```
300/300 [=====] - 0s 116us/step
```

We can print out the accuracy by retrieving the second element in the `results` tuple:

```
In [24]: 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:

```
In [25]: 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)

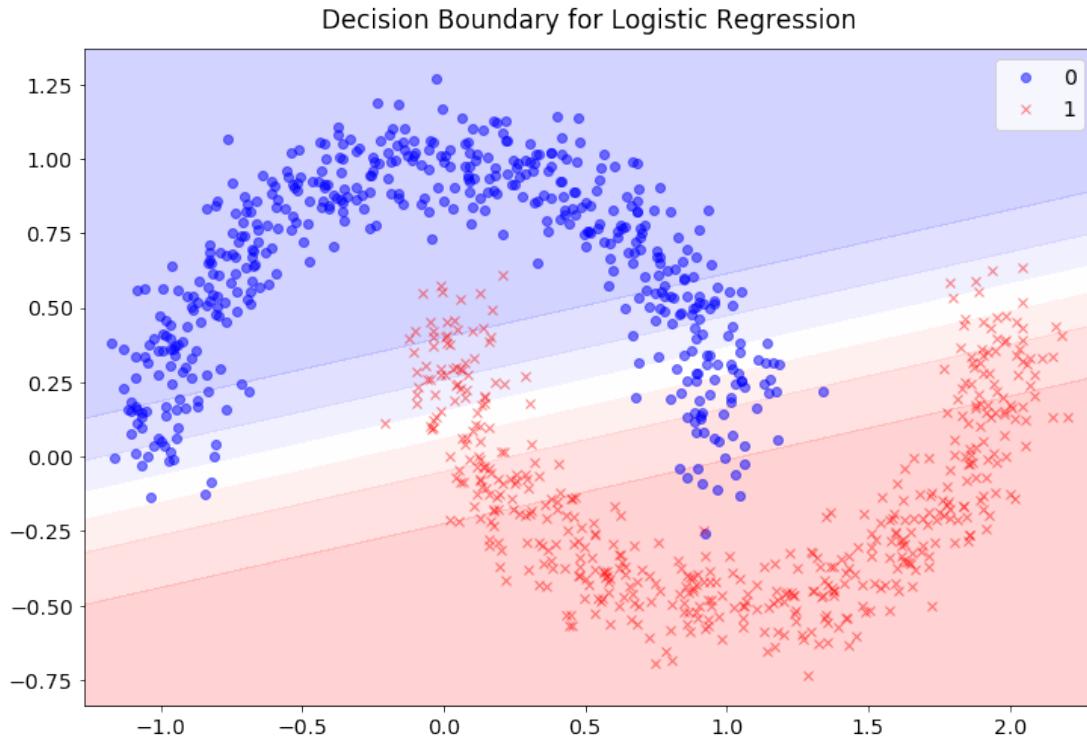
    aa, bb = np.meshgrid(hticks, vticks)
    ab = np.c_[aa.ravel(), bb.ravel()]

    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)

plt.title("Decision Boundary for Logistic Regression");
```



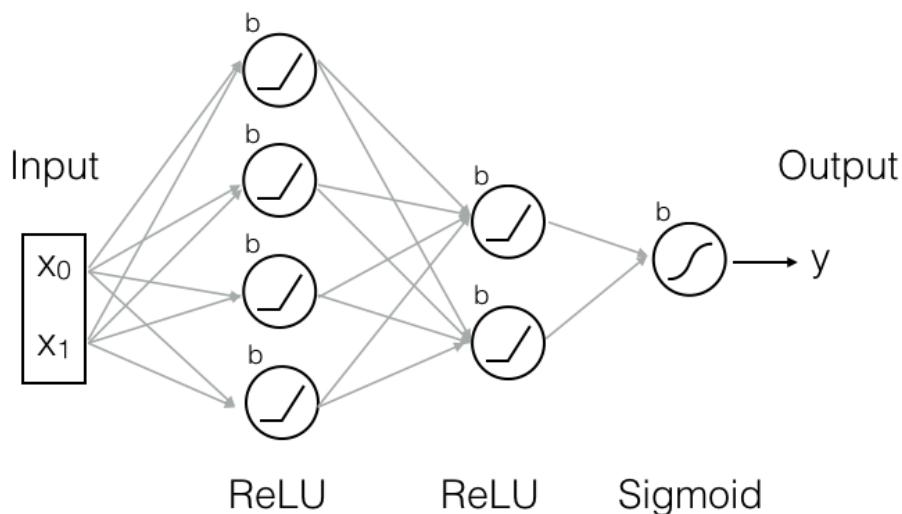
As you can see in the figure, since a shallow model like logistic regression is not able to draw curved

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:



Graph of our network

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:

```
In [26]: 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 ( $2 \times 4$ ) plus 1 bias for each of the nodes (4).

Let's now add a second layer after the first one, with 2 nodes:

```
In [27]: 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:

```
In [28]: model.add(Dense(1, activation='sigmoid'))
```

and let's compile the model:

```
In [29]: 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:

```
In [30]: model.fit(X_train, y_train, epochs=100, verbose=0);
```

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:

```
In [31]: 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.

```
In [32]: y_train_prob = model.predict(X_train)
          y_test_prob = model.predict(X_test)
```

Let's look at the first few values for comparison:

```
In [33]: y_train_pred[:3]
```

```
Out[33]: array([[1],
                 [1],
                 [0]], dtype=int32)
```

```
In [34]: y_train_prob[:3]
```

```
Out[34]: array([0.9999733 ,
                 0.999522 ,
                 0.00104994], 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`:

```
In [35]: from sklearn.metrics import accuracy_score
          from sklearn.metrics import confusion_matrix
```

Let's check out the score accuracy here for both the training set and the test set:

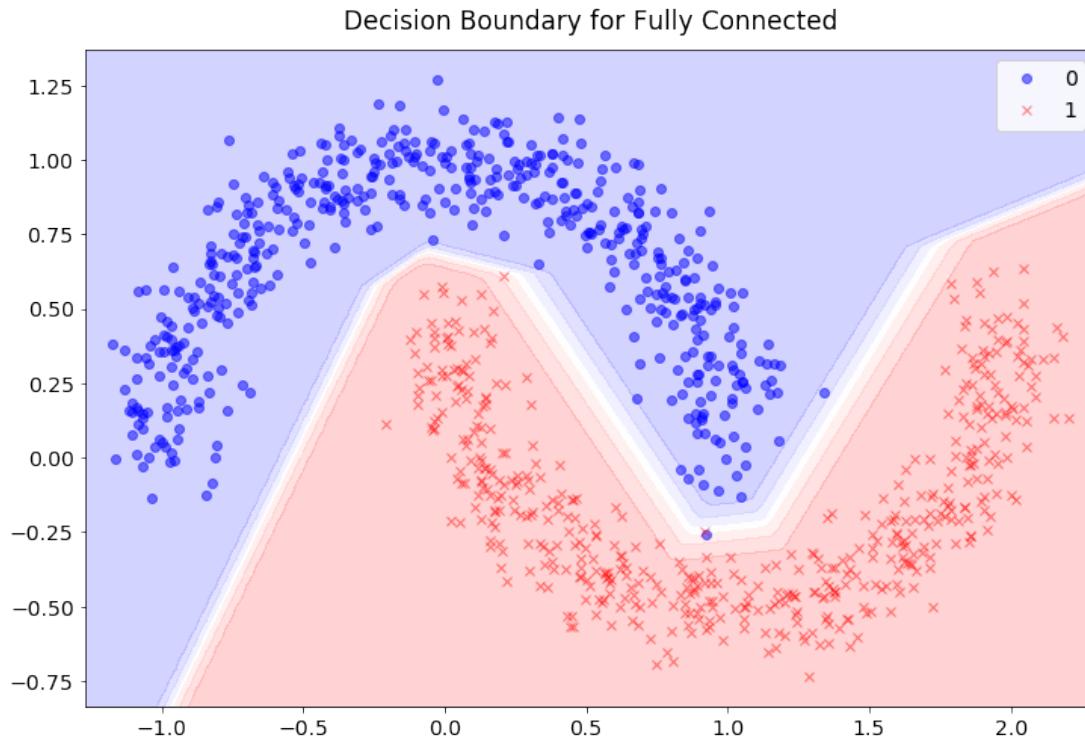
```
In [36]: acc = accuracy_score(y_train, y_train_pred)
          print("Accuracy (Train set):\t{:0.3f}".format(acc))

          acc = accuracy_score(y_test, y_test_pred)
          print("Accuracy (Test set):\t{:0.3f}".format(acc))
```

```
Accuracy (Train set): 0.999
Accuracy (Test set): 1.000
```

Let's plot the decision boundary for the model:

```
In [37]: plot_decision_boundary(model, X, y)
          plt.title("Decision Boundary for Fully Connected");
```



As you can see, our network learned to separate the two classes with a zig-zag boundary, which is typical of the ReLU activation.

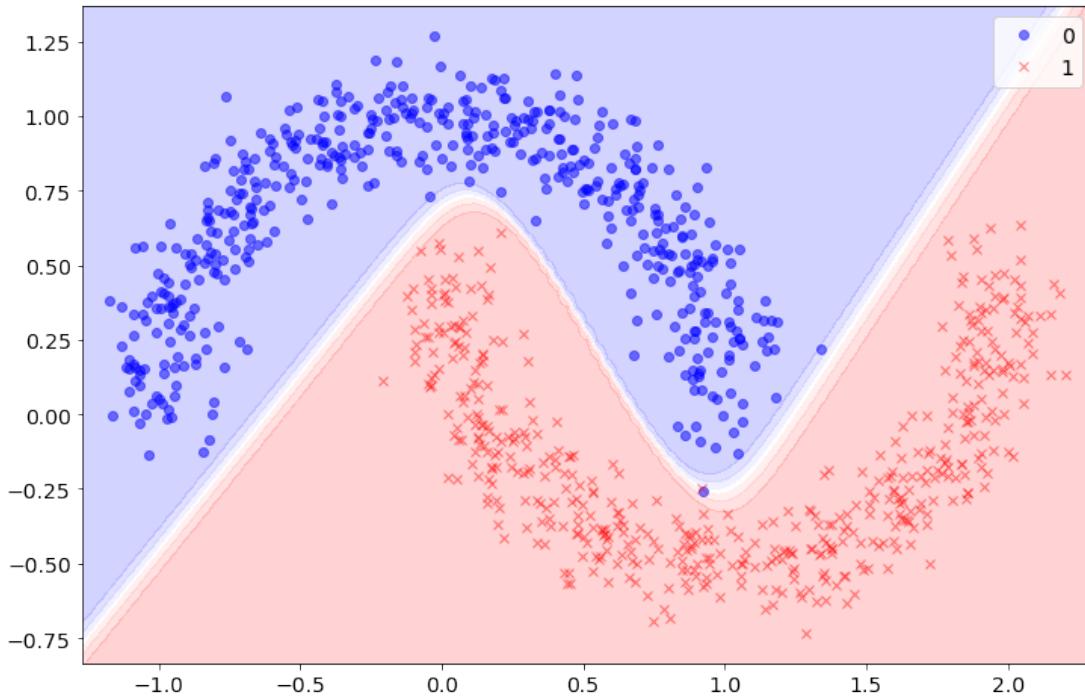
TIP: if your the model has not learned to separate the data well, just re-initialize the model and re-train it. As you'll see later in this book, the model is initialized randomly and this may have a huge effect on it's ability to effectively learn.

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:

```
In [38]: 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)
```



```
In [39]: y_train_pred = model.predict_classes(X_train)
y_test_pred = model.predict_classes(X_test)
```

```
In [40]: acc = accuracy_score(y_train, y_train_pred)
print("Accuracy (Train set):\t{:0.3f}".format(acc))

acc = accuracy_score(y_test, y_test_pred)
print("Accuracy (Test set):\t{:0.3f}".format(acc))
```

```
Accuracy (Train set): 0.999
Accuracy (Test set): 1.000
```

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:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \text{ for } j = 1, \dots, K. \quad (4.14)$$

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.

```
In [41]: df = pd.read_csv('../data/iris.csv')
```

```
In [42]: df.head()
```

**Out [42] :**

	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:

```
In [43]: X = df.drop('species', axis=1)
X.head()
```

**Out [43] :**

	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:

```
In [44]: targets = df['species'].unique()
targets
```

```
Out[44]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

And then build a dictionary where we assign an index to each target name in alphabetical order:

```
In [45]: target_dict = {n:i for i, n in enumerate(targets)}
target_dict
```

```
Out[45]: {'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`:

```
In [46]: y= df['species'].map(target_dict)
y.head()
```

```
Out[46]:
```

	species
0	0
1	0
2	0
3	0
4	0

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:

```
In [47]: from keras.utils import to_categorical
```

```
In [48]: y_cat = to_categorical(y)
```

Let's check out what the data looks like by looking at the first 5 values:

```
In [49]: y_cat[:5]
```

```
Out[49]: array([[1., 0., 0.],
   [1., 0., 0.],
   [1., 0., 0.],
   [1., 0., 0.],
   [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.

```
In [50]: 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.

```
In [51]: model = Sequential()
model.add(Dense(3, input_dim=4, activation='softmax'))
model.compile(Adam(lr=0.1),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

```
In [52]: model.fit(X_train, y_train,
                  validation_split=0.1,
                  epochs=30, verbose=0);
```

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:

```
In [53]: y_pred = model.predict(X_test)
y_pred
```

```
Out[53]: array([[9.8024070e-01, 1.9749358e-02, 9.9601284e-06],
   [1.6103700e-02, 5.9058928e-01, 3.9330697e-01],
   [9.3280071e-01, 6.7088291e-02, 1.1102484e-04],
   [1.1895786e-03, 2.5968346e-01, 7.3912692e-01],
   [9.6339279e-01, 3.6573283e-02, 3.3838107e-05],
   [1.7135940e-02, 5.9908730e-01, 3.8377672e-01],
   [8.2506472e-04, 1.8012747e-01, 8.1904745e-01],
   [9.4950175e-01, 5.0426044e-02, 7.2208233e-05],
   [9.3665427e-01, 6.3233301e-02, 1.1248481e-04],
   [7.7709131e-02, 7.6407224e-01, 1.5821865e-01],
   [8.4380146e-05, 5.8892172e-02, 9.4102347e-01],
   [3.9271075e-02, 7.0223731e-01, 2.5849167e-01],
   [1.4533973e-02, 5.6776279e-01, 4.1770327e-01],
   [6.3411851e-04, 1.6341269e-01, 8.3595318e-01],
   [6.8567425e-02, 7.7941293e-01, 1.5201971e-01],
   [4.1240861e-04, 1.2742312e-01, 8.7216449e-01],
   [2.0971078e-04, 8.6805493e-02, 9.1298473e-01],
   [4.6339136e-02, 7.4134523e-01, 2.1231568e-01],
   [3.8887326e-02, 7.4175382e-01, 2.1935885e-01],
   [9.7016585e-01, 2.9813653e-02, 2.0478823e-05],
   [9.6865195e-01, 3.1322919e-02, 2.5115298e-05],
   [4.1269182e-04, 1.1924139e-01, 8.8034588e-01],
   [4.7990444e-04, 1.5782441e-01, 8.4169573e-01],
   [2.1963108e-03, 3.4429601e-01, 6.5350765e-01],
   [9.5849311e-01, 4.1458704e-02, 4.8205155e-05],
   [2.0923292e-02, 6.4100039e-01, 3.3807626e-01],
   [3.3716213e-02, 6.7946661e-01, 2.8681713e-01],
   [1.5672502e-05, 2.6626769e-02, 9.7335762e-01],
   [9.7098070e-01, 2.8997714e-02, 2.1596850e-05],
   [9.5597339e-01, 4.3978024e-02, 4.8622391e-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:

```
In [54]: 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()` is found in `sklearn.metrics` package:

```
In [55]: from sklearn.metrics import classification_report
```

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).

```
In [56]: print(classification_report(y_test_class, y_pred_class))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	10
1	1.00	1.00	1.00	10
2	1.00	1.00	1.00	10
avg / total	1.00	1.00	1.00	30

We get the confusion matrix by running the `confusion_matrix()` method passing it the same arguments as the classification report:

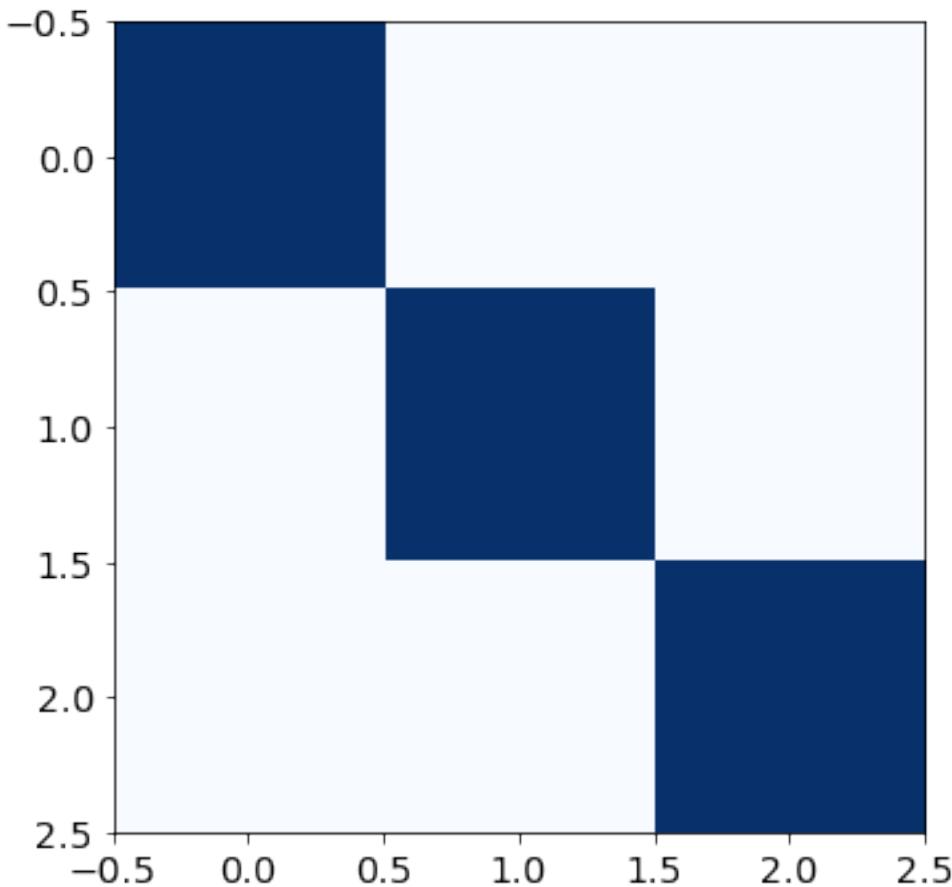
```
In [57]: cm = confusion_matrix(y_test_class, y_pred_class)

pd.DataFrame(cm, index = targets,
             columns = ['pred_'+c for c in targets])
```

Out[57] :

	pred_setosa	pred_versicolor	pred_virginica
setosa	10	0	0
versicolor	0	10	0
virginica	0	0	10

```
In [58]: plt.imshow(cm, cmap='Blues');
```



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:

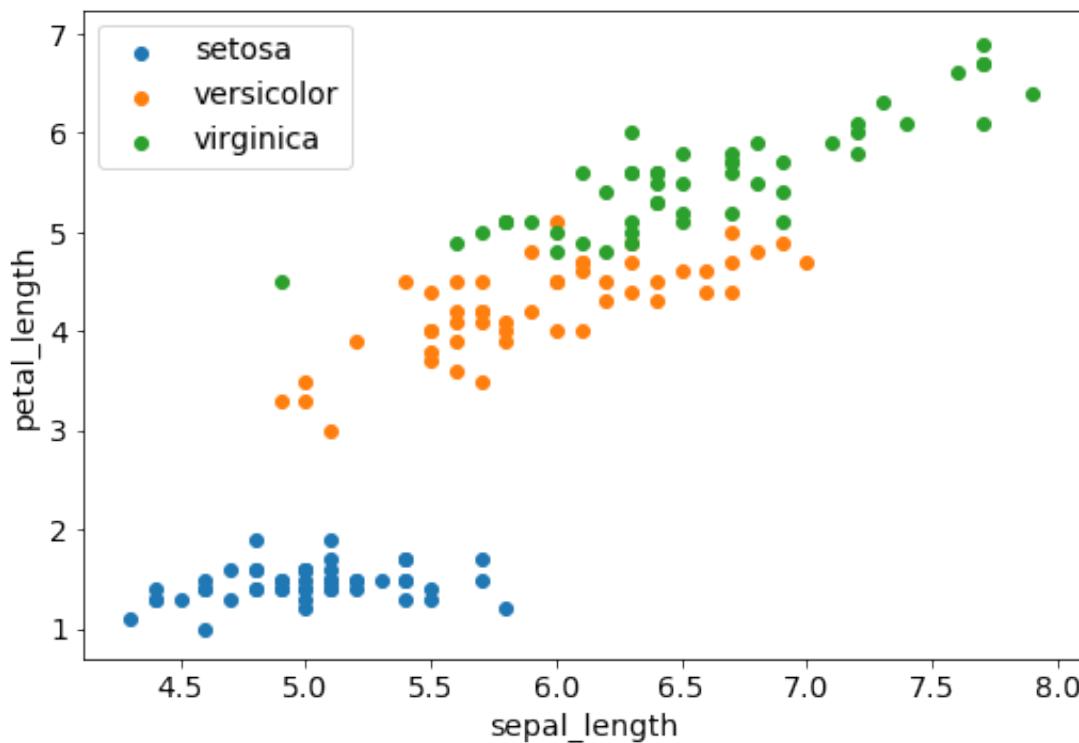
```
In [59]: plt.scatter(X.loc[y==0, 'sepal_length'],
                   X.loc[y==0, 'petal_length'])

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(targets)
plt.title("The Iris Dataset");
```

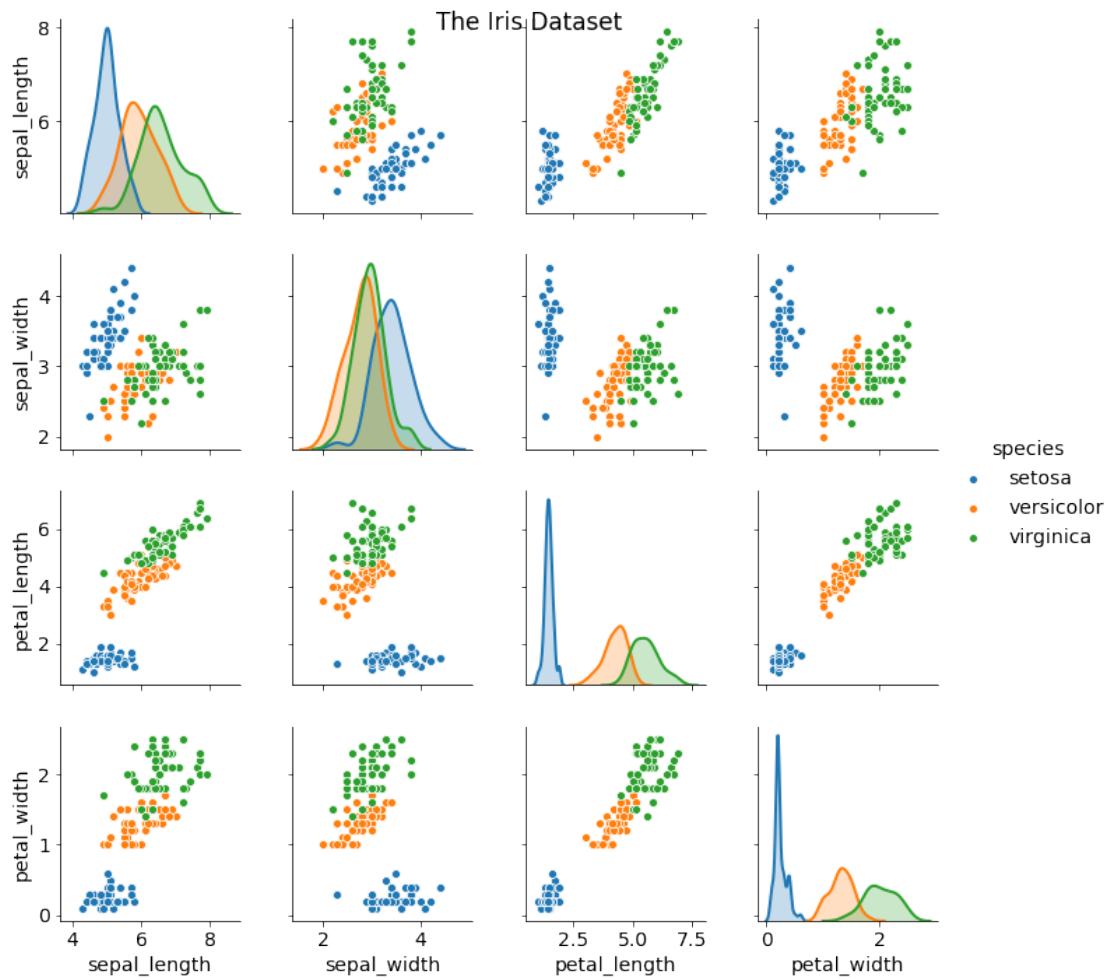
## The Iris Dataset



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:

```
In [60]: import seaborn as sns
```

```
In [61]: g = sns.pairplot(df, hue="species")
g.fig.suptitle("The Iris Dataset");
```



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)<sup>2</sup>)
- 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.

# 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:

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:  
    exec(fin.read())
```

## Derivatives

As the name suggests a **derivative** is a function that derives from another function.

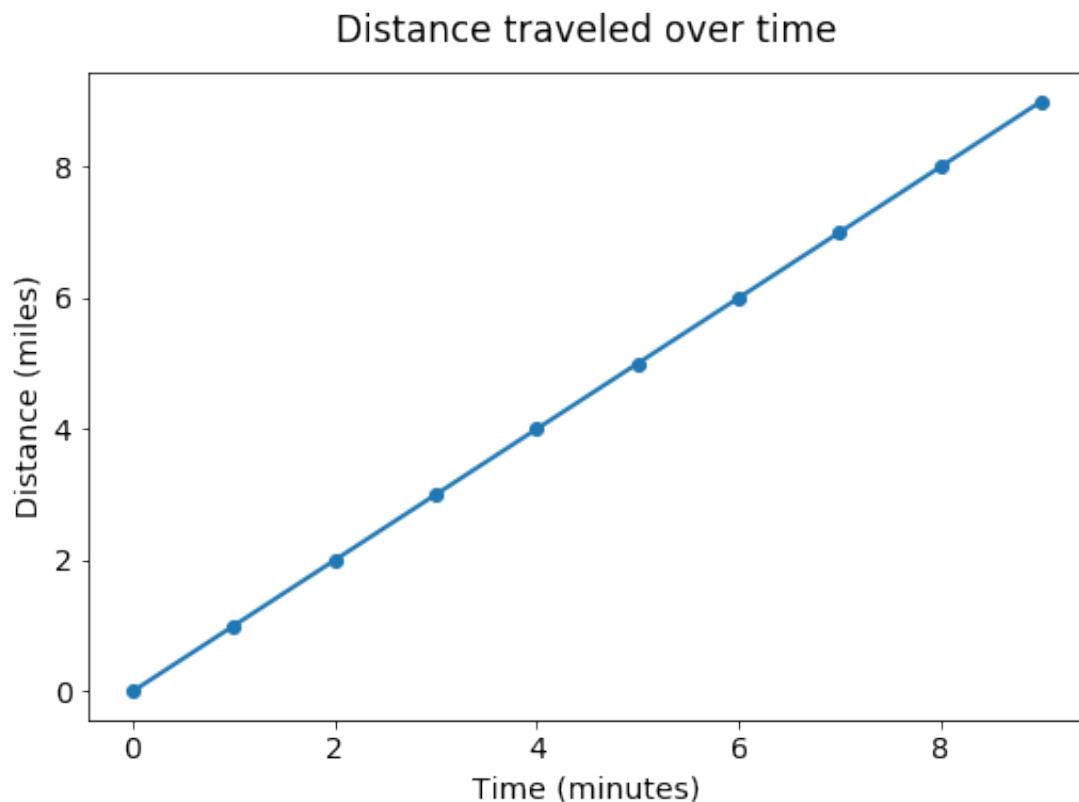
Let's start with an example. Imagine you are driving on the highway. As time goes by you mark your position along the highway, filling a table of values as a function of time. If your speed is 60 miles an hour, every minute your position will be increased by 1 mile.

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  $x$ :

```
In [3]: 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.

```
In [4]: plt.plot(t, x, 'o-')
plt.title("Distance traveled over time")
plt.ylabel("Distance (miles)")
plt.xlabel("Time (minutes)");
```



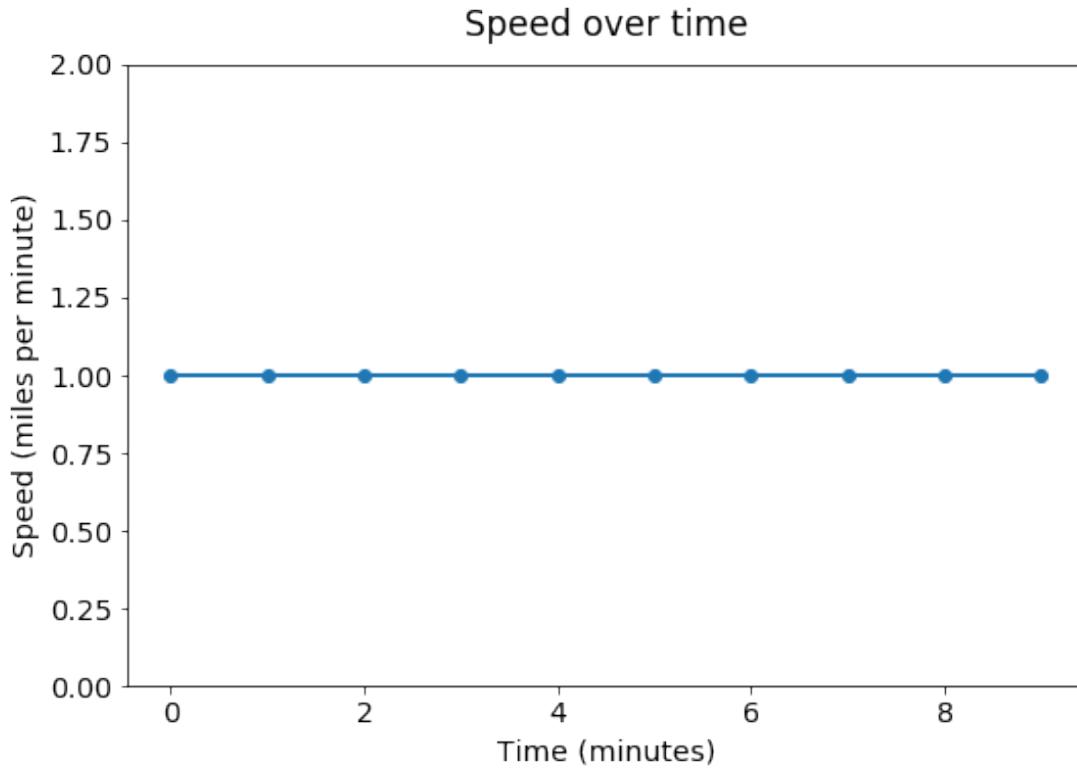
The derivative  $x'(t)$  of this function is the **rate of change** in position with respect to time. In this example it is the speed of your car indicated by the odometer. In the example just mentioned, the derivative is a constant value of 60 miles per hour, or 1 mile per minute. Let's create an array containing the speed at each moment in time:

```
In [5]: v = np.ones(10) # 1 mile per minute or 60 miles per hour
```

and let's plot it too:

```
In [6]: plt.plot(t, v, 'o-')
plt.ylim(0, 2)
```

```
plt.title("Speed over time")
plt.ylabel("Speed (miles per minute)")
plt.xlabel("Time (minutes)");
```



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:

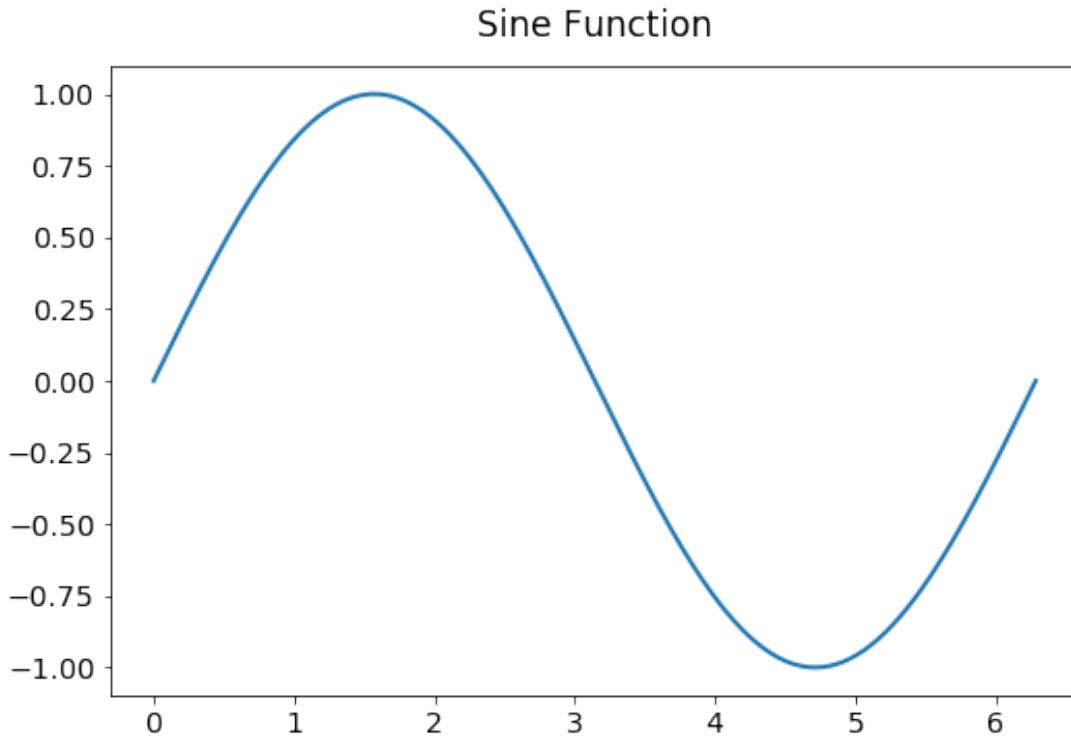
$$x'(t) := \frac{dx}{dt}(t) \quad (5.1)$$

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:

In [7]: `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:

```
In [8]: f = np.sin(t)
plt.plot(t, f)
plt.title("Sine Function");
```



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](#):

$$\frac{df}{dt}(t_i) \approx \frac{\Delta f}{\Delta t}(t_i) = \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \quad (5.2)$$

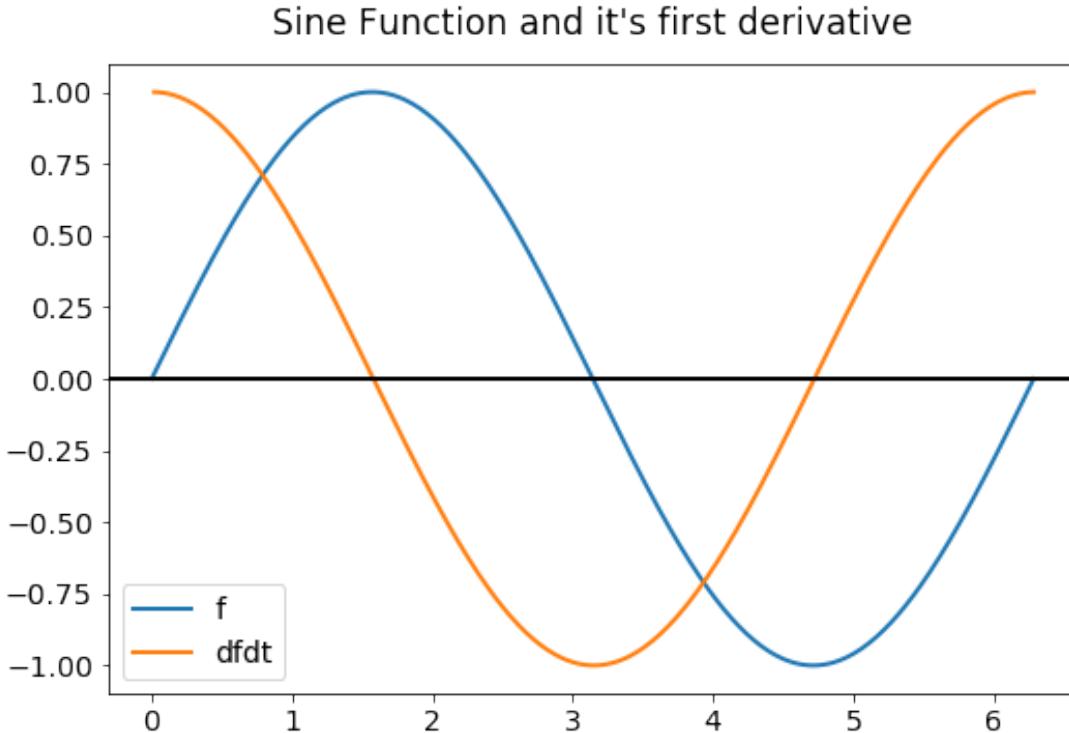
where we indicated with  $\Delta f$  the difference between two consecutive values of  $f$ .

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:

```
In [9]: dfdt = np.diff(f)/np.diff(t)
```

Let's plot it together with the original function:

```
In [10]: plt.plot(t, f)
    plt.plot(t[1:], dfdt)
    plt.legend(['f', 'dfdt'])
    plt.axhline(0, color='black')
    plt.title("Sine Function and it's first derivative");
```



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:

```
In [11]: 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)

```

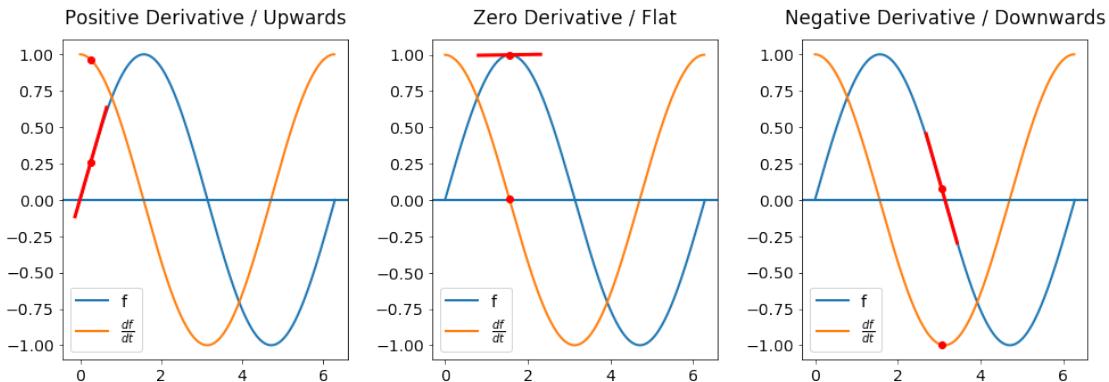
We can use this helper function to display the relationship between the inclination (slope) of our tangent line and the value of the derivative function. As you can see, a positive derivative corresponds to an uphill tangent while negative derivative corresponds to a downhill tangent line.

```

In [12]: 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.tight_layout();

```



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	$\log(x)$	$\frac{1}{x}$

Common functions and their derivatives

## 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

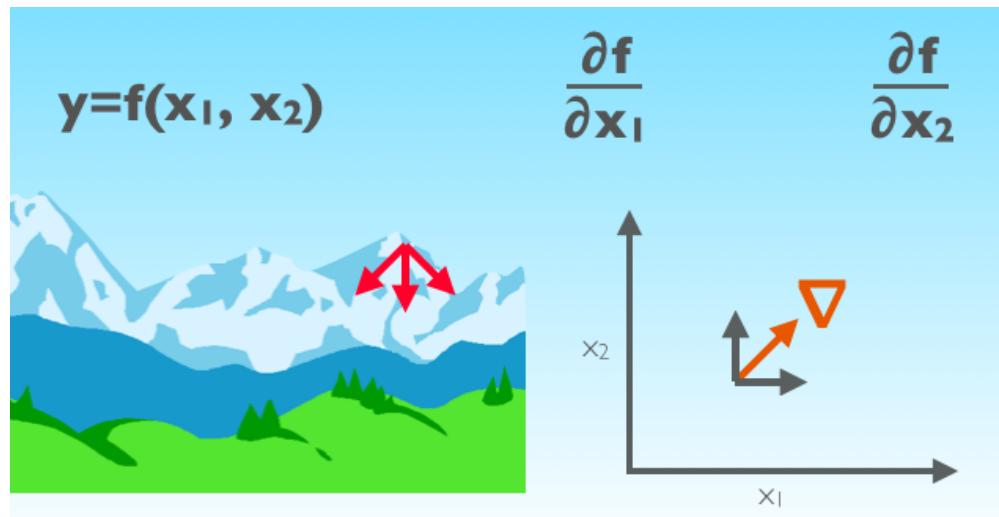
east-west directions, it will be in whatever direction the hill is more steeply descending down.

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*:  $\nabla$ .

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!



Visualization of gradient in a landscape

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.

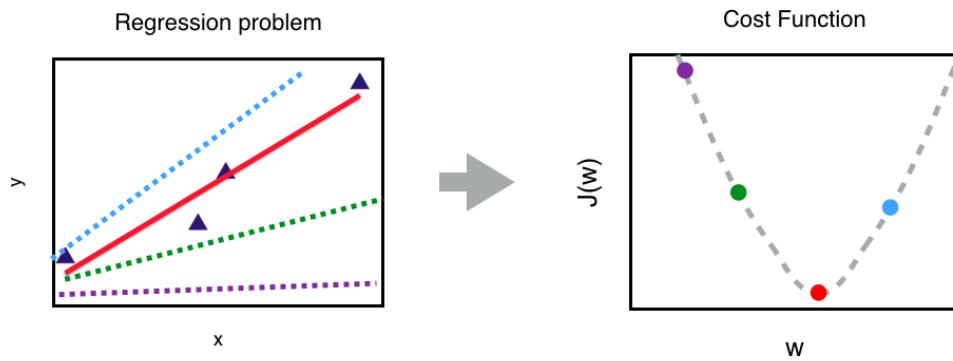
At a high-level, the backpropagation algorithm is a Supervised Learning method for training our networks.

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  $w$



Linear regression and its loss

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.

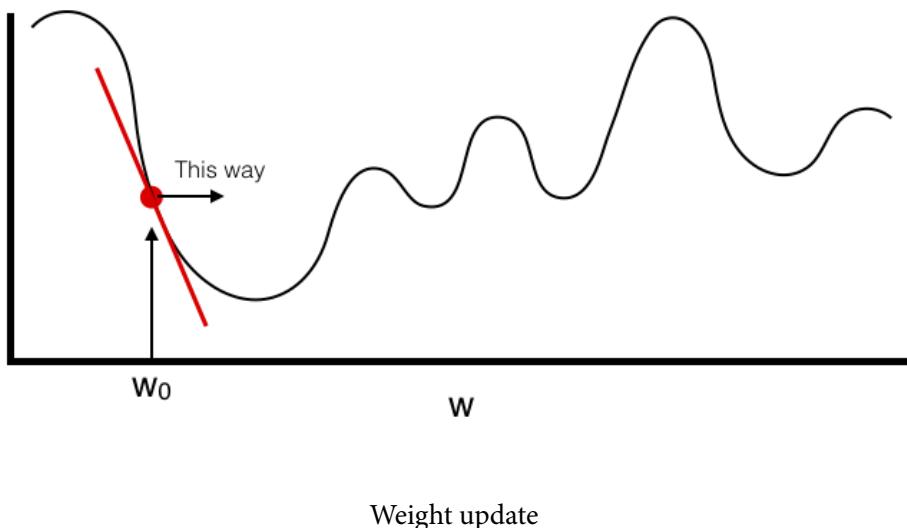
What is clear is that the shape of such a cost function is not going to be a smooth parabola like the one in the figure, and we will need a way to navigate a very complex landscape in search for a minimum value.

Let's say we are sitting at a particular point  $w_o$  with corresponding cost  $J(w_o)$ , 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  $w_o$  and then change our position by subtracting the value of the derivative from the value of our starting position  $w_o$ .

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:

$$w_o \rightarrow w_o - \frac{dJ}{dw}(w_o) \quad (5.3)$$

Let's check that this does move us towards lower values on the vertical axis.

If we are at  $w_o$  like in the figure, the slope of the curve is negative and thus the quantity  $-\frac{dJ}{dw}(w_o)$  is positive. So, the value of  $w_o$  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  $w_o$  where the value of the slope is positive, we would subtract a positive quantity  $\frac{dJ}{dw}(w_o)$  that is now negative. This would move  $w_o$  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  $\eta$ :

$$w_o \leftarrow w_o - \eta \frac{dJ}{dw}(w_o)$$

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  $[-4, 4]$ :

```
In [13]: x = np.linspace(-4, 4, 100)
```

Then let's define an invented cost function  $J(w)$  that depends on  $w$  in some weird way.

$$J(w) = 70.0 - 15.0w^2 + 0.5w^3 + w^4 \quad (5.4)$$

```
In [14]: 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.

$$\frac{dJ}{dw}(w) = -30.0w + 1.5w^2 + 4w^3 \quad (5.5)$$

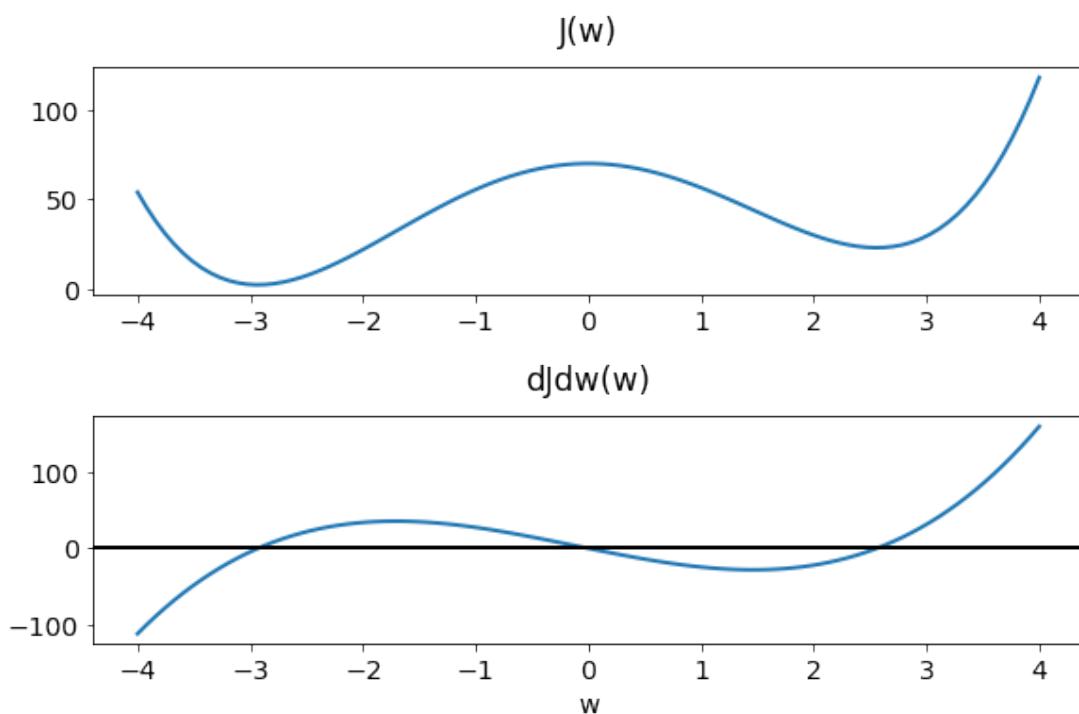
```
In [15]: def dJdw(w):
    return - 30.0*w + 1.5*w**2 + 4*w**3
```

Let's plot both functions:

```
In [16]: 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();
```



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  $w_0 = -4$ :

In [17]: `w0 = -4`

and let's apply the update rule:

$$w_o \leftarrow w_o - \eta \frac{dJ}{dw}(w_o) \quad (5.6)$$

We will choose a small learning rate of  $\eta = 0.001$  initially:

In [18]: `lr = 0.001`

The update step is:

In [19]: `step = lr * dJdw(w0)`  
`step`

Out[19]: `-0.112`

and the new value of  $w_0$  is:

In [20]: `w0 - step`

Out[20]: `-3.888`

i.e. we moved to the right, towards the minimum!

Let's do 30 iterations and see where we get:

In [21]: `iterations = 30`

`w = w0`

```

ws = [w]

for i in range(iterations):
    step = lr * dJdw(w)
    w -= step
    ws.append(w)

ws = np.array(ws)

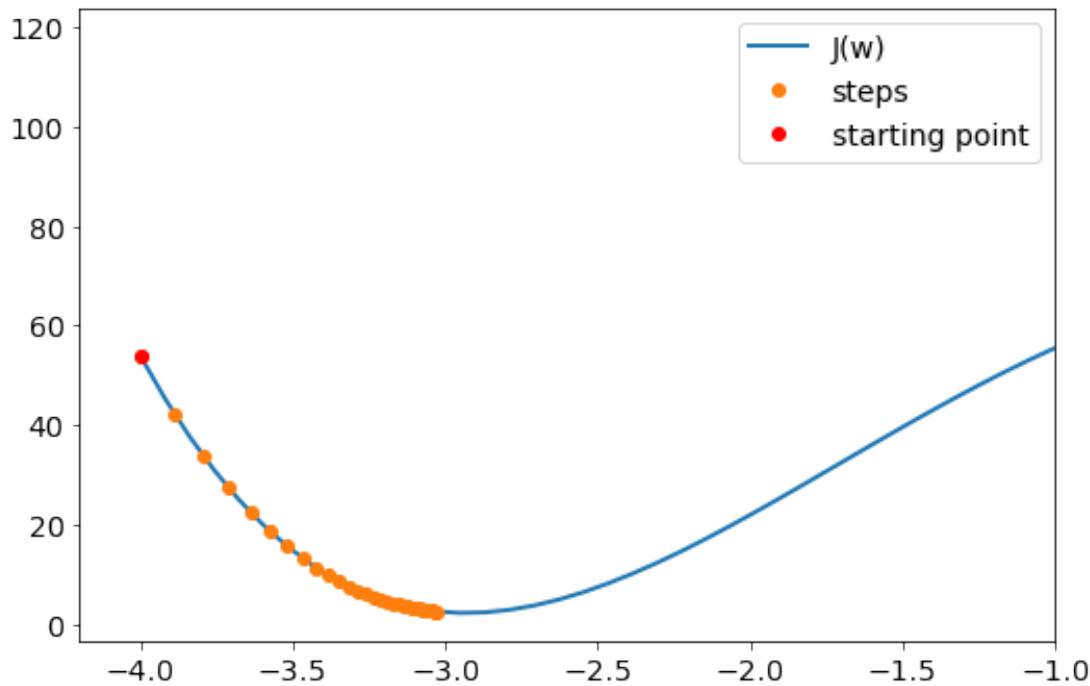
```

Let's visualize our descent, zooming in the interesting region of the curve:

```

In [22]: 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);

```



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  $b$ .

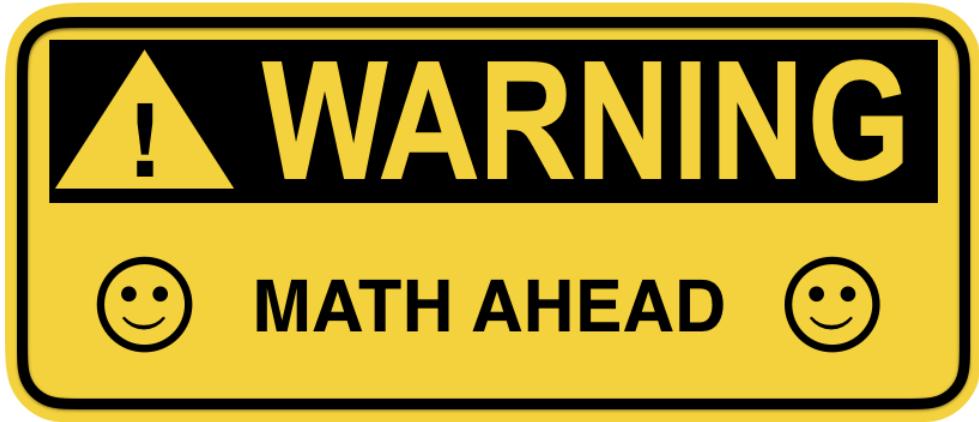
Also, remember that we defined a cost  $J(\hat{y}, y) = J(f(X, w, b), y)$  that is calculated over the training set. So, for fixed training data, the cost  $J$  is a function of the parameters  $w$  and  $b$ .

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:  $w_o -> w_o - \eta \frac{\partial J}{\partial w}$ .

All we need to do at this point is to learn how to calculate the gradient  $\frac{\partial J}{\partial w}$ .

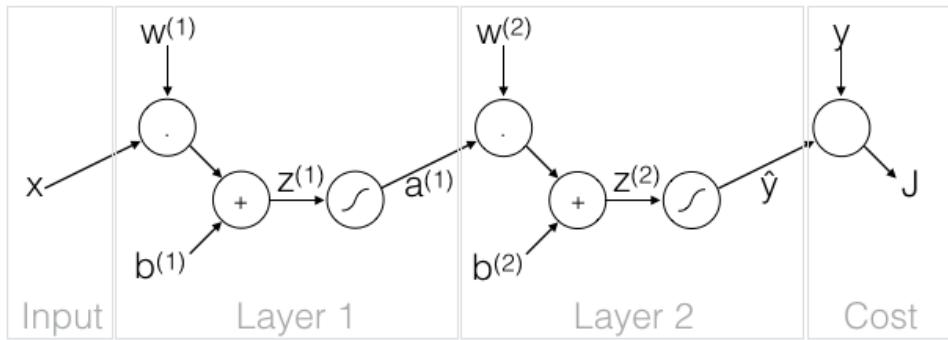
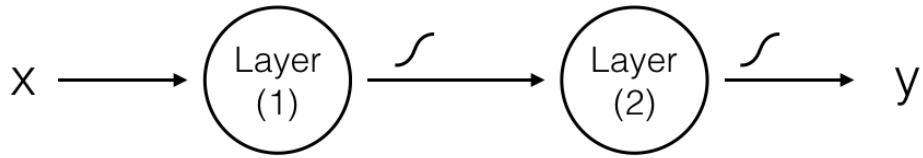
## The math of backpropagation



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:

$$z^{(1)} = x w^{(1)} + b^{(1)} \quad (5.7)$$

$$a^{(1)} = \sigma(z^{(1)}) \quad (5.8)$$

$$z^{(2)} = a^{(1)} w^{(2)} + b^{(2)} \quad (5.9)$$

$$\hat{y} = a^{(2)} = \sigma(z^{(2)}) \quad (5.10)$$

$$J = J(\hat{y}, y) \quad (5.11)$$

$$(5.12)$$

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  $\sigma$  (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.

$$\frac{\partial J}{\partial w^{(2)}} \quad (5.13)$$

$w^{(2)}$  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*.

$$\frac{\partial J}{\partial w^{(2)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}} \quad (5.14)$$

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.:

$$\overline{w^{(2)}} := \frac{\partial J}{\partial w^{(2)}} \quad (5.15)$$

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:

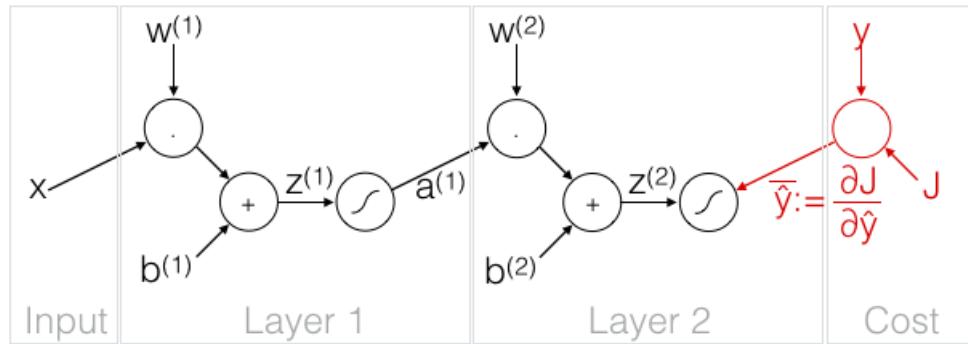
$$\overline{w^{(2)}} = \overline{z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}} = \bar{y} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w^{(2)}} \quad (5.16)$$

And we can start to see why it is called backpropagation: in order to get to calculate  $\overline{w^{(2)}}$  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  $\overline{w^{(2)}}$ .

**Step 1:**  $\bar{y} = \frac{\partial J}{\partial \hat{y}}$

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  $\bar{y}$  which is only the labels and the predictions:



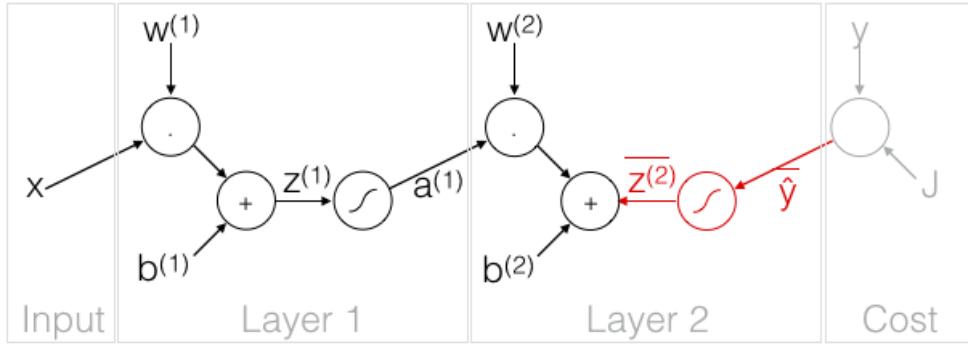
**Step 2:**  $\overline{z^{(2)}} = \frac{\partial J}{\partial z^{(2)}}$

As noted before, the chain rule tells us that  $\overline{z^{(2)}}$  is the product of the derivative of the sigmoid with the term we just calculated  $\bar{y}$ :

$$\overline{z^{(2)}} = \bar{y} \frac{\partial \hat{y}}{\partial z^{(2)}} = \bar{y} \sigma'(z^{(2)}) \quad (5.17)$$

Notice how information is propagating backwards in the graph:

Since we have already calculated  $\bar{y}$  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  $\sigma'$ .

$$\text{Step 3: } \overline{w^{(2)}} = \frac{\partial J}{\partial w^{(2)}}$$

Now we can calculate  $\overline{w^{(2)}}$ .

Looking at the formulas above, we know that:

$$\overline{w^{(2)}} = \overline{z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(2)}} \quad (5.18)$$

Since we have already calculated  $\overline{z^{(2)}}$  we only need to calculate  $\frac{\partial z^{(2)}}{\partial w^{(2)}}$ , which is equal to  $\overline{z^{(2)}} a^{(1)}$

So we have:

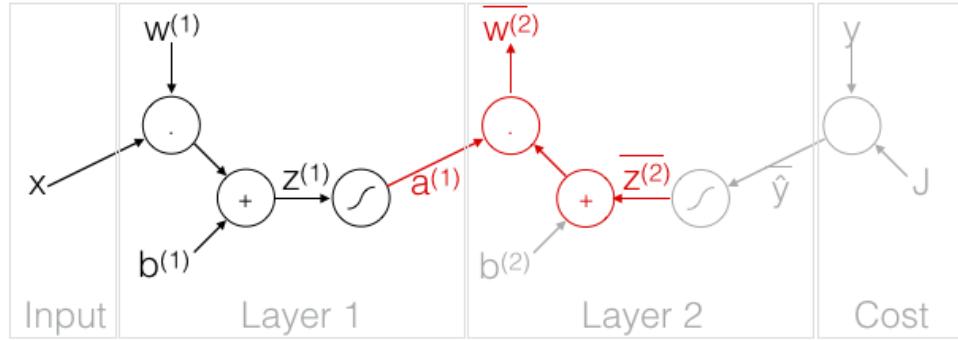
$$\overline{w^{(2)}} = \overline{z^{(2)}} \overline{z^{(2)}} a^{(1)} \quad (5.19)$$

This last formula is really interesting because it tells us that the update to the weights  $\overline{w^{(2)}}$  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.  $\overline{z^{(2)}}$ , is a constant, representing the downstream contribution of the network to the error.

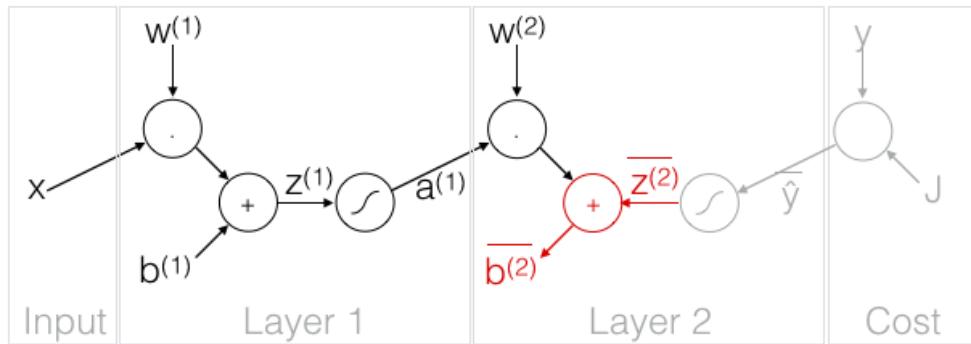
Using the same procedure we can calculate the corrections to the bias  $\overline{b^{(2)}}$  as well:

$$\text{Step 4: } \overline{b^{(2)}} = \frac{\partial J}{\partial b^{(2)}}$$

We can apply the chain rule again and obtain:

$$\overline{b^{(2)}} = \overline{z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}} = \overline{z^{(2)}}$$
 (5.20)

Since the  $\frac{\partial z^{(2)}}{\partial b^{(2)}} = 1$

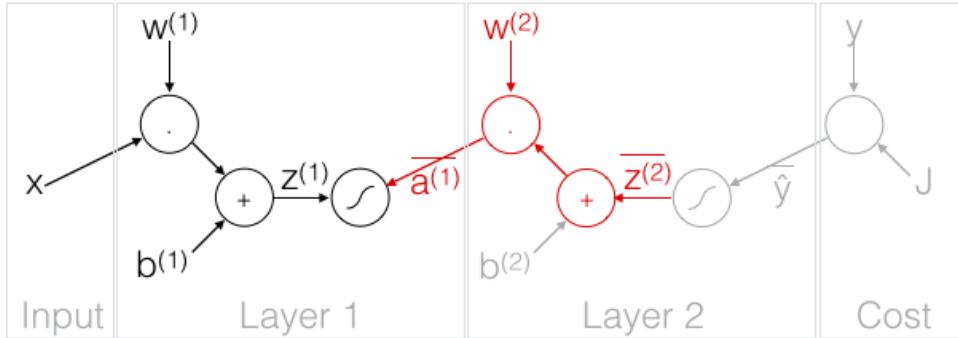


Following a similar procedure we can keep propagating the error back and calculate the corrections to  $w^{(1)}$  and  $b^{(1)}$ . Proceeding backwards, the next term we need to calculate is  $\overline{a^{(1)}}$ .

**Step 5:**  $\overline{a^{(1)}} = \frac{\partial J}{\partial a^{(1)}}$

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:

$$\overline{a^{(1)}} = \overline{z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} = \overline{z^{(2)}} w^{(2)} \quad (5.21)$$



At this point the calculation of the other terms is mechanical, and we will just summarize them all here:

$$\overline{\hat{y}} = \frac{\partial J}{\partial \hat{y}} \quad (5.22)$$

$$\overline{z^{(2)}} = \overline{\hat{y}} \sigma'(z^{(2)}) \quad (5.23)$$

$$\overline{b^{(2)}} = \overline{z^{(2)}} \quad (5.24)$$

$$\overline{w^{(2)}} = \overline{z^{(2)}} a^{(1)} \quad (5.25)$$

$$\overline{a^{(1)}} = \overline{z^{(2)}} w^{(2)} \quad (5.26)$$

$$\overline{z^{(1)}} = \overline{a^{(1)}} \sigma'(z^{(1)}) \quad (5.27)$$

$$\overline{b^{(1)}} = \overline{z^{(1)}} \quad (5.28)$$

$$\overline{w^{(1)}} = \overline{z^{(1)}} x \quad (5.29)$$

$$(5.30)$$

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.

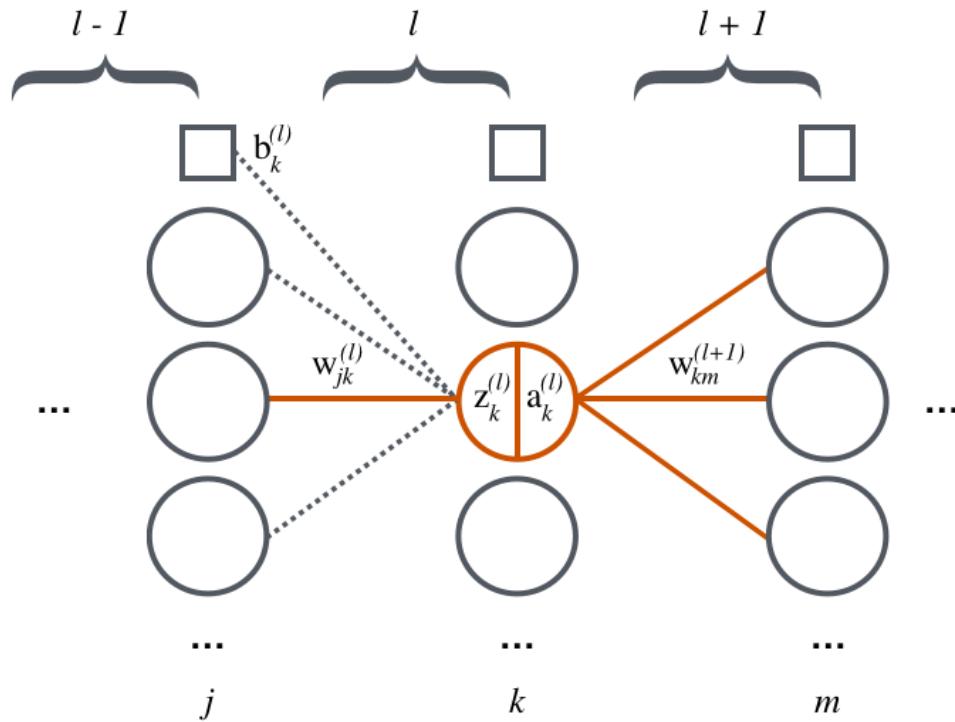
## Fully Connected Backpropagation

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  $k$ ,  $z_k^{(l)}$  is the weighted sum of the activations of layer  $l - 1$  plus the bias term of layer  $l$ :

$$z_k^{(l)} = \sum_j a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)}$$



## Forward Pass

The forward pass equations can be written as follows:

$$\dots \quad (5.31)$$

$$z_k^{(l)} = \sum_j a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)} \quad (5.32)$$

$$a_k^{(l)} = \sigma(z_k^{(l)}) \quad (5.33)$$

$$\dots \quad (5.34)$$

$$(5.35)$$

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:

$$z_s^{(L)} = \sum_r a_r^{(L-1)} w_{rs}^{(L)} + b_s^{(L)} \quad (5.36)$$

$$\hat{y}_s = \sigma(z_s^{(L)}) \quad (5.37)$$

$$J = \sum_s J(\hat{y}_s, y_s) \quad (5.38)$$

$$(5.39)$$

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:

```
In [23]: 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:

```
In [24]: b = np.array([0., 0.])
```

The output of the layer with 4 elements is represented by the array  $a$ , whose elements are  $a_j^{(l-1)}$ :

```
In [25]: a = np.array([0.5, -0.2, 0.3, 0.])
```

Then, the layer  $l$  performs the operation:

```
In [26]: z = np.dot(a, W) + b
```

returning the array of  $z$  with elements  $z_k^{(l)} = \sum_j a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)}$ :

```
In [27]: z
```

```
Out[27]: 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:

$$\overline{\hat{y}_s} = \frac{\partial J}{\partial \hat{y}_s} \quad (5.40)$$

$$\overline{z_s^{(L)}} = \overline{\hat{y}_s} \sigma'(z_s^{(L)}) \quad (5.41)$$

$$\overline{b_s^{(L)}} = \overline{z_s^{(L)}} \quad (5.42)$$

$$\overline{w_{rs}^{(L)}} = \overline{z_s^{(L)}} a_r^{(L-1)} \quad (5.43)$$

$$\dots \quad (5.44)$$

$$\dots \quad (5.45)$$

$$\overline{a_k^{(l)}} = \sum_m \overline{w_{km}^{(l+1)}} \overline{z_m^{(l+1)}} \quad (5.46)$$

$$\overline{z_k^{(l)}} = \overline{a_k^{(l)}} \sigma'(z_k^{(l)}) \quad (5.47)$$

$$\overline{b_k^{(l)}} = \overline{z_k^{(l)}} \quad (5.48)$$

$$\overline{w_{jk}^{(l)}} = \overline{z_k^{(l)}} a_j^{(l-1)} \quad (5.49)$$

$$(5.50)$$

These equations are equivalent to the ones for the unidimensional case, with only **one major difference**.

The term  $\overline{a_k^{(l)}}$ , 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:

$$\dots \quad (5.51)$$

$$\mathbf{z}^{(l)} = \mathbf{a}^{(l-1)} \mathbf{W}^{(l)} + \mathbf{b}^{(l)} \quad (5.52)$$

$$\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) \quad (5.53)$$

$$\dots \quad (5.54)$$

$$(5.55)$$

### Backpropagation

The equations for the backpropagation read:

$$\dots \quad (5.56)$$

$$\overline{\mathbf{a}^{(l)}} = \mathbf{W}^{(l+1) T} \overline{\mathbf{z}^{(l+1)}} \quad (5.57)$$

$$\overline{\mathbf{z}^{(l)}} = \overline{\mathbf{a}^{(l)}} \odot \sigma'(\mathbf{z}^{(l)}) \quad (5.58)$$

$$\overline{\mathbf{b}^{(l)}} = \overline{\mathbf{z}^{(l)}} \quad (5.59)$$

$$\overline{\mathbf{W}^{(l)}} = \mathbf{a}^{(l-1)} \overline{\mathbf{z}^{(l) T}} \quad (5.60)$$

$$\dots \quad (5.61)$$

$$(5.62)$$

Circle dot indicates the element-wise product and it is also called **Hadamard product**, whereas when we

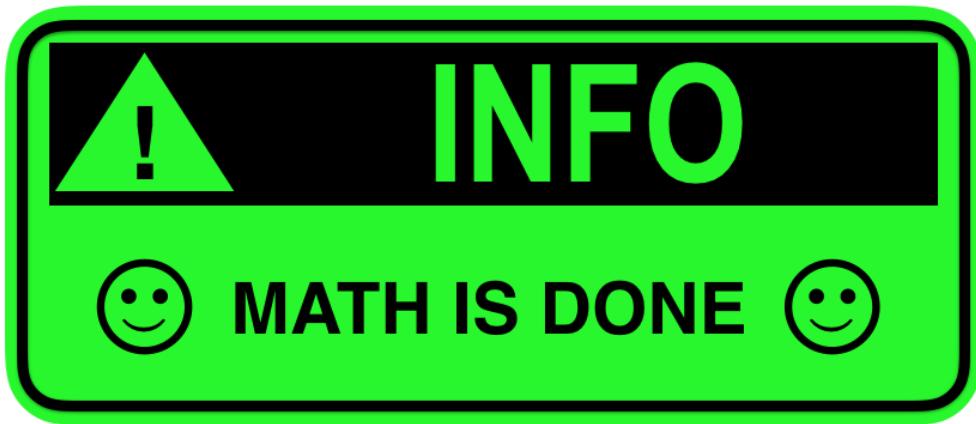
have two matrices next to each other, we indicate the matrix multiplication is taking place.

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.
2. 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.
3. 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.
4. 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.



Banknotes

First of all let's load and inspect the dataset:

```
In [28]: df = pd.read_csv('../data/banknotes.csv')
df.head()
```

Out[28] :

	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:

```
In [29]: df['class'].value_counts()
```

Out[29] :

class
0 762
1 610

We can also calculate the fraction of the larger class by dividing the first row by the total number of rows:

```
In [30]: df['class'].value_counts()[0]/len(df)
```

Out[30] : 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:

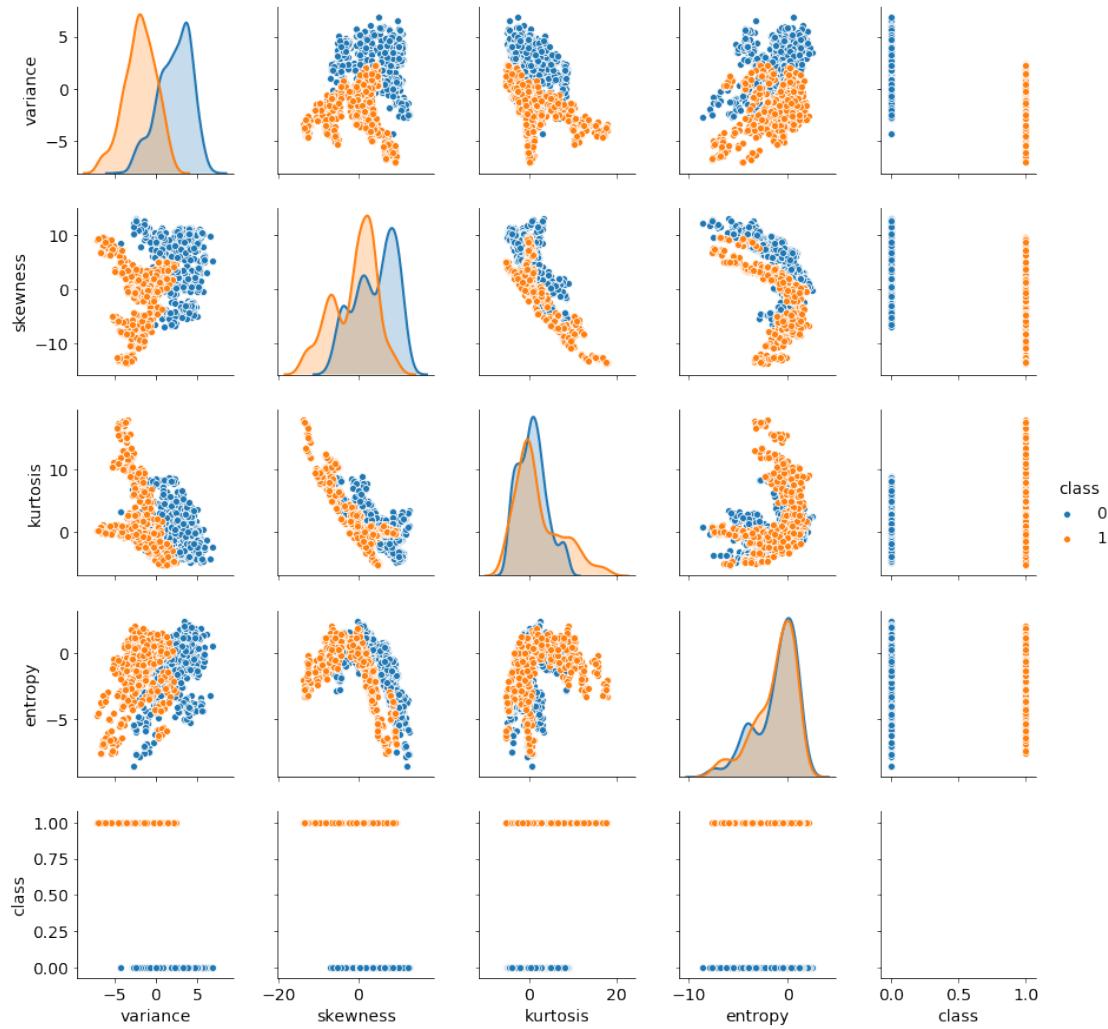
```
In [31]: import seaborn as sns
```

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!

```
In [32]: sns.pairplot(df, hue="class")
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kde.py:488: RuntimeWarning: invalid value
encountered in true_divide
    binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kdetools.py:34: RuntimeWarning: invalid value
encountered in double_scalars
    FAC1 = 2*(np.pi*bw/RANGE)**2
```

```
Out[32]: <seaborn.axisgrid.PairGrid at 0x7f18d2994198>
```



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:

```
In [33]: from sklearn.ensemble import RandomForestClassifier
```

and let's create an instance of the model with default parameters:

```
In [34]: model = RandomForestClassifier()
```

Now let's separate the features from labels as usual:

```
In [35]: 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:

```
In [36]: 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.

```
In [37]: cross_val_score(model, X, y)
```

```
Out[37]: array([0.99126638, 0.9868709 , 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.

```
In [38]: from sklearn.model_selection import train_test_split
```

Let's split up our data using the `train_test_split` function:

```
In [39]: 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:

```
In [40]: model.fit(X_train, y_train)
model.score(X_test, y_test)
```

```
Out[40]: 0.9927184466019418
```

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.

$$\frac{dJ(y, \hat{y}(w, X))}{dw}$$

$X$  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:

```
In [41]: from keras.models import Sequential
        from keras.layers import Dense, Activation
```

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:

```
In [42]: 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:

$$J(\hat{y}, y) = - (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (5.63)$$

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.

```
In [43]: 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.

```
In [44]: 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.

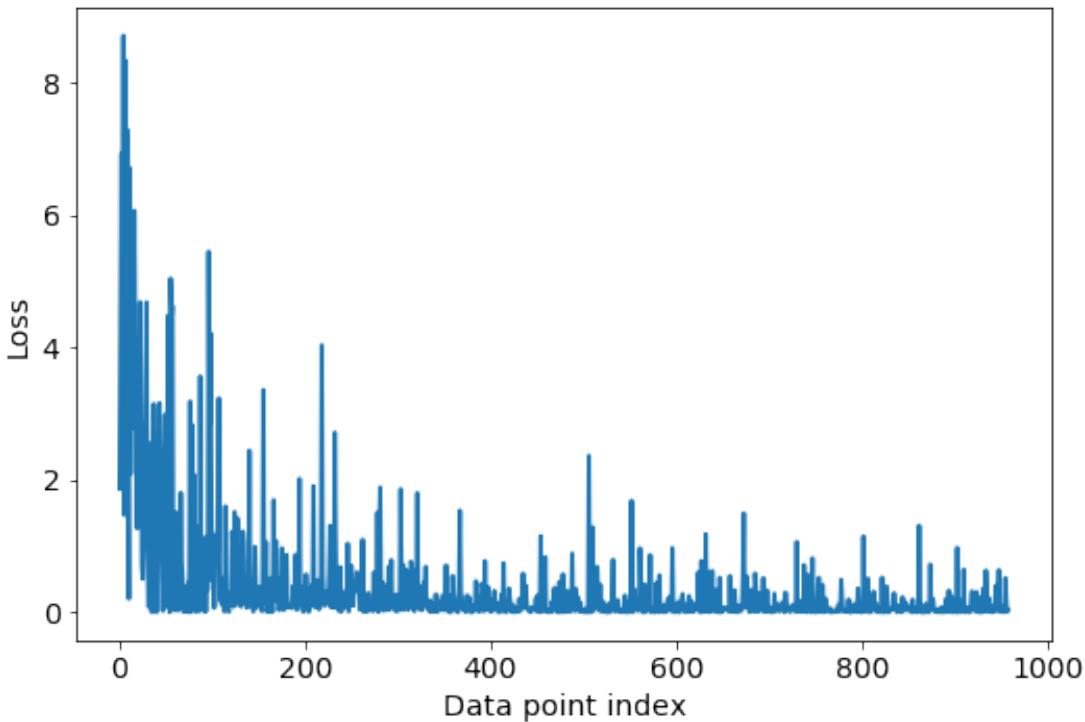
Let's train the model one point at a time first, for one epoch, i.e. passing all of the training data once:

```
In [45]: 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:

```
In [46]: plt.plot(losses)
plt.title('Binary Crossentropy Loss, One Epoch')
plt.xlabel('Data point index')
plt.ylabel('Loss');
```

### Binary Crossentropy Loss, One Epoch



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:

```
In [47]: model.set_weights(weights)
```

Now let's train the model with batches of 16 points each:

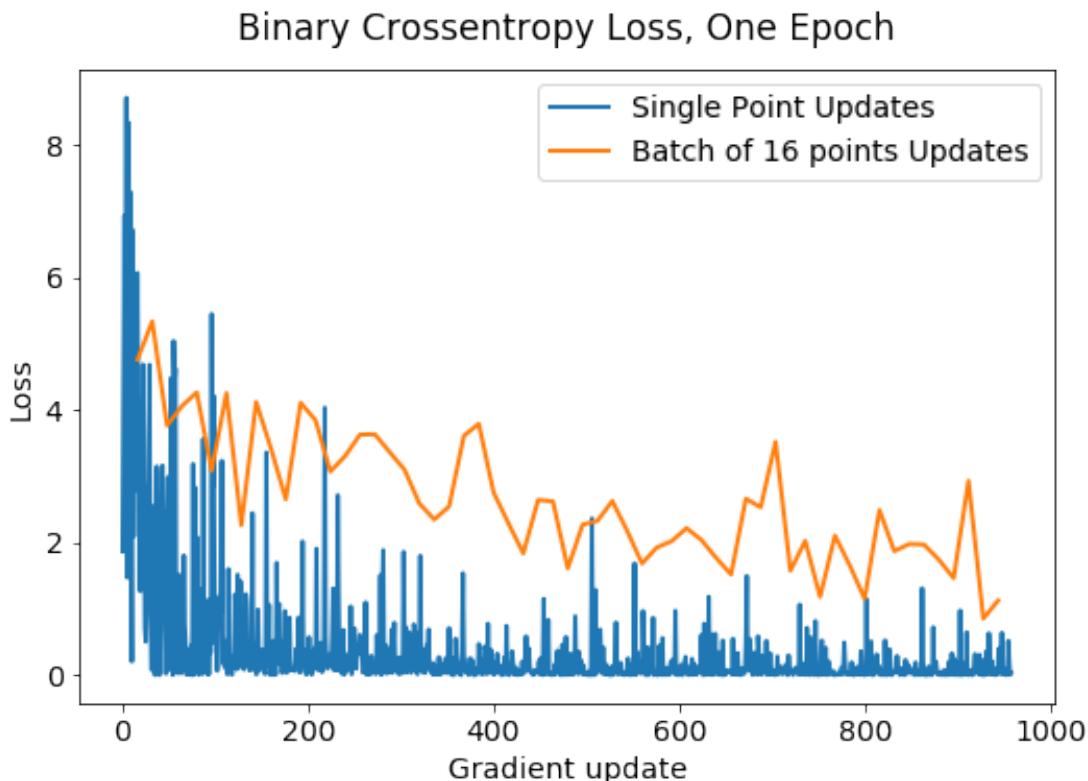
```
In [48]: B = 16
```

```
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:

```
In [49]: 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']);
```



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:

```
In [50]: model.set_weights(weights)
```

```
history = model.fit(X_train, y_train, batch_size=16,
                     epochs=20, verbose=0)
```

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`.

```
In [51]: result = model.evaluate(X_test, y_test)
      "Test accuracy: {:.2f} %".format(result[1]*100)
```

```
412/412 [=====] - 0s 89us/step
```

```
Out[51]: '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.

```
In [52]: from keras.optimizers import SGD
```

```
In [53]: 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))
```

We can concatenate all our results in a single file for easy visualization using the `pd.concat` function along the columns axis.

```
In [54]: historydf = pd.concat(dflist, axis=1)
```

In [55]: historydf

Out [55] :

	loss	acc	loss	acc	loss	acc	loss	acc
0	2.649101	0.251042	0.975350	0.671875	0.595705	0.809375	0.412307	0.907292
1	0.974554	0.557292	0.176434	0.942708	0.112163	0.971875	0.051603	0.980208
2	0.481472	0.791667	0.124239	0.971875	0.085588	0.978125	0.045794	0.982292
3	0.327701	0.885417	0.103229	0.976042	0.072776	0.979167	0.044522	0.984375
4	0.256964	0.911458	0.091143	0.979167	0.064128	0.983333	0.039981	0.983333
5	0.216080	0.929167	0.081839	0.973958	0.058486	0.985417	0.036770	0.986458
6	0.189600	0.938542	0.075496	0.976042	0.053476	0.986458	0.038738	0.984375
7	0.170926	0.942708	0.070037	0.981250	0.050074	0.986458	0.043822	0.981250
8	0.156660	0.953125	0.065652	0.984375	0.046785	0.985417	0.031647	0.987500
9	0.145569	0.956250	0.062748	0.984375	0.045427	0.987500	0.029799	0.990625

And we can add information about the learning rate in a secondary column index using the `pd.MultiIndex` class.

In [57]: historydf

Out [57] :

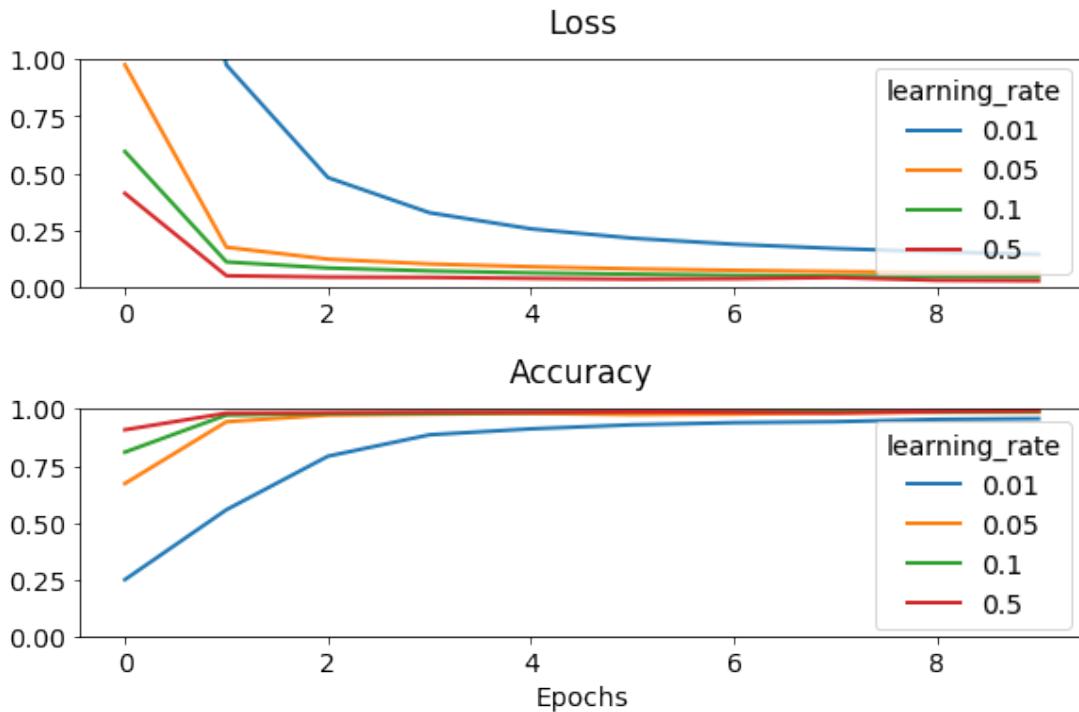
learning_rate	0.01		0.05		0.10		0.50	
metric	loss	acc	loss	acc	loss	acc	loss	acc
0	2.649101	0.251042	0.975350	0.671875	0.595705	0.809375	0.412307	0.907292
1	0.974554	0.557292	0.176434	0.942708	0.112163	0.971875	0.051603	0.980208
2	0.481472	0.791667	0.124239	0.971875	0.085588	0.978125	0.045794	0.982292
3	0.327701	0.885417	0.103229	0.976042	0.072776	0.979167	0.044522	0.984375
4	0.256964	0.911458	0.091143	0.979167	0.064128	0.983333	0.039981	0.983333
5	0.216080	0.929167	0.081839	0.973958	0.058486	0.985417	0.036770	0.986458
6	0.189600	0.938542	0.075496	0.976042	0.053476	0.986458	0.038738	0.984375
7	0.170926	0.942708	0.070037	0.981250	0.050074	0.986458	0.043822	0.981250
8	0.156660	0.953125	0.065652	0.984375	0.046785	0.985417	0.031647	0.987500
9	0.145569	0.956250	0.062748	0.984375	0.045427	0.987500	0.029799	0.990625

Now we can display the behavior of loss and accuracy as a function of the learning rate.

```
In [58]: ax = plt.subplot(211)
        hxs = historydf.xs('loss', axis=1, level='metric')
        hxs.plot(ylim=(0,1), ax=ax)
        plt.title("Loss")

        ax = plt.subplot(212)
        hxs = historydf.xs('acc', axis=1, level='metric')
        hxs.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.

```
In [59]: 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.

```
In [60]: 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
```

```
In [61]: historydf
```

```
Out[61]:
```

batch_size	1	4	16	32	64	128				
metric	loss	acc								
0	0.409446	0.869792	1.121512	0.623958	2.652906	0.250000	3.320561	0.186458	3.818151	0.127083
1	0.091748	0.972917	0.209368	0.929167	0.970648	0.543750	1.955286	0.330208	2.814991	0.238542
2	0.069933	0.980208	0.142254	0.959375	0.477751	0.797917	1.192059	0.463542	2.192038	0.307292
3	0.058504	0.986458	0.116638	0.972917	0.326435	0.881250	0.756515	0.647917	1.718532	0.353125
4	0.052077	0.984375	0.101401	0.971875	0.256029	0.912500	0.538703	0.767708	1.338816	0.423958
5	0.047357	0.985417	0.092057	0.976042	0.215487	0.927083	0.421322	0.827083	1.045915	0.516667
6	0.044281	0.986458	0.084392	0.977083	0.189522	0.939583	0.350943	0.872917	0.82616	0.604167
7	0.041591	0.988542	0.078510	0.977083	0.170528	0.944792	0.303833	0.896875	0.683029	0.685417
8	0.039047	0.986458	0.073547	0.978125	0.156586	0.952083	0.269729	0.906250	0.577486	0.742708
9	0.038182	0.988542	0.069614	0.982292	0.145689	0.955208	0.244082	0.917708	0.500669	0.791667
10	0.036937	0.988542	0.066473	0.983333	0.136822	0.965625	0.224280	0.927083	0.443691	0.819792
11	0.035518	0.988542	0.063356	0.982292	0.129631	0.967708	0.208520	0.931250	0.399779	0.840625
12	0.034495	0.987500	0.060760	0.984375	0.123468	0.969792	0.195154	0.935417	0.364787	0.864583
13	0.033401	0.988542	0.058514	0.986458	0.118133	0.972917	0.184068	0.939583	0.336677	0.884375
14	0.032379	0.988542	0.056788	0.983333	0.113554	0.971875	0.174797	0.942708	0.313450	0.890625
15	0.032016	0.988542	0.054849	0.985417	0.109539	0.975000	0.166773	0.945833	0.293672	0.897917
16	0.031432	0.987500	0.053285	0.985417	0.105950	0.978125	0.159736	0.950000	0.276935	0.903125
17	0.031232	0.988542	0.051734	0.986458	0.102629	0.979167	0.153550	0.951042	0.262416	0.907292
18	0.030454	0.989583	0.050314	0.986458	0.099699	0.977083	0.148115	0.955208	0.249905	0.914583
19	0.029632	0.987500	0.049410	0.986458	0.096997	0.980208	0.143223	0.957292	0.238602	0.921875

```
In [62]: ax = plt.subplot(211)
hxs = historydf.xs('loss', axis=1, level='metric')
```

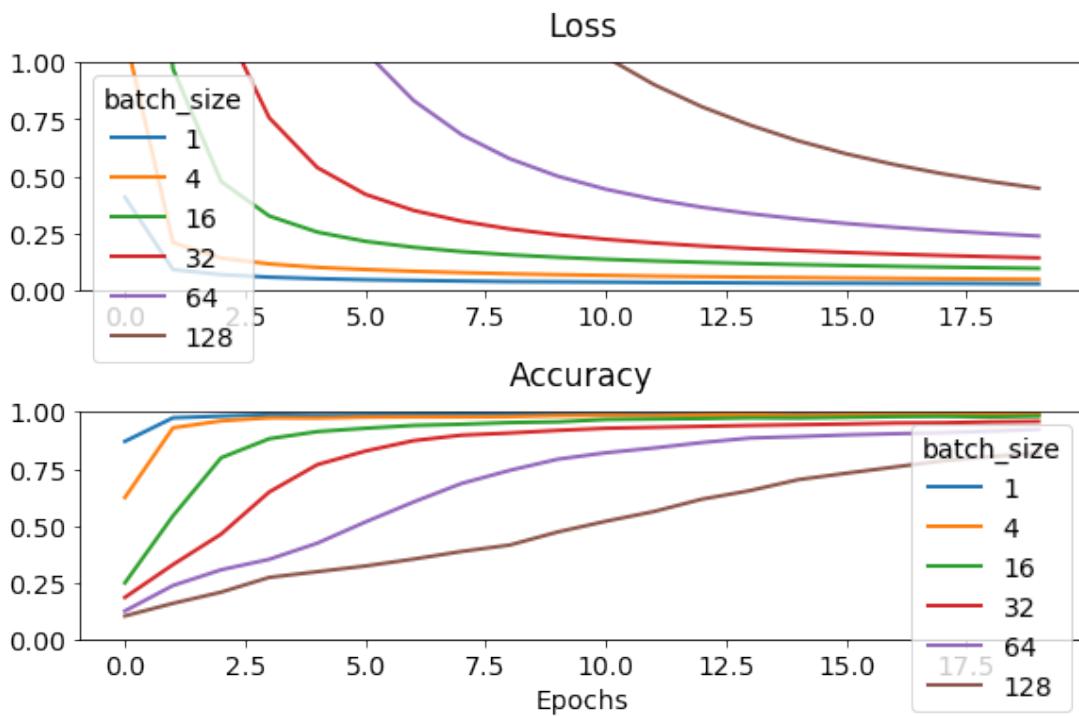
```

hxs.plot(ylim=(0,1), ax=ax)
plt.title("Loss")

ax = plt.subplot(212)
hxs = historydf.xs('acc', axis=1, level='metric')
hxs.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 [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  $\eta$
- 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  $\eta$  and a momentum parameter  $\mu$
- 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:  $w \leftarrow w - v$

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  $\eta$ , a small constant  $\delta = 10^{-7}$  to avoid division by zero
- Repeat until stop rule:
  - Same 3 steps as SGD (get batch, evaluate cost, evaluate gradient)
  - Accumulate the square of the gradient:  $r \leftarrow r + g \odot g$
  - Compute update:  $\Delta w = \eta \frac{1}{\delta + \sqrt{r}} \odot g$
  - Apply the update rule  $w \leftarrow w - \Delta w$

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  $w_i$ ,  $g \odot g$  is equivalent to  $g_i^2$ , 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.  $\eta$  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  $r_i$ . 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  $\delta$  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  $\eta$ , a small constant  $\delta = 10^{-7}$  to avoid division by zero and an EWMA mixing factor  $\rho$  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:  $r \leftarrow \rho r + (1 - \rho)g \odot g$
  - 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  $\eta$ , a small constant  $\delta = 10^{-7}$  to avoid division by zero and an EWMA mixing factors  $\rho_1$  and  $\rho_2$  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:  $v \leftarrow \rho_1 v + (1 - \rho_1)g$
  - Accumulate EWMA the square of the gradient:  $r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$
  - Correct bias 1:  $\hat{v} = \frac{v}{1 - \rho_1^t}$
  - Correct bias 2:  $\hat{r} = \frac{r}{1 - \rho_2^t}$
  - Compute update:  $\Delta w = \eta \frac{1}{\delta + \sqrt{\hat{r}}} \odot \hat{v}$
  - Apply the update rule  $w \leftarrow w - \Delta w$

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:

```
In [63]: 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:

```
In [64]: dflist = []

opts = ['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 opts:
    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:

```
In [65]: historydf = pd.concat(dflist, axis=1)
metrics_ = dflist[0].columns
idx = pd.MultiIndex.from_product([opts, metrics_],
                                 names=['optimizers',
                                         'metric'])
historydf.columns = idx
```

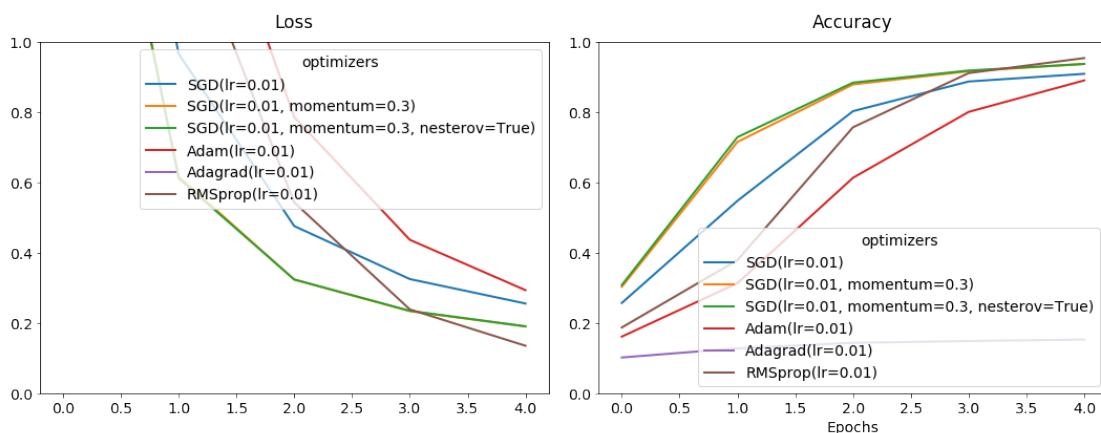
And plot them for comparison:

```
In [66]: plt.figure(figsize=(15, 6))

ax = plt.subplot(121)
hxs = historydf.xs('loss', axis=1, level='metric')
hxs.plot(ylim=(0,1), ax=ax)
plt.title("Loss")

ax = plt.subplot(122)
hxs = historydf.xs('acc', axis=1, level='metric')
hxs.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 limit is  $\frac{\sqrt{3}}{\# \text{ inputs}}$ . 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.

```
In [67]: import keras.backend as K
```

```
In [68]: dflist = []
```

```
inits = ['zeros', 'ones', 'uniform', 'lecun_uniform',
        'normal', 'he_normal', 'glorot_normal']
```

```

for init in inits:

    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

```

In [69]: historydf = pd.concat(dflist, axis=1)
         metrics_ = dflist[0].columns
         idx = pd.MultiIndex.from_product([inits, metrics_],
                                           names=['initializers',
                                                  'metric'])
         historydf.columns = idx

In [70]: styles = ['-+', '-*', '-x', '-d', '-^', '-o', '-s']

```

```

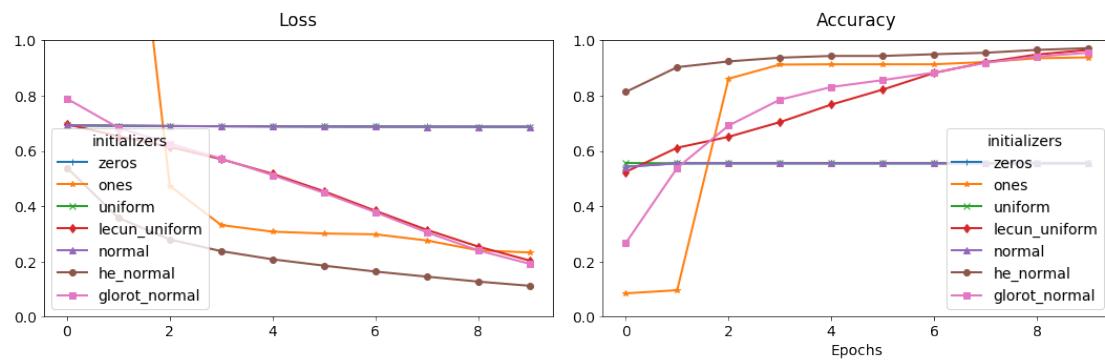
plt.figure(figsize=(15, 5))

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:

```
In [71]: K.clear_session()
```

Then we define and compile the model:

```
In [72]: 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:

```
In [73]: 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

```
In [74]: h = model.fit(X_train, y_train,
                      batch_size=16, epochs=20,
                      verbose=0, validation_split=0.3)
```

Let's look at the layers using the `model.summary` function:

```
In [75]: 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  
Non-trainable params: 0

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:

```
In [76]: model.layers
```

```
Out[76]: [<keras.layers.core.Dense at 0x7f1708e77a90>,
           <keras.layers.core.Dense at 0x7f17980757b8>]
```

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:

```
In [77]: 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.

```
In [78]: inp
```

```
Out[78]: <tf.Tensor 'dense_1_input:0' shape=(?, 4) dtype=float32>
```

```
In [79]: out
```

```
Out[79]: <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:

```
In [80]: features_function = K.function([inp], [out])
```

Notice that `features_function` is a function itself, so `K.function` is a function that returns a function.

```
In [81]: features_function
```

```
Out[81]: <keras.backend.tensorflow_backend.Function at 0x7f16cc0e3a58>
```

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:

```
In [82]: features = features_function([X_test])[0]
```

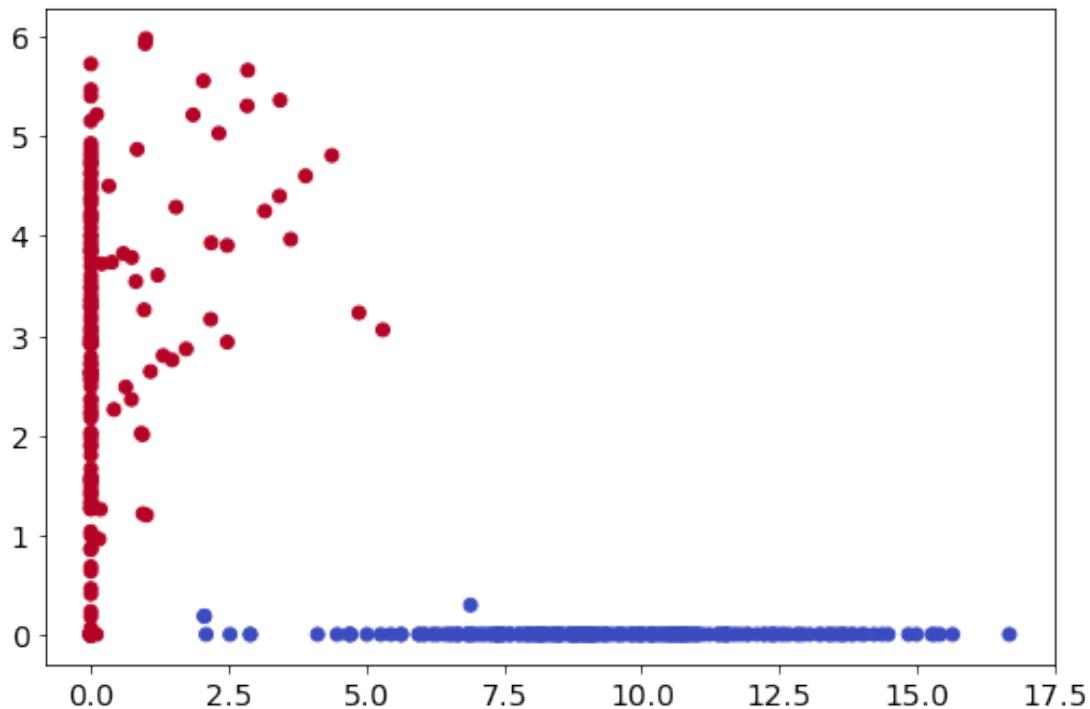
The output tensor contains as many points as X\_test each represented by 2 numbers, the output values of the 2 nodes in the first layer:

```
In [83]: features.shape
```

```
Out[83]: (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:

```
In [84]: plt.scatter(features[:, 0], features[:, 1], c=y_test, cmap='coolwarm');
```



Let's plot the output of the second-to-last layer at each epoch in a training loop. First we re-initialize the model:

```
In [85]: model.set_weights(weights)
```

Then we create a K.function between the input and the output of layer o:

```
In [86]: inp = model.layers[0].input
        out = model.layers[0].output
        features_function = K.function([inp], [out])
```

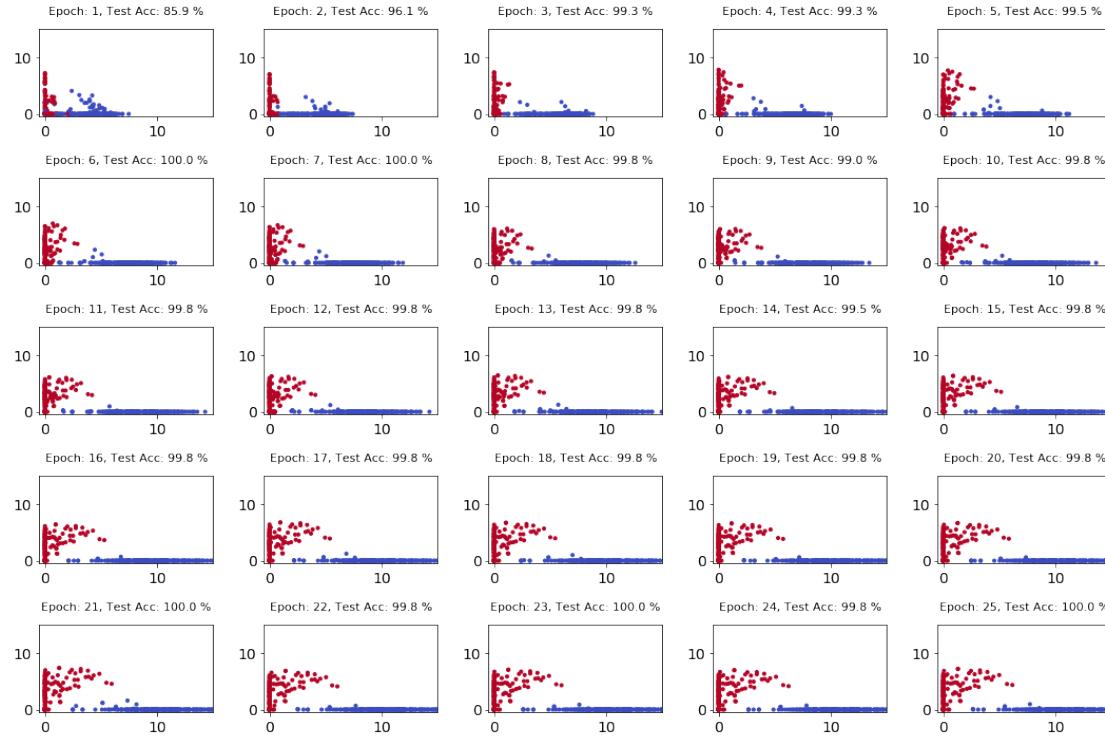
Then we train the model one epoch at a time, plotting the 2D representation of the data as it comes out from layer o:

```
In [87]: 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', marker='.')
        plt.xlim(-0.5, 15)
        plt.ylim(-0.5, 15)

        acc_ = test_acc * 100.0
        t = 'Epoch: {}, Test Acc: {:.3f} %'.format(i, acc_)
        plt.title(t, fontsize=11)

    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

# 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



Picture of a cat

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](#):

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:  
    exec(fin.read())
```

Let's use the `np.random.binomial` function to generate a  $10 \times 10$  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  $n$  ( $\geq 0$ ) 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:

```
In [3]: bw = np.random.binomial(1, 0.5, size=(10, 10))
```

Let's print out the array `bw`:

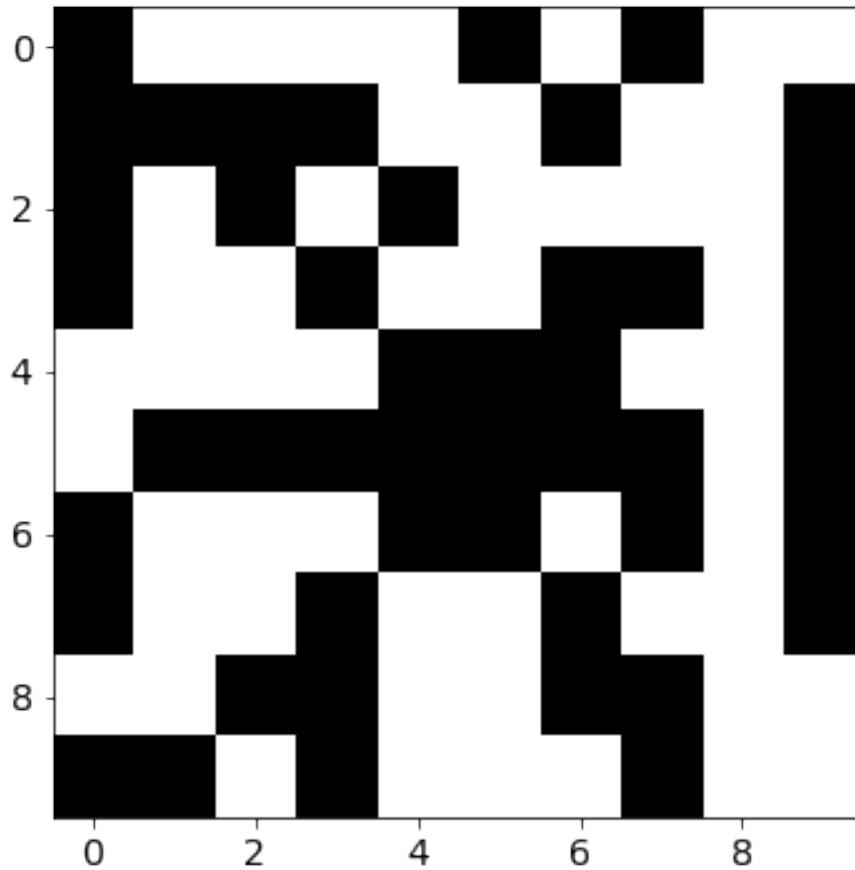
```
In [4]: bw
```

```
Out[4]: array([[0, 1, 1, 1, 1, 0, 1, 0, 1, 1],  
               [0, 0, 0, 0, 1, 1, 0, 1, 1, 0],  
               [0, 1, 0, 1, 0, 1, 1, 1, 1, 0],  
               [0, 1, 1, 0, 1, 1, 0, 0, 1, 0],  
               [1, 1, 1, 1, 0, 0, 0, 1, 1, 0],  
               [1, 0, 0, 0, 0, 0, 0, 0, 1, 0],  
               [0, 1, 1, 1, 0, 0, 1, 0, 1, 0],  
               [0, 1, 1, 0, 1, 1, 0, 1, 1, 0],  
               [1, 1, 0, 0, 1, 1, 0, 0, 1, 1],  
               [0, 0, 1, 0, 1, 1, 1, 0, 1, 1]])
```

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:

```
In [5]: plt.imshow(bw, cmap='gray')  
plt.title("Black and White pixels");
```

## Black and White pixels



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 distribution.

```
In [6]: gs = np.random.randint(0, 256, size=(10, 10))
```

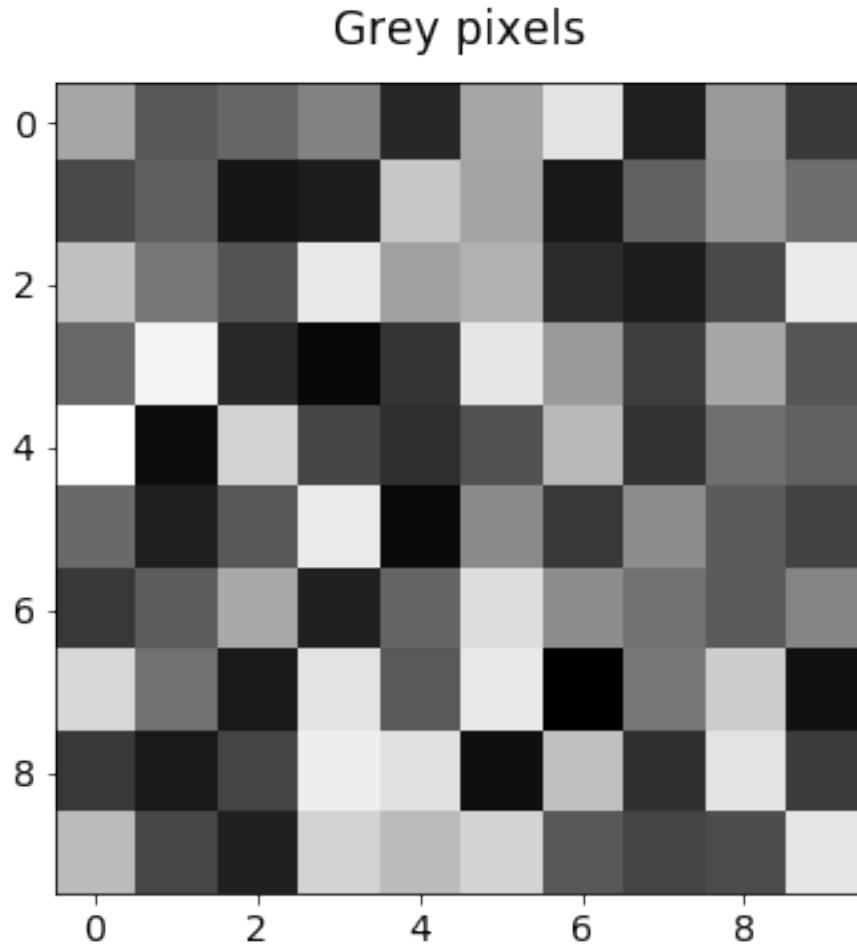
Let's print out the array *gs*:

```
In [7]: gs
```

```
Out[7]: array([[168,  92, 106, 133,  44, 167, 228,  37, 157,  63],
   [ 78, 100,  28,  35, 200, 165,  30, 101, 151, 113],
   [193, 121,  88, 233, 163, 179,  48,  35,  78, 235],
   [108, 243,  46,  13,  58, 230, 156,  67, 169,  89],
   [255,  19, 211,  74,  52,  85, 185,  55, 114, 102],
   [109,  37,  92, 235,  16, 141,  61, 144,  95,  72],
   [ 62,  96, 171,  39, 104, 221, 144, 116,  95, 136],
   [216, 116,  31, 228,  94, 233,    6, 121, 206,  23],
   [ 62,  31,  73, 238, 225,  21, 193,  53, 227,  64],
   [188,  75,  38, 213, 188, 213,  92,  73,  80, 230]])
```

As expected it's a 10x10 grid of random integers between 0 and 255. Let's visualize it as an image:

```
In [8]: plt.imshow(gs, cmap='gray')
plt.title("Grey pixels");
```



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

```
In [9]: from keras.datasets import mnist
```

Using TensorFlow backend.

```
In [10]: (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:

```
In [11]: X_train.shape
```

```
Out[11]: (60000, 28, 28)
```

```
In [12]: X_test.shape
```

```
Out[12]: (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:

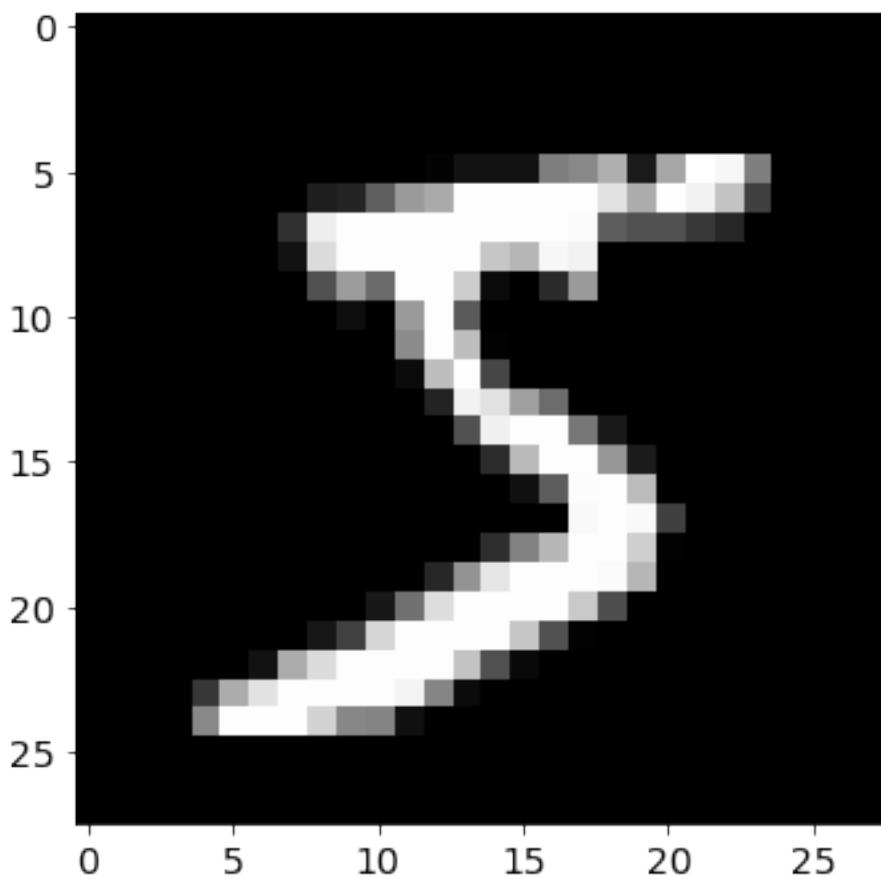
```
In [13]: first_img = X_train[0]
```

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:

```
In [14]: plt.imshow(first_img, cmap='gray');
```

The first number in the mnist training set is a 5



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:

```
In [15]: X_train_flat = X_train.reshape((60000, 784))
```

We can check that the operation worked by printing the shape of `X_train_flat`:

```
In [16]: X_train_flat.shape
```

```
Out[16]: (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:

```
In [17]: X_test_flat = X_test.reshape(-1, 28*28)
```

Again, let's print the shape to be sure:

```
In [18]: X_test_flat.shape
```

```
Out[18]: (10000, 784)
```

Great! Now we have 2 tabular datasets like the ones we are familiar with. The features contain values between 0 and 255:

```
In [19]: X_train_flat.min()
```

```
Out[19]: 0
```

```
In [20]: X_train_flat.max()
```

```
Out[20]: 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.

```
In [21]: 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:

```
In [22]: y_train
```

```
Out[22]: 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:

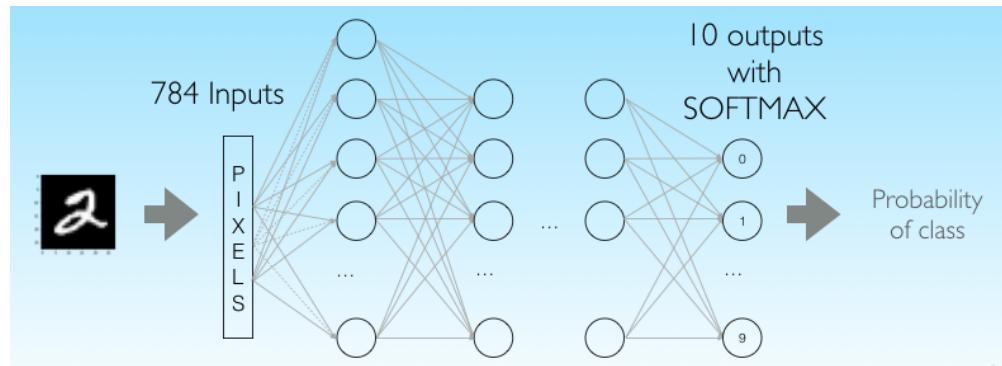
```
In [23]: np.unique(y_train)
```

```
Out[23]: 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:



Fully connected network to solve MNIST

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

```
In [24]: from keras.utils.np_utils import to_categorical
```

and let's convert both `y_train` and `y_test`:

```
In [25]: 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.

```
In [26]: y_train[0]
```

```
Out[26]: 5
```

The corresponding binary version of the label is the following array:

In [27]: `y_train_cat[0]`

Out[27]: `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.

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:

In [28]: `y_train_cat.shape`

Out[28]: `(60000, 10)`

Let's check our test dataset to make sure it matches as well.

In [29]: `y_test_cat.shape`

Out[29]: `(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:

```
In [30]: 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.

```
In [31]: 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

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

Let's print out the model summary:

```
In [32]: model.summary()
```

```

Layer (type)          Output Shape         Param #
=====
dense_1 (Dense)      (None, 512)          401920
=====
dense_2 (Dense)      (None, 256)          131328
=====
dense_3 (Dense)      (None, 128)          32896
=====
dense_4 (Dense)      (None, 32)           4128
=====
dense_5 (Dense)      (None, 10)           330
=====

Total params: 570,602
Trainable params: 570,602
Non-trainable params: 0
=====
```

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.

```
In [33]: h = model.fit(X_train_sc, y_train_cat, batch_size=128,
                     epochs=10, verbose=1,
                     validation_split=0.1)
```

```

Train on 54000 samples, validate on 6000 samples
Epoch 1/10
54000/54000 [=====] - 3s 53us/step - loss: 0.2886 -
acc: 0.9093 - val_loss: 0.0988 - val_acc: 0.9700
Epoch 2/10
54000/54000 [=====] - 1s 26us/step - loss: 0.1024 -
acc: 0.9693 - val_loss: 0.1996 - val_acc: 0.9382
Epoch 3/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0666 -
acc: 0.9800 - val_loss: 0.0976 - val_acc: 0.9738
Epoch 4/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0495 -
acc: 0.9847 - val_loss: 0.0725 - val_acc: 0.9803
Epoch 5/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0386 -
acc: 0.9887 - val_loss: 0.0930 - val_acc: 0.9758
Epoch 6/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0308 -
```

```
acc: 0.9906 - val_loss: 0.0855 - val_acc: 0.9823
Epoch 7/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0253 -
acc: 0.9925 - val_loss: 0.0853 - val_acc: 0.9825
Epoch 8/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0228 -
acc: 0.9933 - val_loss: 0.1108 - val_acc: 0.9792
Epoch 9/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0192 -
acc: 0.9946 - val_loss: 0.1006 - val_acc: 0.9820
Epoch 10/10
54000/54000 [=====] - 1s 26us/step - loss: 0.0179 -
acc: 0.9949 - val_loss: 0.0987 - val_acc: 0.9823
```

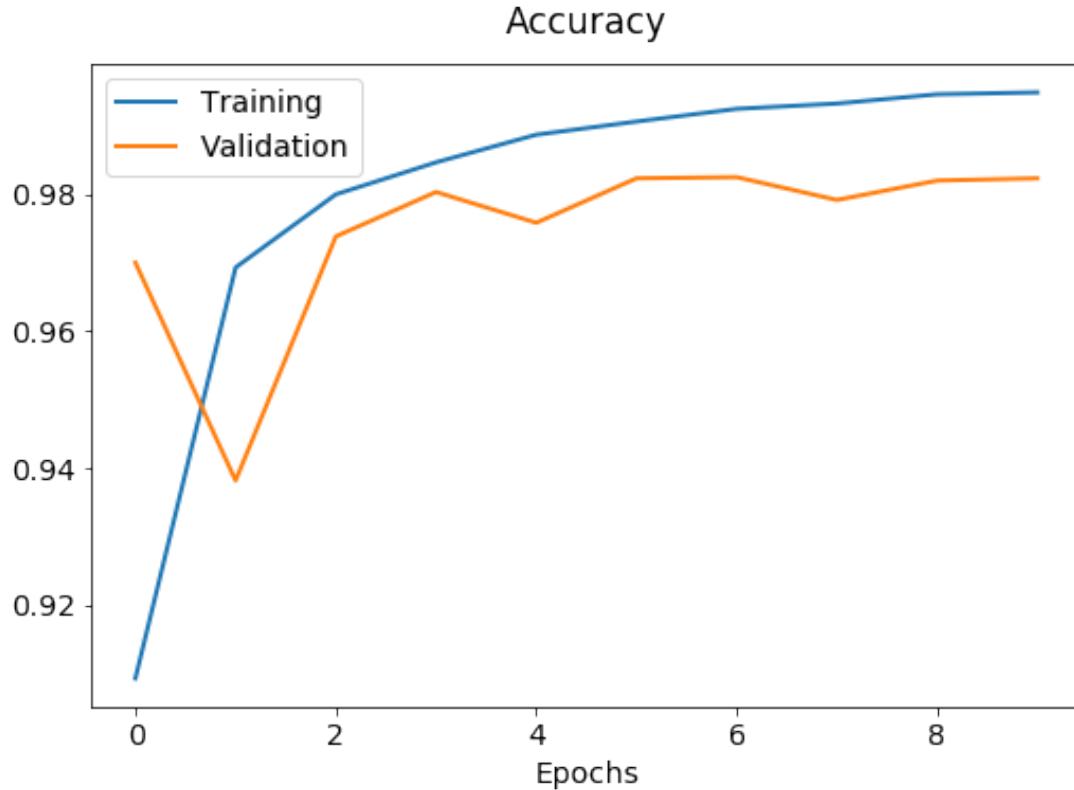
The model seems to be doing very well on the training data (as we can see by the acc output).

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.

```
In [34]: plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
plt.title('Accuracy')
plt.xlabel('Epochs');
```



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:

```
In [35]: test_acc = model.evaluate(X_test_sc, y_test_cat)[1]
      test_acc
```

```
10000/10000 [=====] - 0s 25us/step
```

```
Out[35]: 0.9813
```

and let's compare it with the performance on the training set:

```
In [36]: train_acc = model.evaluate(X_train_sc, y_train_cat)[1]
      train_acc
```

```
60000/60000 [=====] - 2s 26us/step
```

Out [36] : 0.9954666666666667

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 as 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



Feature extraction

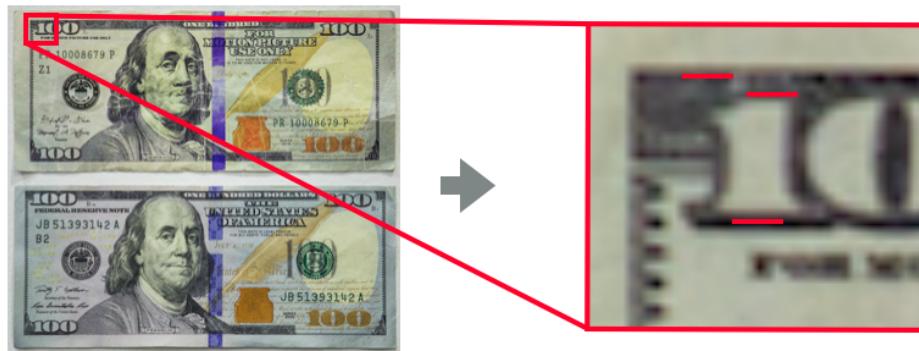
information we are trying to capture from a rich data point (like an image), is a time consuming and complex effort, often involving several Ph.D. students doing their thesis on it.

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.



Local patterns in images

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.

In [37]: 5

Out[37]: 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:

In [38]: `v = np.array([1, 3, 2, 1])  
v.shape`

Out[38]: (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.

```
In [39]: M = np.array([[1, 3, 2, 2],  
                      [0, 1, 3, 2]])  
M.shape
```

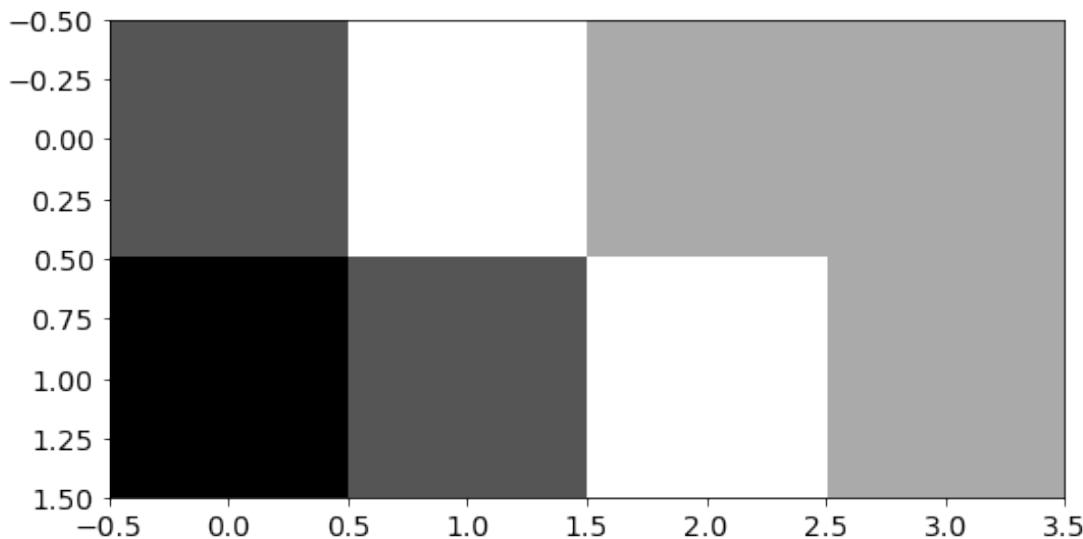
```
Out[39]: (2, 4)
```

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.

```
In [40]: plt.imshow(M, cmap='gray');
```



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:

In [41]: `M[0].shape`

Out[41]: (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.

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.

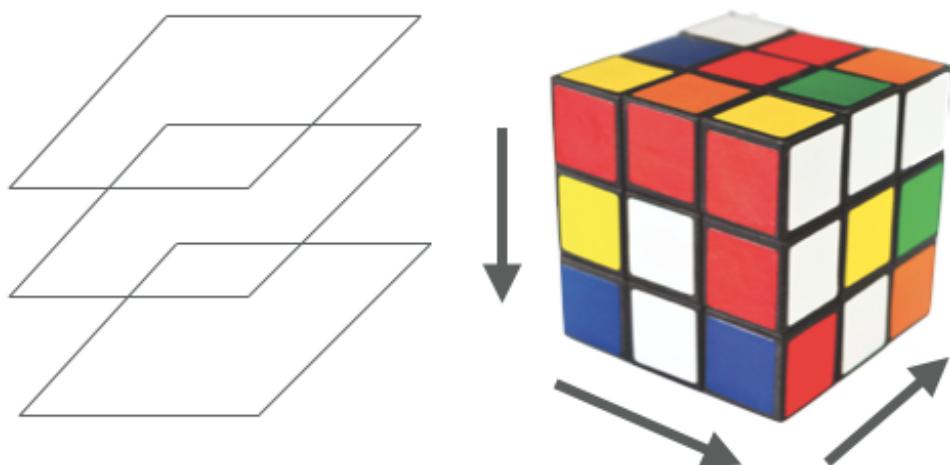
The shape of a tensor tells us how many objects there are when counting along a particular axis. So for example, a vector has only one axis and a matrix has two axes, indicating the number of rows and the number of columns. Since most of the data (images, sounds, texts, etc.) we will use are stored as 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.

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]]]

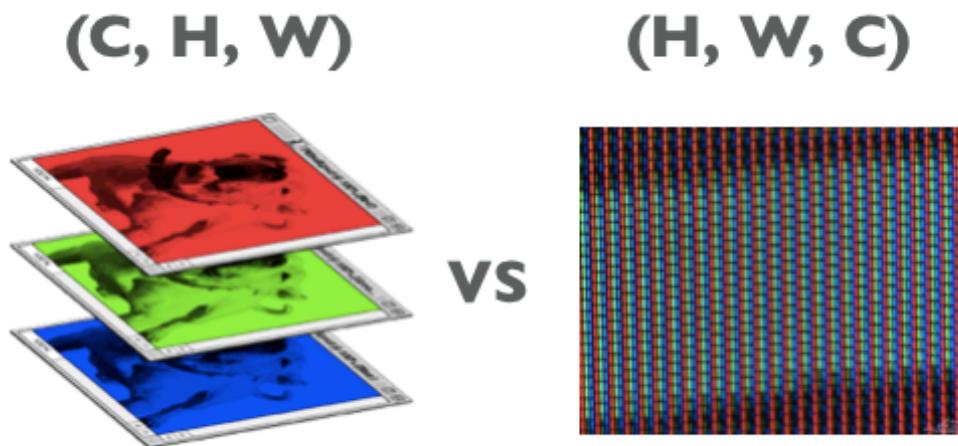
Tensors



Multi-dimensional array

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.



Channel order when an image is represented as a tensor

Let's create and display a random color image by creating a list of random pixels between 0 and 255:

```
In [42]: img = np.random.randint(255, size=(4, 4, 3),
                                 dtype='uint8')
img
```

```
Out[42]: array([[[116, 137, 246],
   [ 75,  31,  55],
   [157,  73,  80],
   [227,  56, 200]],

  [[131, 184,    8],
   [ 77,    5, 210],
   [ 50, 218,   58],
   [142,   12, 101]],

  [[ 30, 109,   63],
   [246, 167, 156],
   [ 36, 202,    9],
   [  6, 193,   27]],
```

```
[[170,   3,   53],
 [146, 251,   17],
 [180, 211,   82],
 [ 60, 177,   27]]], dtype=uint8)
```

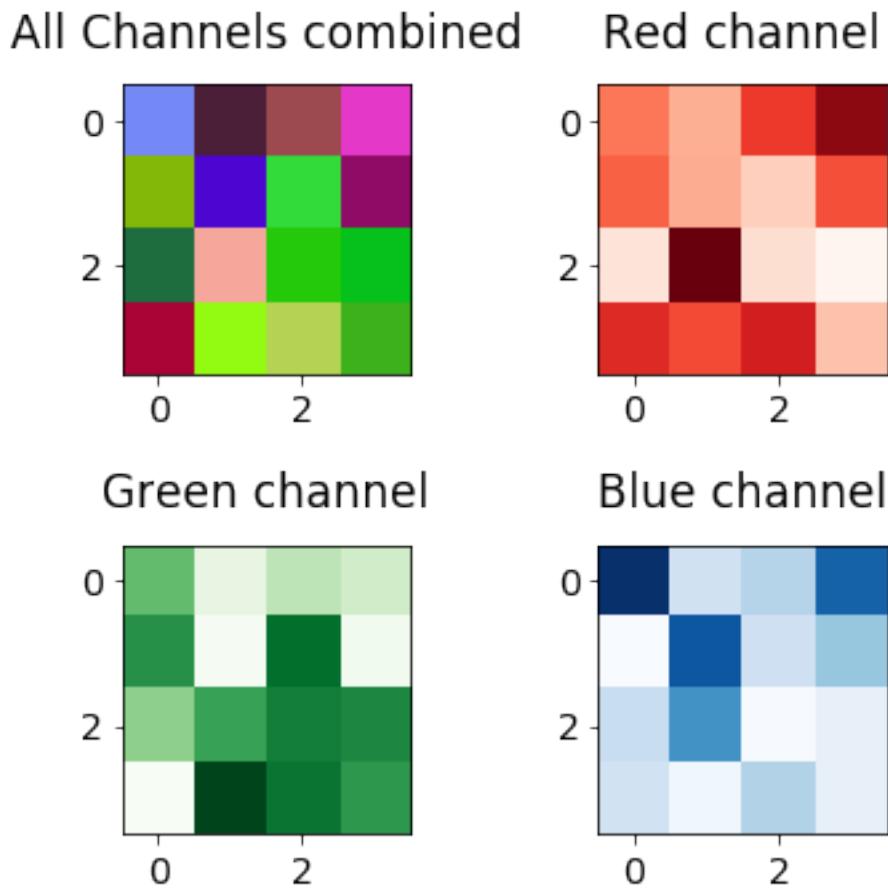
Now let's display it as a figure, showing each of the dominant pixels in each list.

```
In [43]: 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.

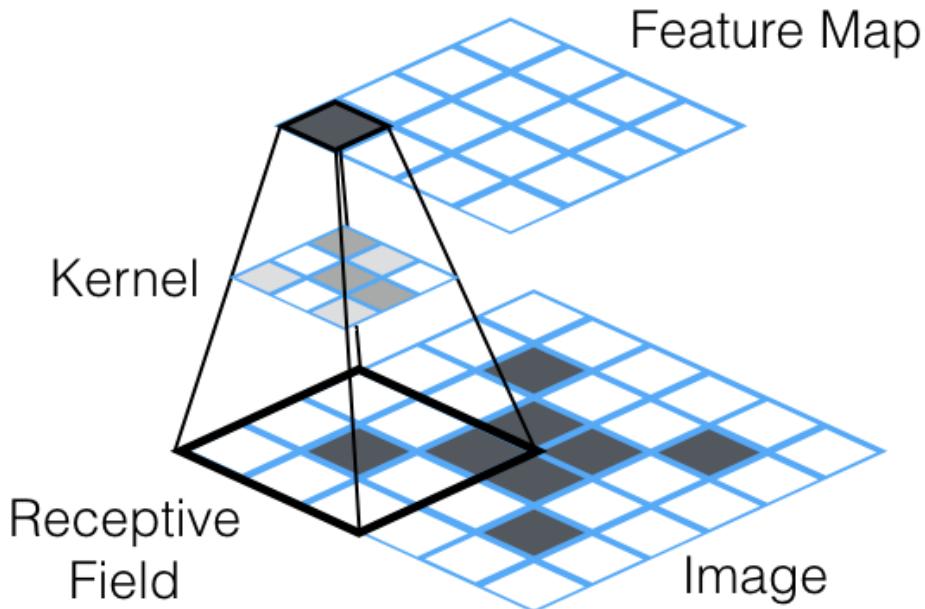
TIP: If you'd like to know a bit more about Tensors and how they work, we displayed a few operations in the [Appendix](#).

## 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 image, also called a **feature map**, whose pixels represent the “degree of matching” of the corresponding **receptive field** with the kernel.



Feature map and receptive field

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 axes 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).

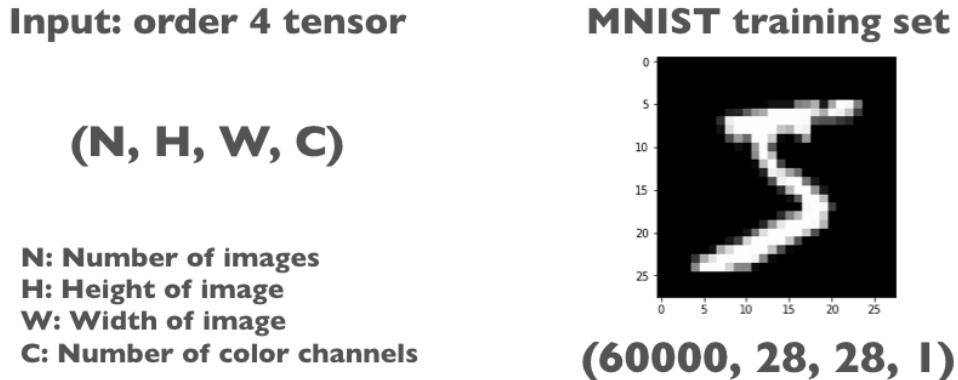


Image represented as tensor of order-4

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 3x3 filters. The order-4 tensor has a shape of (3, 3, 1, 4), i.e. four filters of 3x3 pixels each with a single input color channel each.

When we convolve each input image with the convolutional layer, we still obtain an order-4 tensor.

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

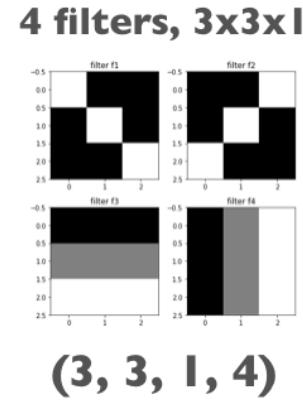
Convolutional layers are available in `keras.layers.Conv2D`. Let's apply a convolutional layer to an image and see what happens.

First, let's import the Conv2D layer from keras:

```
In [44]: from keras.layers import Conv2D
```

**CONV: order 4 tensor**  
**(Hf, Wf, Ci, Co)**

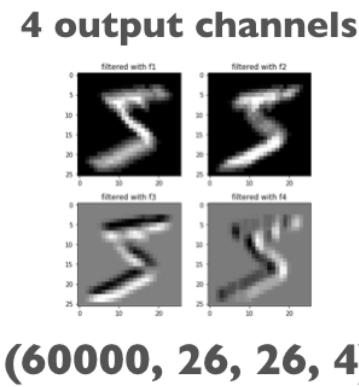
**Hf:** Height of filter patch  
**Wf:** Width of filter patch  
**Ci:** Channels in input  
**Co:** Channels in output (# filters)



Convolutional kernel represented as tensor of order-4

**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



Feature map represented as tensor of order-4

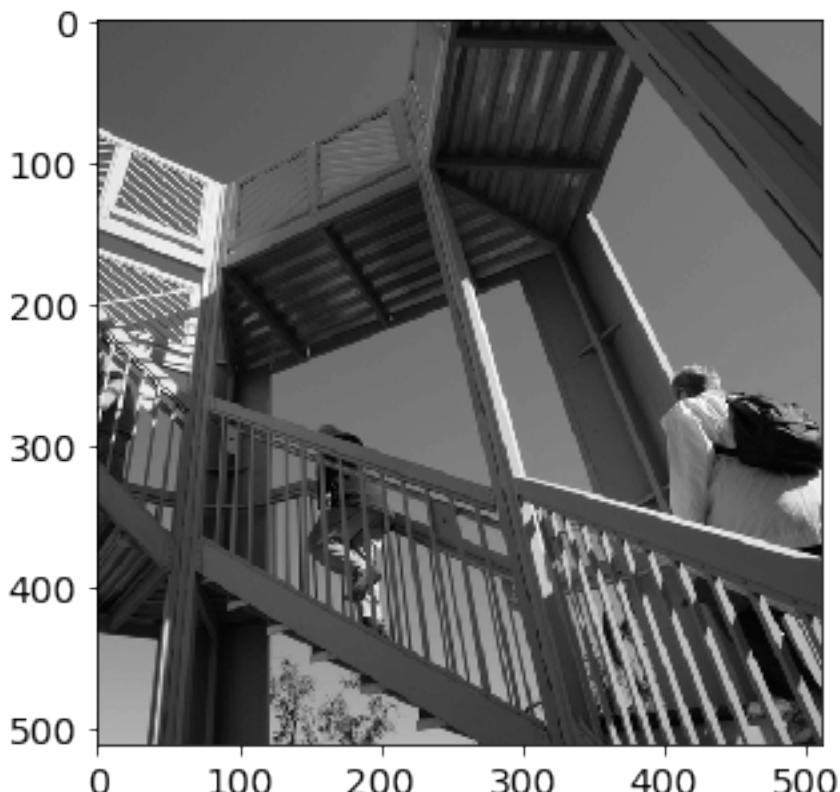
Now let's load an example image from `scipy.misc`:

```
In [45]: from scipy import misc
```

```
In [46]: img = misc.ascent()
```

and let's display it:

```
In [47]: plt.figure(figsize=(5, 5))
plt.imshow(img, cmap='gray');
```



Let's check the shape of `img`:

```
In [48]: img.shape
```

```
Out[48]: (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.

```
In [49]: 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).

```
In [50]: 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:

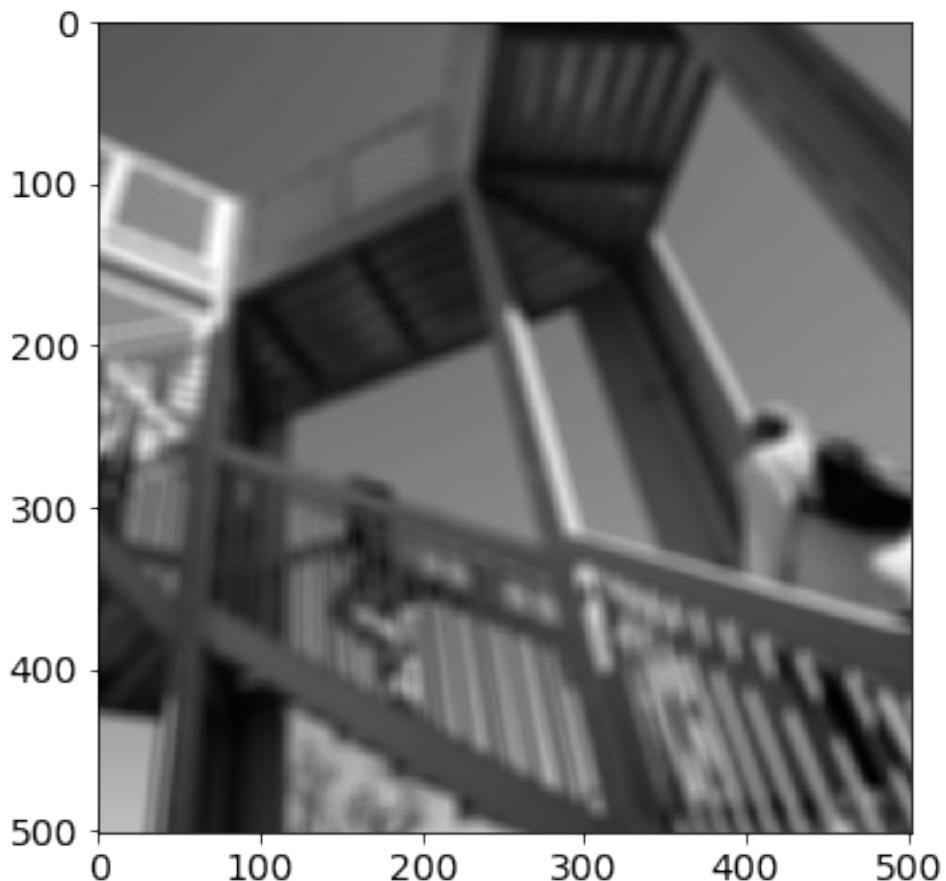
```
In [51]: img_pred_tensor = model.predict(img_tensor)
```

To visualize the image we extract it from the tensor.

```
In [52]: img_pred = img_pred_tensor[0, :, :, 0]
```

and we can use `plt.imshow` as before:

```
In [53]: plt.imshow(img_pred, cmap='gray');
```



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:

In [54]: `img_pred_tensor.shape`

Out [54]: (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.

In keras we have two padding modes: - `valid` which means **no padding** - `same` which means **pad** to keep the same image size.

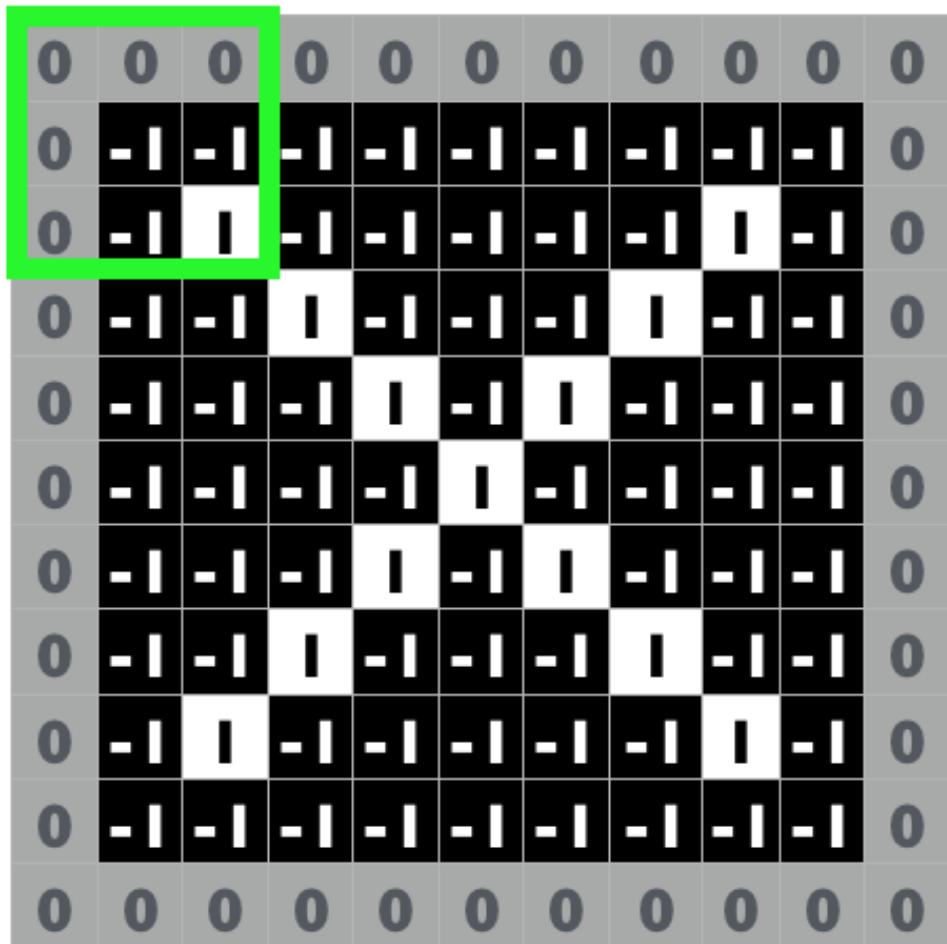
Let's check that `padding same` works as expected:

```
In [55]: 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
```

Out [55]: (1, 512, 512, 1)

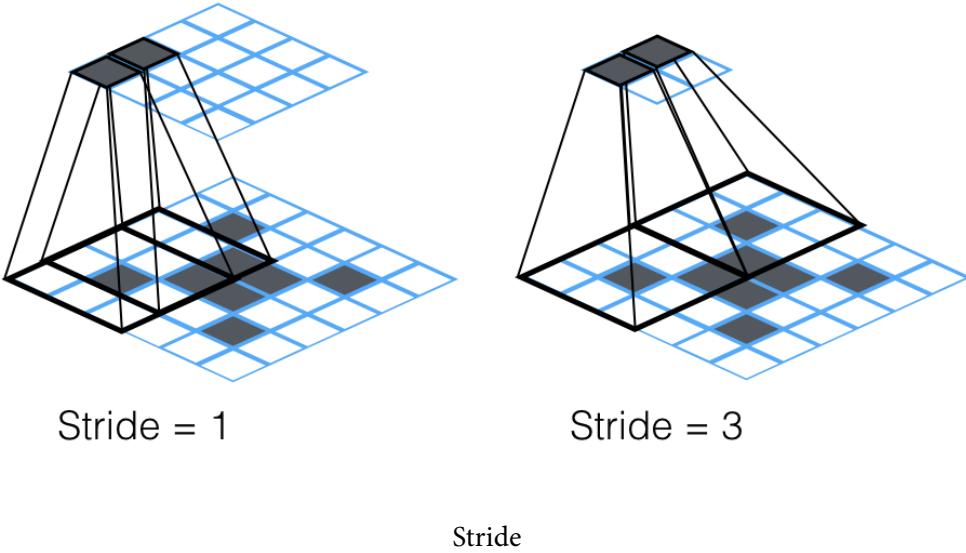
Awesome! We know how padding works. Why use padding? We can use padding if we think that the pixels at the border contain useful information to solve the classification task.



Padding

## 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:



The input image has size  $6 \times 6$ , the filter (not shown) is  $3 \times 3$  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  $4 \times 4$  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  $2 \times 2$ .

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.

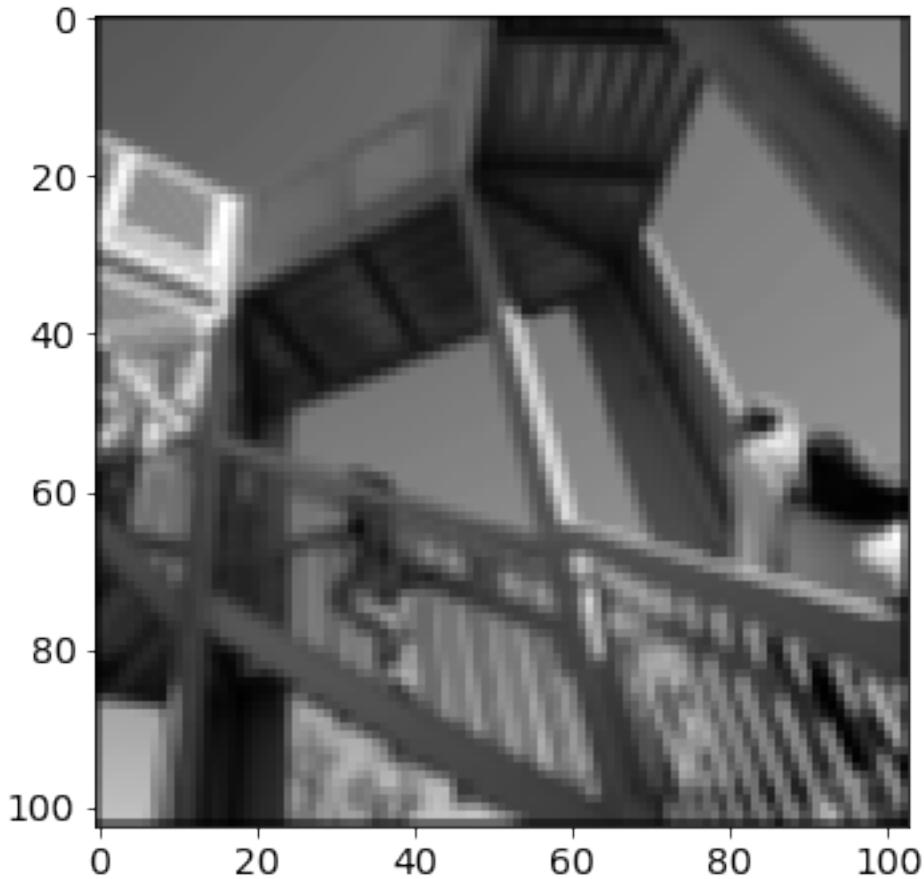
This will produce output images that are smaller. For example, let's jump by 5 pixels in both directions in our example:

```
In [56]: 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
```

Out[56]: (1, 103, 103, 1)

In [57]: plt.imshow(small\_img\_tensor[0, :, :, 0], cmap='gray');



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:

```
In [58]: 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
```

Out[58]: (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.

```
In [59]: from keras.layers import MaxPooling2D
         from keras.layers import AveragePooling2D
         from keras.layers import GlobalMaxPooling2D
```

Let's add a MaxPooling2D layer in a simple network (containing this single layer):

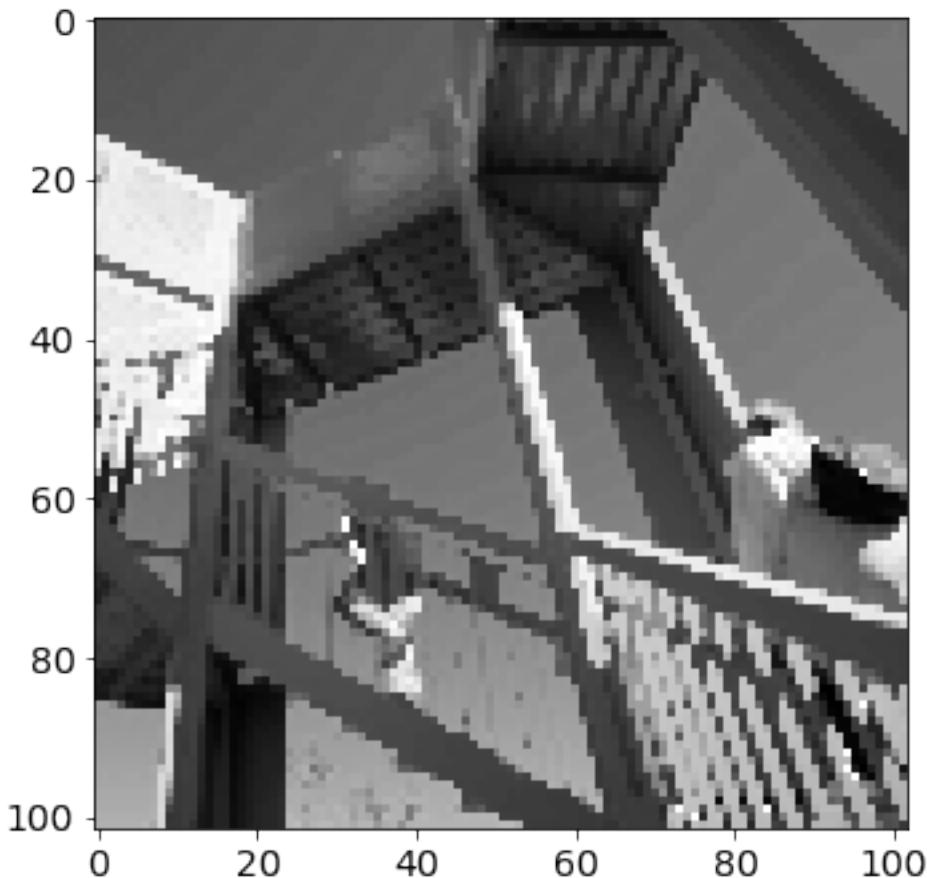
```
In [60]: model = Sequential()
         model.add(MaxPooling2D(pool_size=(5, 5),
                               input_shape=(512, 512, 1)))
         model.compile('adam', 'mse')
```

and let's apply it to our example image:

```
In [61]: img_pred = model.predict(img_tensor)[0, :, :, 0]
         img_pred.shape
```

Out[61]: (102, 102)

```
In [62]: plt.imshow(img_pred, cmap='gray');
```



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:

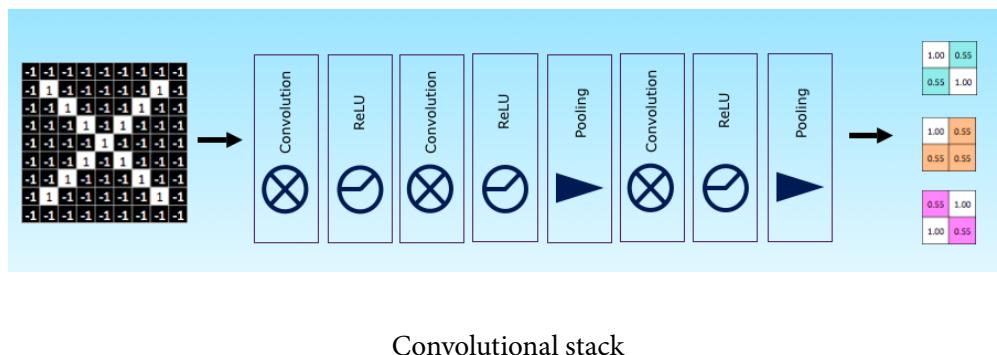
```
In [63]: model = Sequential()
model.add(GlobalMaxPooling2D(input_shape=(512, 512, 1)))
model.compile('adam', 'mse')
```

```
In [64]: img_pred_tensor = model.predict(img_tensor)
          img_pred_tensor.shape
```

Out[64]: (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 a feature extraction pipeline that will gradually transform an image into a tensor with more channels and less pixels:



Convolutional stack

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.

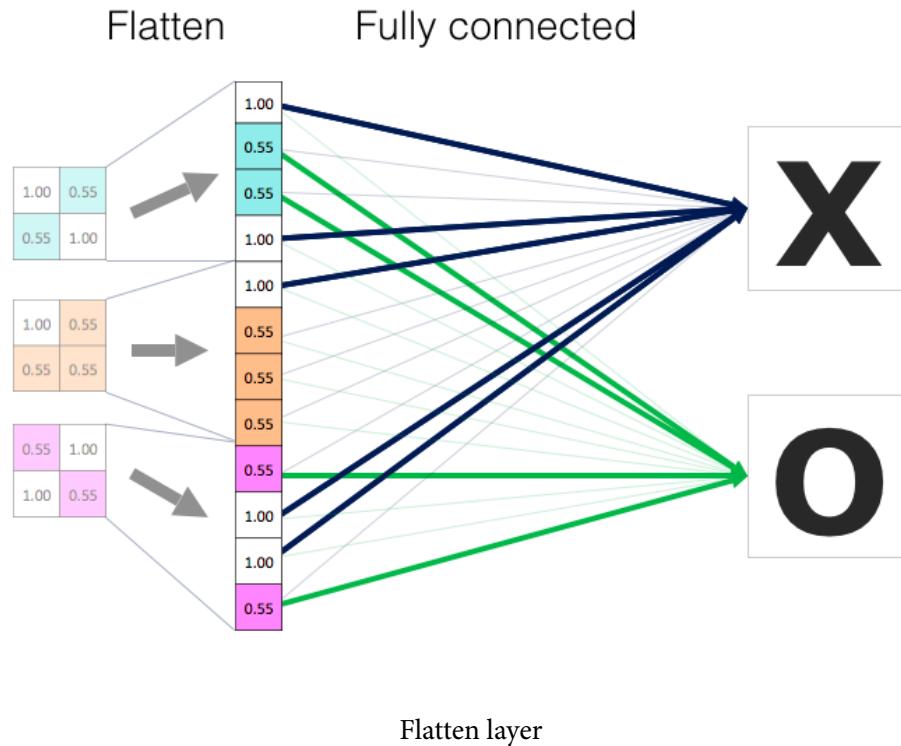
In practice, early layers will specialize to recognize contrast lines in different orientations, while deeper layers will combine those contrast lines to recognize parts of objects. The typical example of this is the face recognition task where middle layers recognize facial features like eyes, noses 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.

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`.

```
In [65]: X_train_t = X_train_sc.reshape(-1, 28, 28, 1)
        X_test_t = X_test_sc.reshape(-1, 28, 28, 1)
```

```
In [66]: X_train_t.shape
```

```
Out[66]: (60000, 28, 28, 1)
```

Then we import the Flatten and Activation layers:

```
In [67]: 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:

```
In [68]: model = Sequential()
```

```
model.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1),
                 kernel_initializer='normal'))
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
<hr/>		
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 32)	0
<hr/>		
activation_1 (Activation)	(None, 13, 13, 32)	0
<hr/>		
flatten_1 (Flatten)	(None, 5408)	0
<hr/>		
dense_6 (Dense)	(None, 64)	346176
<hr/>		
dense_7 (Dense)	(None, 10)	650
<hr/>		
Total params: 347,146		
Trainable params: 347,146		
Non-trainable params: 0		
<hr/>		

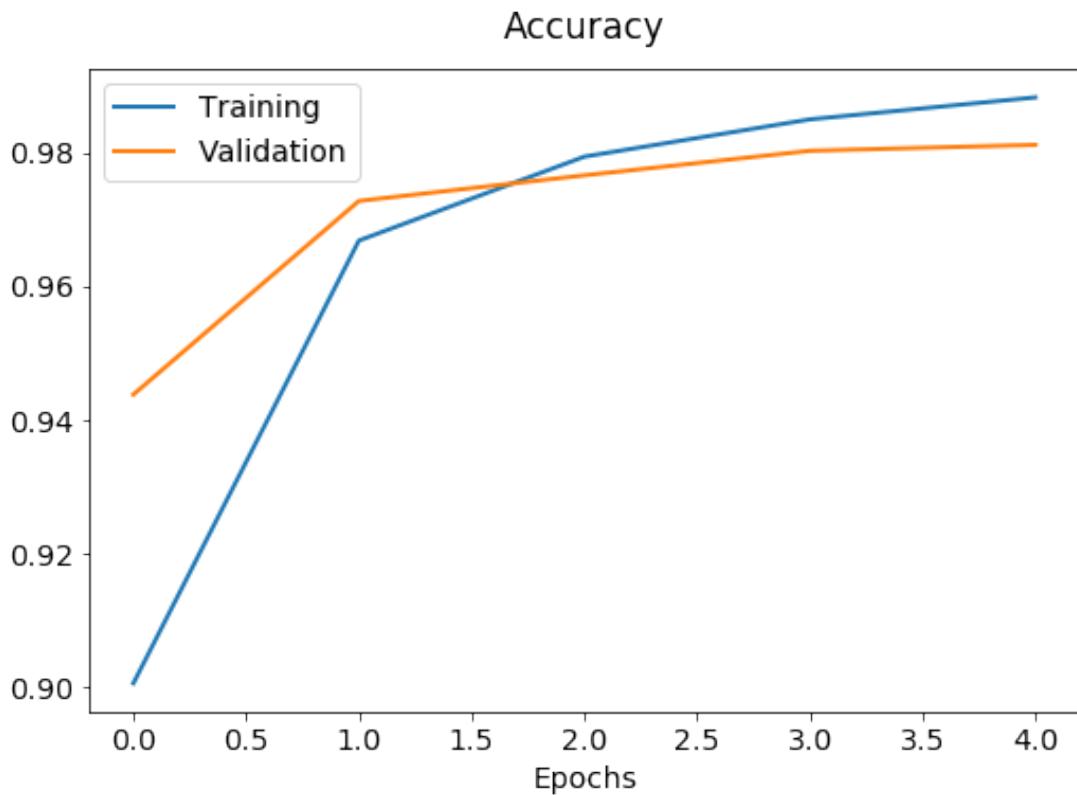
This model has 300k parameters, that's almost half of the the fully connected model we designed at the beginning of this chapter. Let's train it for 5 epochs. Notice that we pass the tensor data we created above:

```
In [69]: 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 45us/step - loss: 0.3335 -
acc: 0.9007 - val_loss: 0.1879 - val_acc: 0.9438
Epoch 2/5
42000/42000 [=====] - 2s 38us/step - loss: 0.1133 -
acc: 0.9669 - val_loss: 0.0937 - val_acc: 0.9728
Epoch 3/5
42000/42000 [=====] - 2s 38us/step - loss: 0.0695 -
acc: 0.9794 - val_loss: 0.0815 - val_acc: 0.9766
Epoch 4/5
42000/42000 [=====] - 2s 38us/step - loss: 0.0501 -
acc: 0.9850 - val_loss: 0.0673 - val_acc: 0.9803
Epoch 5/5
42000/42000 [=====] - 2s 38us/step - loss: 0.0385 -
acc: 0.9883 - val_loss: 0.0616 - val_acc: 0.9812
```

Like before, we can display the training history:

```
In [70]: plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
plt.title('Accuracy')
plt.xlabel('Epochs');
```



and compare the accuracy on train and test sets:

```
In [71]: 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}".format(train_acc))
print("Test accuracy: {:.4f}".format(test_acc))
```

```
Train accuracy: 0.9889
Test accuracy: 0.9835
```

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

In [72]: `from keras.datasets import cifar10`

# 7

## 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).

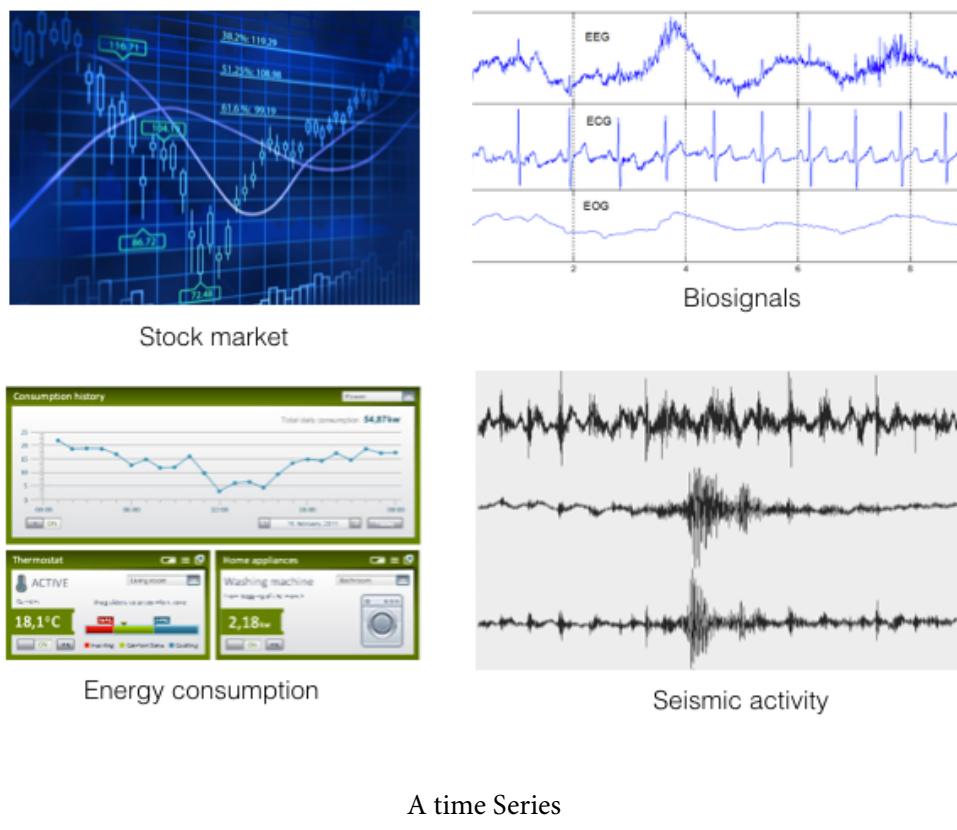
This chapter contains both practical and theoretical parts with some math. Like we did in chapter 5, let us first tell you: **you don't NEED to read the math in this chapter**. This book is meant for the developer and practitioner that is interested in applying Neural Networks to solve great problems. 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.

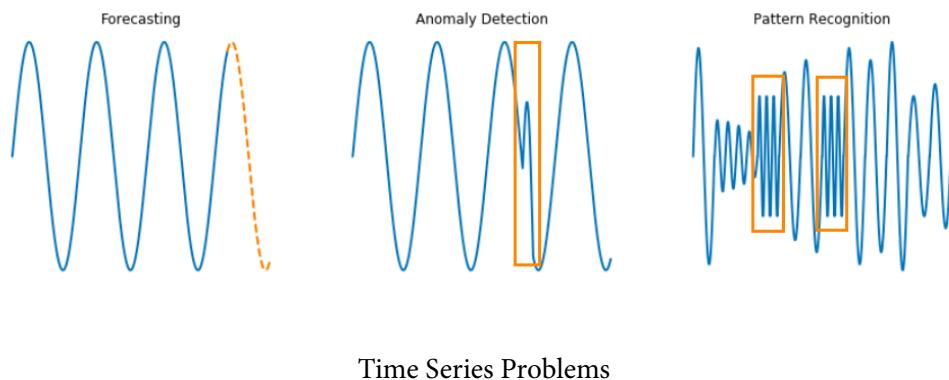
A time series is an ordered sequence of data points, and it can be univariate or multivariate.

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.



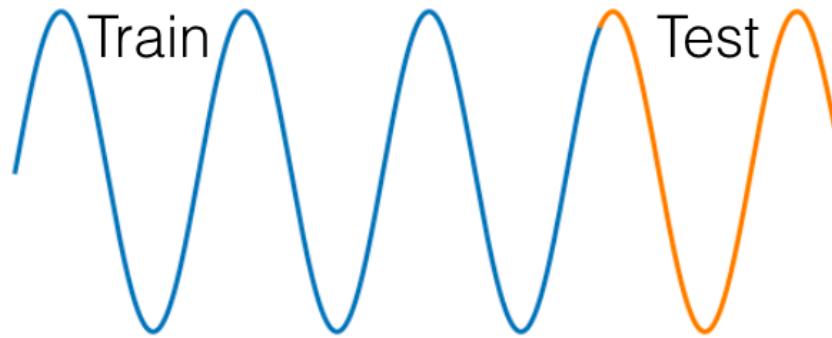
Time Series Problems

**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.

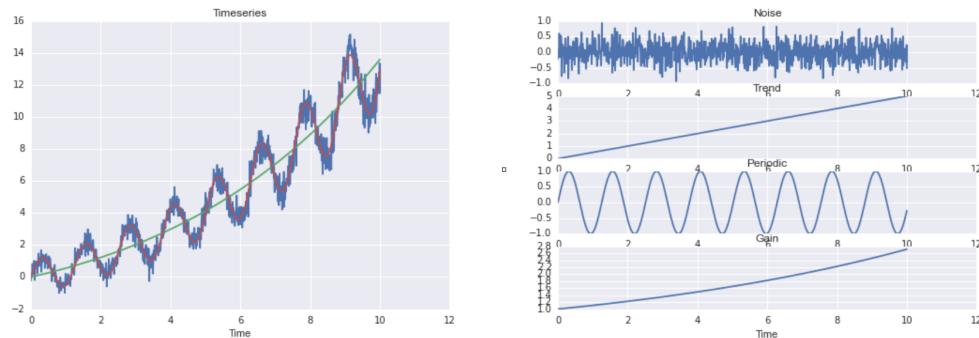
**Pattern recognition** is classification on time series, identifying recurring patterns.

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.



Train/Test approach for a time series problem

Also, sometimes a trend or a seasonal pattern is clearly distinguishable.



Trend and seasonality

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 common files:

```
In [1]: with open('common.py') as fin:
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:
    exec(fin.read())
```

The file `sequence_classification.csv.bz2` contains a set of 4000 curves. Let's load it and look at a few rows and columns:

```
In [3]: fname = '../data/sequence_classification.csv.bz2'
df = pd.read_csv(fname, compression='bz2')
df.iloc[0:5, 0:5]
```

Out [3] :

	anomaly	t_0	t_1	t_2	t_3
0	False	1.000000	0.974399	0.939818	0.906015
1	True	0.626815	0.665145	0.669603	0.693649
2	False	0.983751	0.944806	0.999909	0.975756
3	True	0.977806	1.000000	0.975431	0.966523
4	False	0.691444	0.710671	0.660787	0.690993

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.

```
In [4]: 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).

```
In [5]: df['anomaly'].value_counts()
```

```
Out[5]:
```

---

anomaly
True 2000
False 2000

---

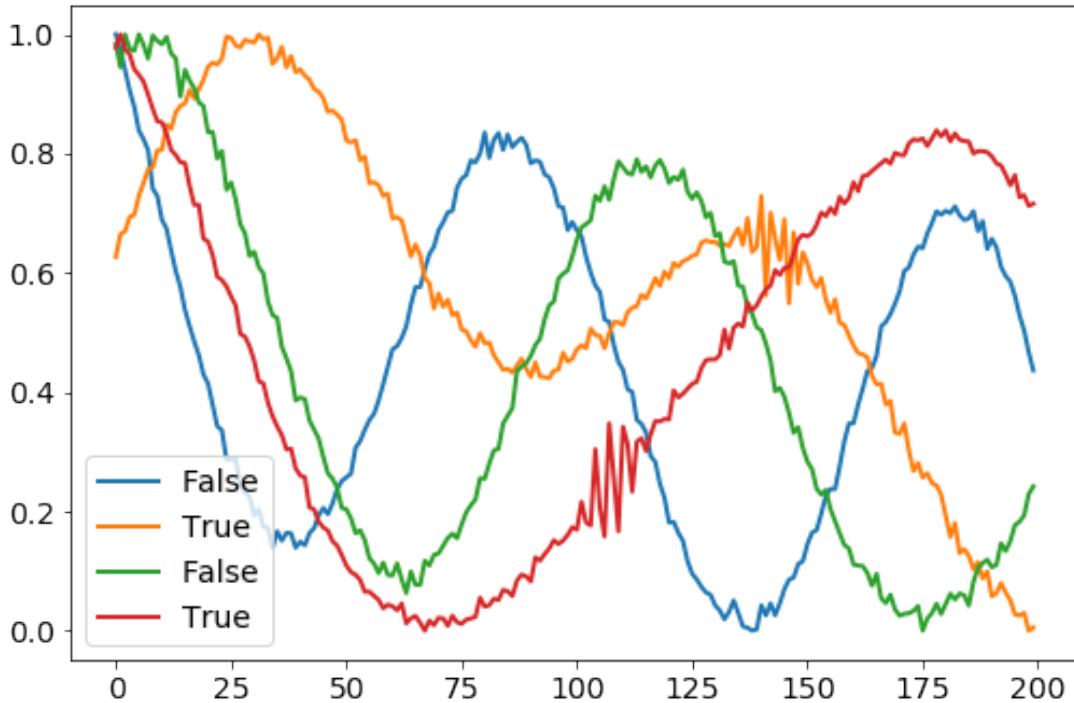
As we can see, 2000 curves present anomalies, while the other 2000 do not. Let's create the X and y arrays and plot the first 4 curves.

```
In [6]: X = df.drop("anomaly", axis="columns").values
y = df["anomaly"].values
```

Now let's plot these curves separated by the anomaly values.

```
In [7]: plt.plot(X[:4].transpose())
plt.legend(y[:4])
plt.title("Curves with Anomalies");
```

### 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:

```
In [8]: from sklearn.model_selection import train_test_split
```

Now let's split the data into the training and test sets:

```
In [9]: 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:

```
In [10]: from keras.models import Sequential
        from keras.layers import Dense
        import keras.backend as K
```

Using TensorFlow backend.

Let's also clear the backend of any data it's holding on to with `clear_session()` on the backend:

```
In [11]: 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:

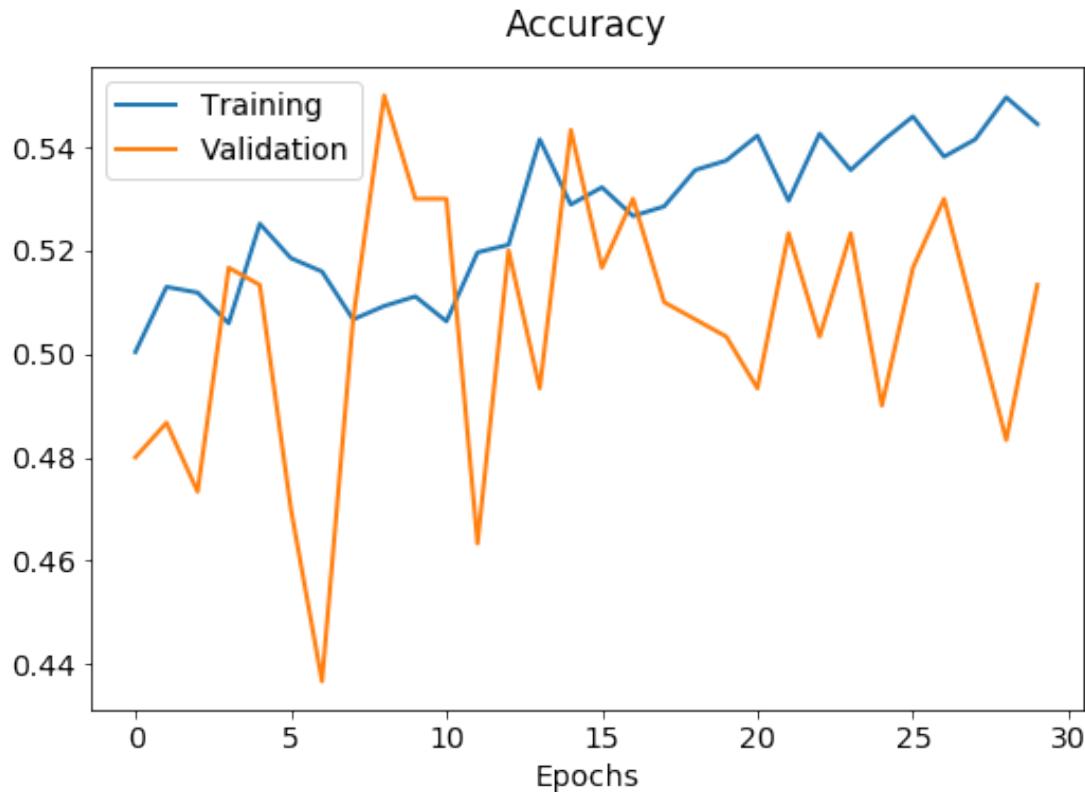
```
In [12]: model = Sequential()
        model.add(Dense(100, input_dim=200, activation='relu'))
        model.add(Dense(50, activation='relu'))
        model.add(Dense(20, activation='relu'))
        model.add(Dense(1, activation='sigmoid'))
        model.compile(optimizer='adam',
                      loss='binary_crossentropy',
                      metrics=['accuracy'])
```

Next, let's train the model to fit against our training set.

```
In [13]: 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.

```
In [14]: plt.plot(h.history['acc'])  
plt.plot(h.history['val_acc'])  
plt.legend(['Training', 'Validation'])  
plt.title('Accuracy')  
plt.xlabel('Epochs');
```



```
In [15]: acc_ = model.evaluate(X_test, y_test, verbose=0)[1]  
print("Test Accuracy: {:.3f}".format(acc_))
```

Test Accuracy: 0.539

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`.

```
In [16]: 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()
```

Out[16] :

	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:

```
In [17]: (eng_f_train, eng_f_test,
      y_train, y_test) = train_test_split(eng_f.values, y,
                                         test_size=0.25,
                                         random_state=0)
```

Let's clear out the backend for any memory we've already used:

```
In [18]: 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.

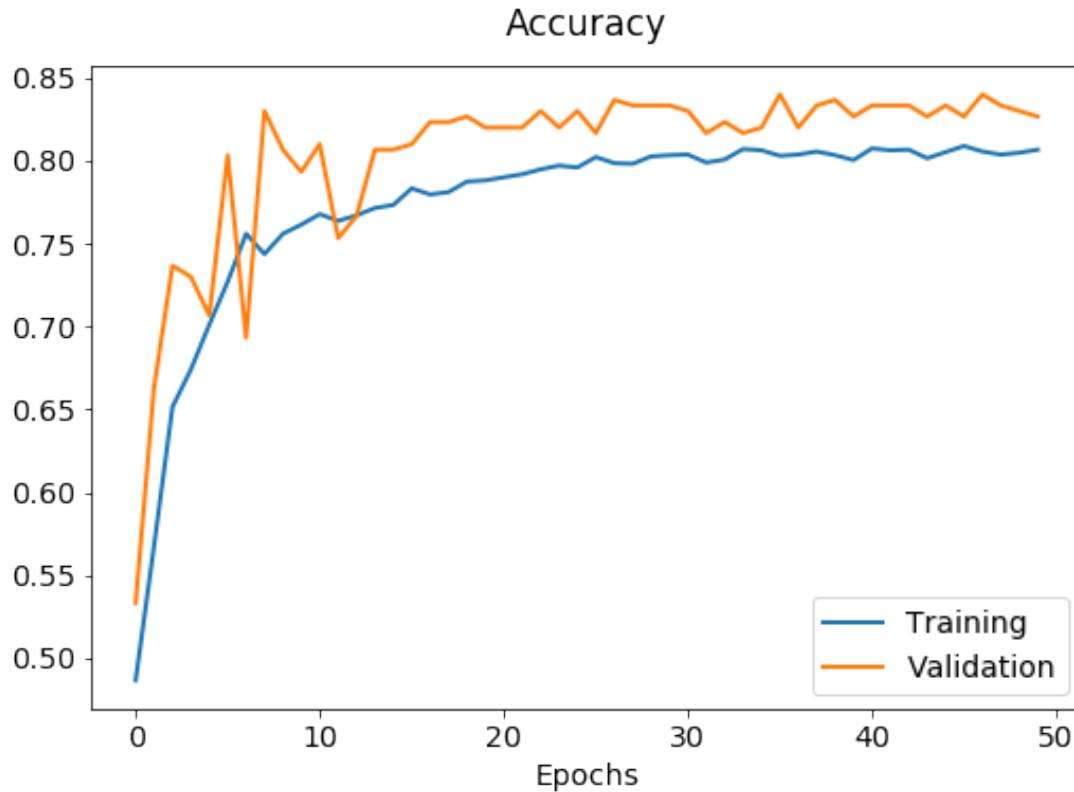
```
In [19]: model = Sequential()
model.add(Dense(30, input_dim=2, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Now let's train our model on our 2 engineered features:

```
In [20]: 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:

```
In [21]: plt.plot(h.history['acc'])
plt.plot(h.history['val_acc'])
plt.legend(['Training', 'Validation'])
plt.title('Accuracy')
plt.xlabel('Epochs');
```



```
In [22]: acc_ = model.evaluate(eng_f_test, y_test, verbose=0)[1]
      print("Test Accuracy: {:.3f}".format(acc_))
```

```
Test Accuracy: 0.817
```

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 nonlinear 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:

```
In [23]: from keras.layers import Conv1D, MaxPool1D
         from keras.layers import Flatten, Activation
```

Next, let's clear out the backend memory, just in case:

```
In [24]: K.clear_session()
```

Next, let's build the model with our layers, considering again the 200 points as input:

```
In [25]: 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(optimizer='adam',
              loss='binary_crossentropy',
              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.

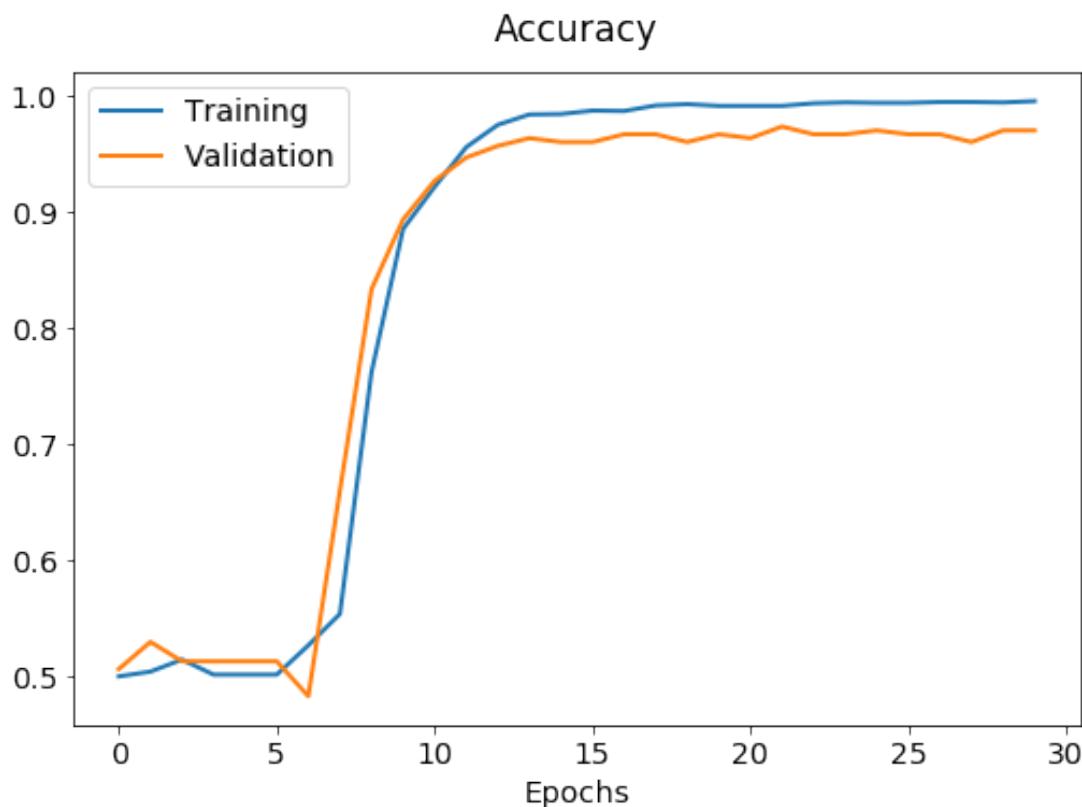
```
In [26]: X_train_t = X_train[:, :, None]  
X_test_t = X_test[:, :, None]
```

Let's train our model over 30 epochs:

```
In [27]: 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:

```
In [28]: plt.plot(h.history['acc'])  
plt.plot(h.history['val_acc'])  
plt.legend(['Training', 'Validation'])  
plt.title('Accuracy')  
plt.xlabel('Epochs');
```



```
In [29]: acc_ = model.evaluate(X_test_t, y_test, verbose=0)[1]
      print("Test Accuracy: {:.3f}".format(acc_))
```

Test Accuracy: 0.983

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

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.

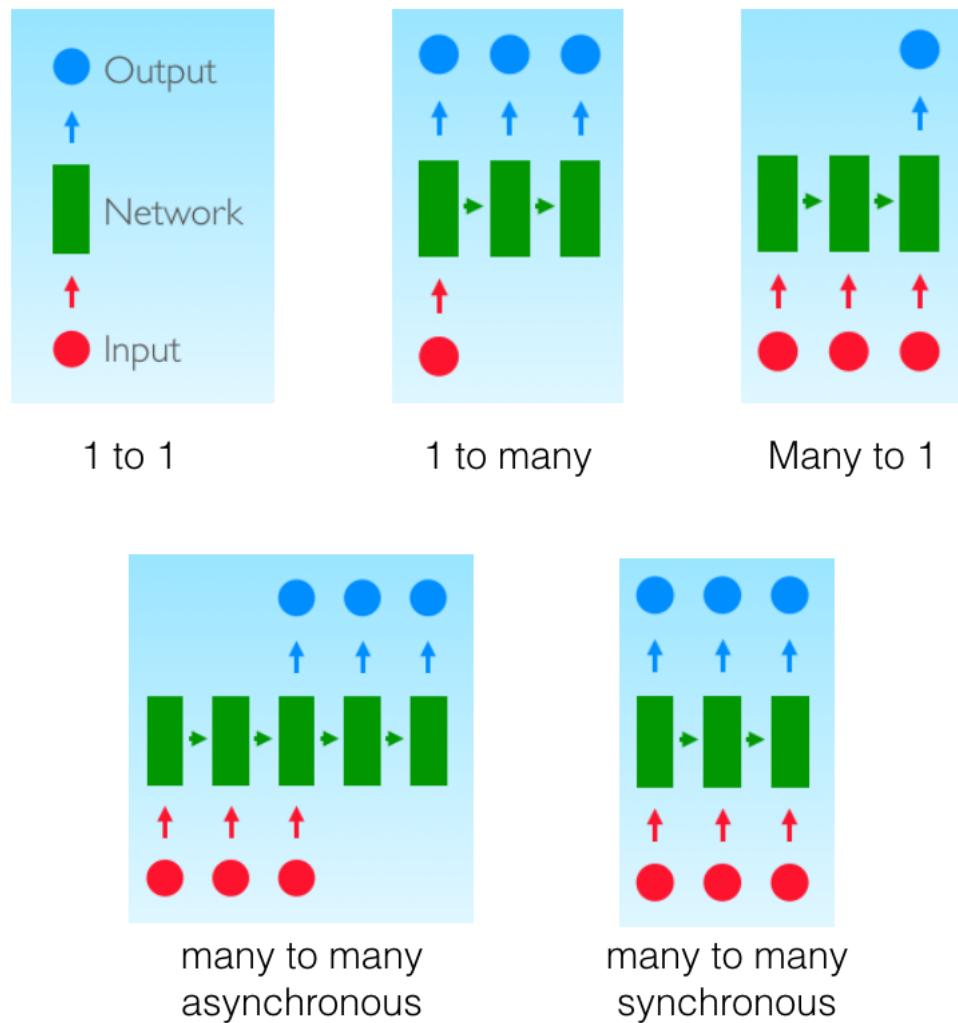
In all of these cases the Machine Learning model learns a stateless function to connect a given input to a given output.

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.



Problems involving sequences

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:

```
In [30]: fname = '../data/ZonalDemands_2003-2016.csv.bz2'
df = pd.read_csv(fname, compression='bz2',
                 engine='python')
```

```
In [31]: df.head(3)
```

Out [31] :

	Date	Hour	Total	Ontario	Northwest	Northeast	Ottawa	East	Toronto	Essa	Bruce	Southwest	Niagara	West	Tot Zones	diff
0	01-May-03	1	13702	809	1284	965	765	4422	622	41	2729	617	1611	13865	163	
1	01-May-03	2	13578	825	1283	923	752	4340	602	43	2731	615	1564	13678	100	
2	01-May-03	3	13411	834	1277	910	751	4281	591	45	2696	596	1553	13534	123	

```
In [32]: df.tail(3)
```

Out [32] :

	Date	Hour	Total	Ontario	Northwest	Northeast	Ottawa	East	Toronto	Essa	Bruce	Southwest	Niagara	West	Tot Zones	diff
119853	2016/12/31	22	15195	495	1476	1051	1203	5665	1045	72	2986	465	1334	15790	595	
119854	2016/12/31	23	14758	495	1476	1051	1203	5665	1045	72	2986	465	1334	15790	1,032	
119855	2016/12/31	24	14153	495	1476	1051	1203	5665	1045	72	2986	465	1334	15790	1,637	

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.

```
In [33]: 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

In [34]: `idx = df.apply(combine_date_hour, axis=1)`

In [35]: `idx.head()`

Out[35] :

	0
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

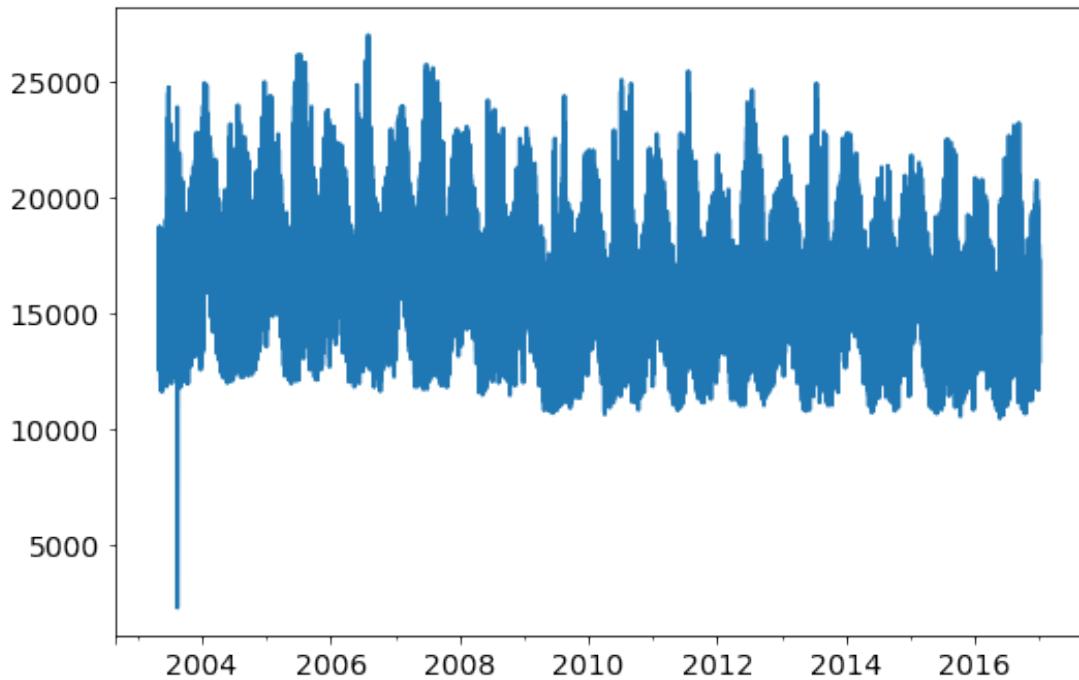
In [36]: `df = df.set_index(idx)`

TIP: the function `set_index()` returns a new DataFrame whose index (row labels) has been set to the 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:

In [37]: `df['Total Ontario'].plot()`

Out[37]: <matplotlib.axes.\_subplots.AxesSubplot at 0x7feb61243898>



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.

```
In [38]: split_date = pd.Timestamp('01-01-2014')
```

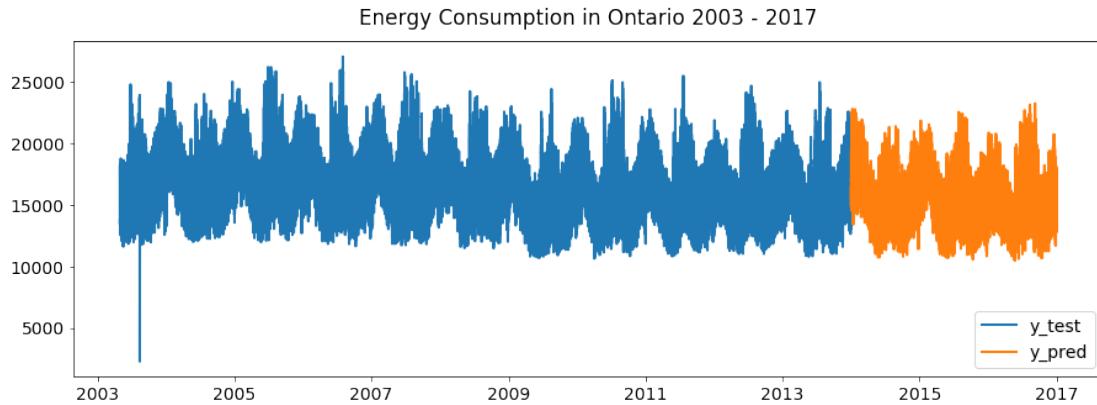
Now we copy the data to a pair of new Pandas data frames that only contain the `Total Ontario` data up to the split date (train) and after the split date (test).

```
In [39]: 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:

```
In [40]: plt.figure(figsize=(15,5))
plt.plot(train, label='y_test')
plt.plot(test, label='y_pred')
plt.legend()
plt.title("Energy Consumption in Ontario 2003 - 2017");
```



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.

```
In [41]: 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.

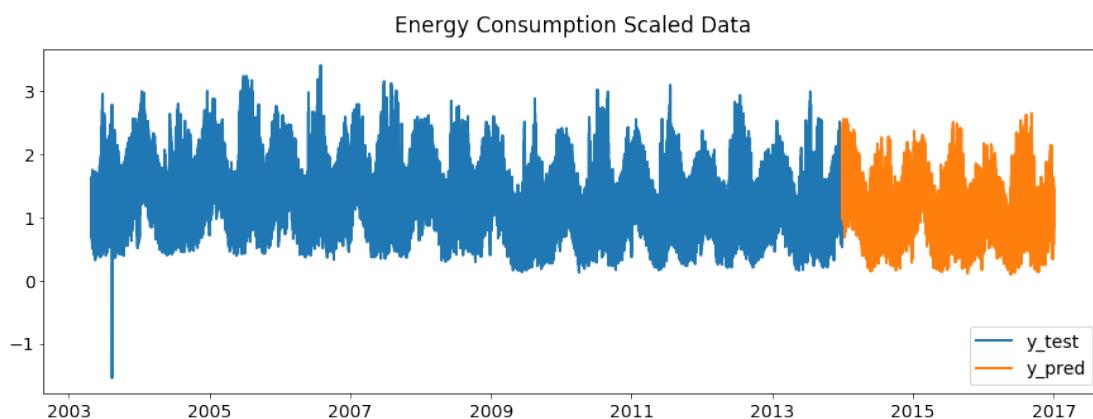
In [42]: `train_sc[:4]`

Out [42] :

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.

```
In [43]: plt.figure(figsize=(15,5))
    plt.plot(train_sc, label='y_test')
    plt.plot(test_sc, label='y_pred')
    plt.legend()
    plt.title("Energy Consumption Scaled Data");
```



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.

```
In [44]: 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:

```
In [45]: K.clear_session()
```

Next, let's build our model:

```
In [46]: 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(optimizer='adam',
              loss='mean_squared_error')
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 24)	48
dense_2 (Dense)	(None, 12)	300
dense_3 (Dense)	(None, 6)	78

```
-----  
dense_4 (Dense)           (None, 1)          7  
=====  
Total params: 433  
Trainable params: 433  
Non-trainable params: 0  
-----
```

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!).

In [47]: `from keras.callbacks import EarlyStopping`

In [48]: `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 1000 points so we can choose large batches.

In [49]: `model.fit(X_train, y_train, epochs=200, batch_size=512, verbose=0, callbacks=[early_stop])`

Epoch 00006: early stopping

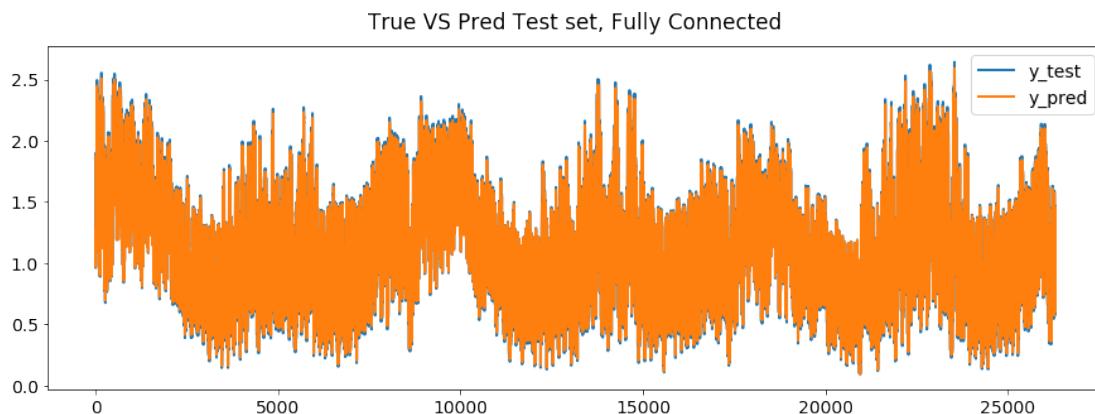
```
Out[49]: <keras.callbacks.History at 0x7fed3649dd0>
```

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`.

```
In [50]: y_pred = model.predict(X_test)
```

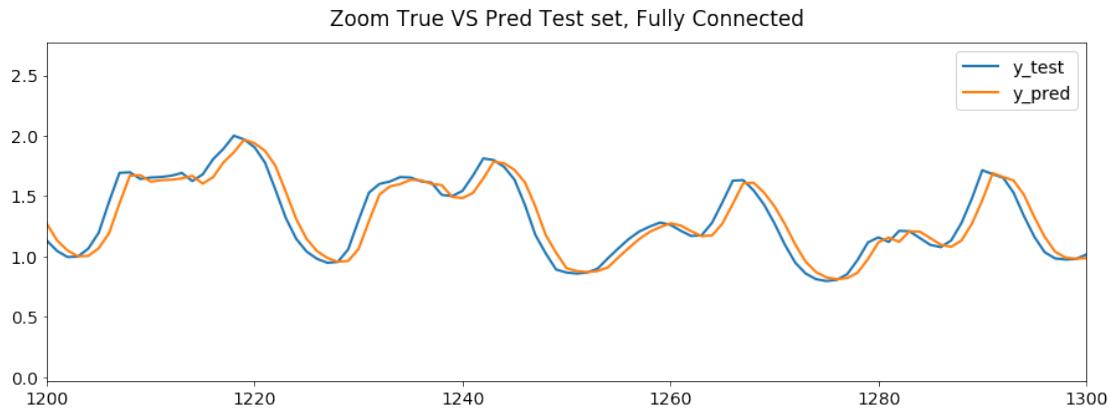
Let's visually compare test values and predictions:

```
In [51]: plt.figure(figsize=(15,5))
plt.plot(y_test, label='y_test')
plt.plot(y_pred, label='y_pred')
plt.legend()
plt.title("True VS Pred Test set, Fully Connected");
```



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.

```
In [52]: 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.title("Zoom True VS Pred Test set, Fully Connected");
```



### 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](#)

```
In [53]: from sklearn.metrics import mean_squared_error
        from sklearn.metrics import r2_score
```

```
In [54]: 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.

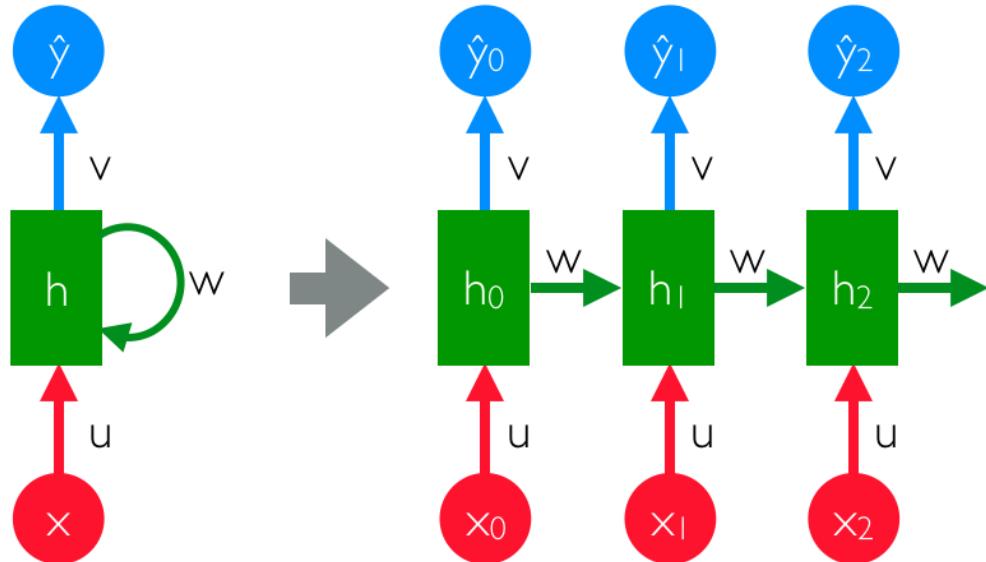
Let's see if a recurrent network improves the situation.

## 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.



Unrolling time for a vanilla RNN

Ignoring the output of the network for the time being, let's focus on the recurrent aspect. The network is

recurrent because its internal state  $h$  at time  $t$  is obtained by mixing current input  $x_t$  with the previous value of the internal state  $h_{t-1}$ :

$$h_t = \tanh(w h_{t-1} + u x_t) \quad (7.1)$$

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:

$$y_t = (1 - \alpha) y_{t-1} + \alpha x_t \quad (7.2)$$

EWMA smooths a signal given by a sequence of data reducing its fluctuations.

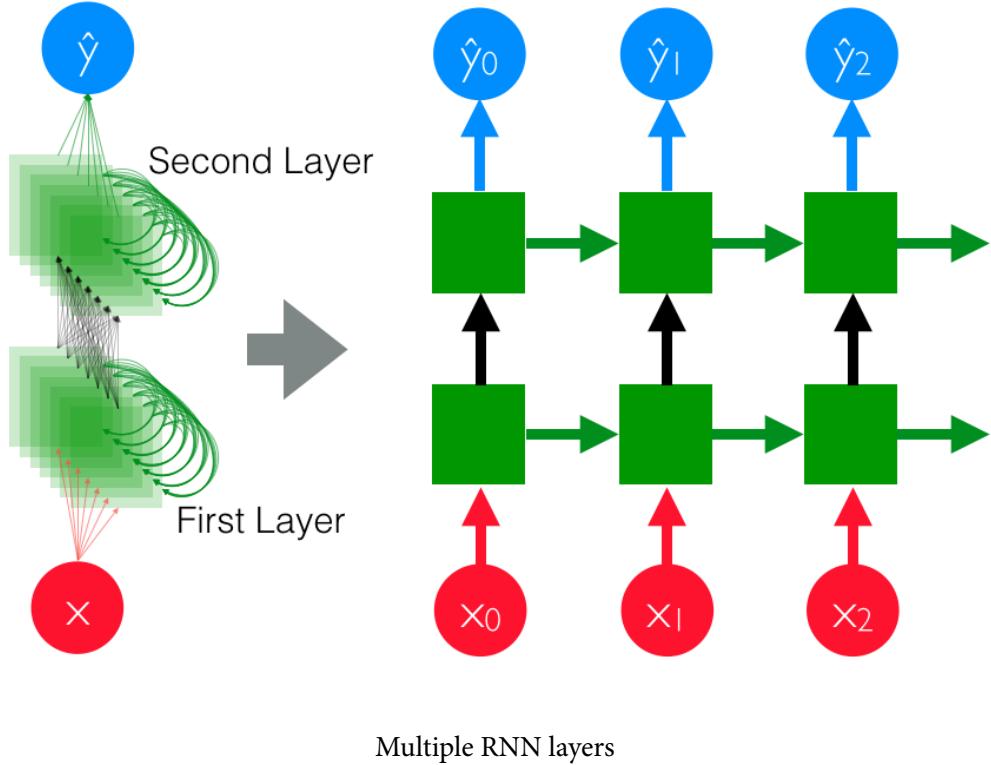
It is not exactly the same, because there is a  $\tanh$  and the two weights are independent but it does look similar: it's a linear mixing of the past output with the present input, followed by a nonlinear activation function.

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.

```
In [55]: 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:

```
In [56]: X_train.shape
```

```
Out[56]: (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:

```
In [57]: 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`:

```
In [58]: X_train_t.shape
```

```
Out[58]: (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.

```
In [59]: from keras.optimizers import Adam, RMSprop
```

Let's clear the backend memory again:

```
In [60]: 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:

```
In [61]: model = Sequential()
model.add(SimpleRNN(6,
                    batch_input_shape=(1, 1, 1),
                    kernel_initializer='ones',
                    stateful=True))
model.add(Dense(1))

model.compile(optimizer=Adam(lr=0.0005),
              loss='mean_squared_error')
```

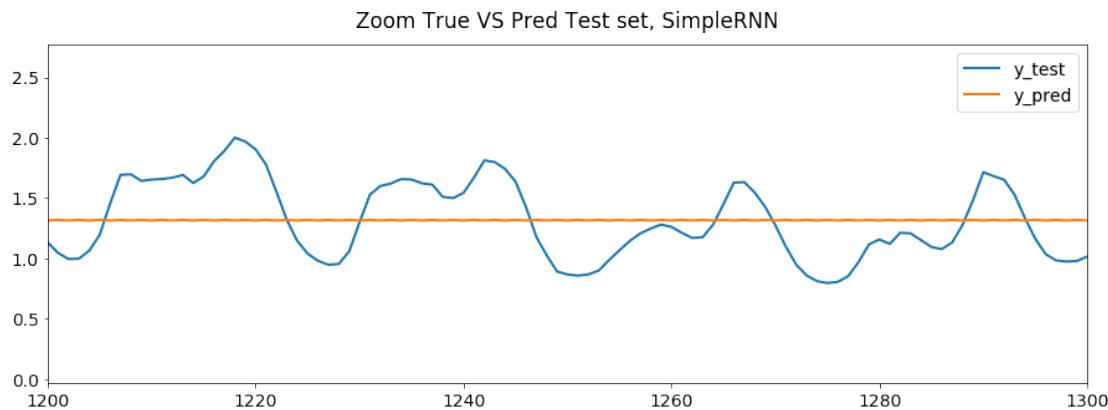
Now we can fit the data. Since we are maintaining states between point, we shall pass the data in order using the `shuffle=False` flag and `batch_size=1`. Also, we run the training for a single epoch. In our experiments this should be sufficient to get decent results:

```
In [62]: model.fit(X_train_t, y_train_t,
                  epochs=1,
                  batch_size=1,
                  verbose=1,
                  shuffle=False);
```

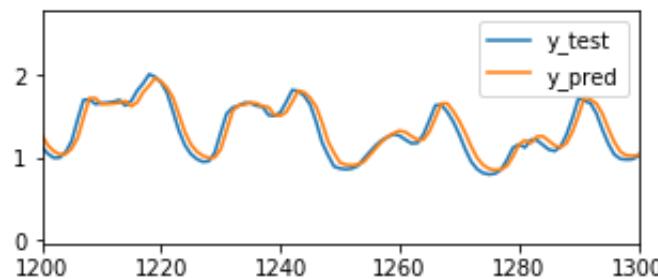
```
Epoch 1/1
93551/93551 [=====] - 243s 3ms/step - loss: 0.2061
```

Let's plot a small part of our predictive model to compare train and test.

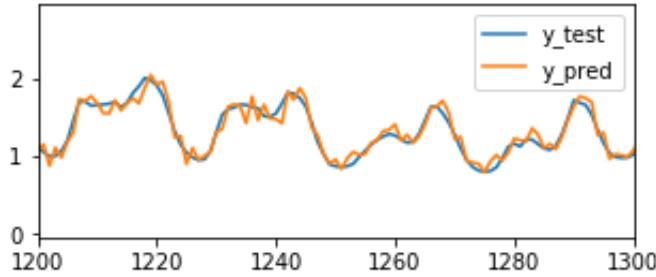
```
In [63]: y_pred = model.predict(X_test_t, batch_size=1)
plt.figure(figsize=(15,5))
plt.plot(y_test_t, label='y_test')
plt.plot(y_pred, label='y_pred')
plt.legend()
plt.xlim(1200,1300)
plt.title("Zoom True VS Pred Test set, SimpleRNN");
```



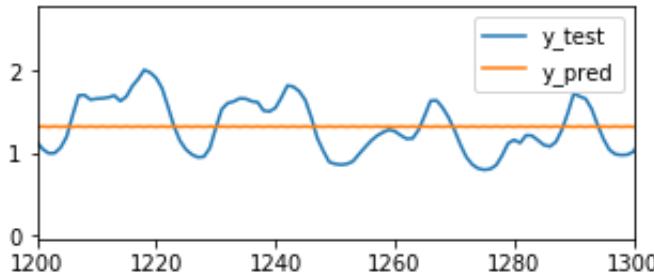
Notice that despite our initialization, the model converges to different solutions at each training run. - Sometimes you will get a graph that looks very similar to the Fully Connected result, with no predictivity at all:



- 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:

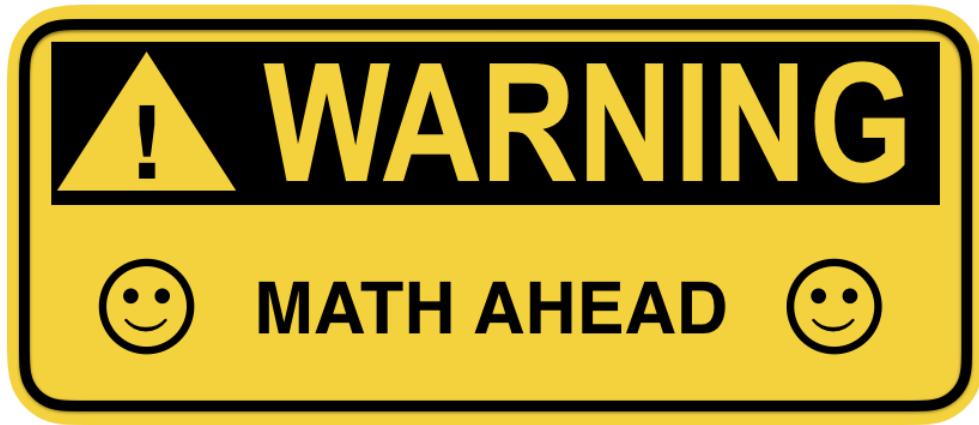
```
In [64]: mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("MSE: {:.3f}".format(mse))
print("R2: {:.3f}".format(r2))
```

```
MSE: 0.252
R2: -0.13
```

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.

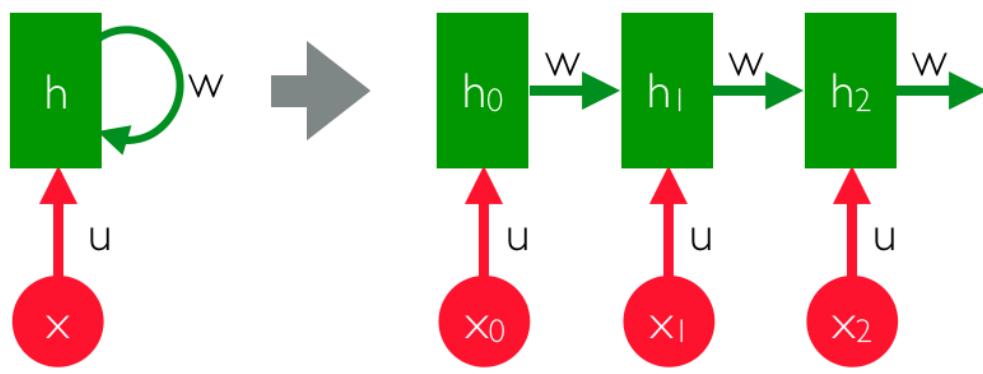


## Recurrent Neural Network Maths

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.



Encoder Network

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:

$$z_t = w h_{t-1} + u x_t \quad (7.3)$$

$$h_t = \phi(z_t) \quad (7.4)$$

$$(7.5)$$

where we substituted the tanh activation function to a generic activation  $\phi$ .

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:

$$\bar{h}_t = \bar{z}_{t+1} w \quad (7.6)$$

$$\bar{z}_t = \bar{h}_t \phi'(z_t) \quad (7.7)$$

$$(7.8)$$

Let's focus our attention on  $\bar{h}_t$ , and let's propagate back all the way to  $\bar{h}_o$ :

$$\bar{h}_o = w \bar{z}_1 \quad (7.9)$$

$$= w \bar{h}_1 \phi'(z_1) \quad (7.10)$$

$$= w^2 \bar{h}_2 \phi'(z_1) \phi'(z_2) \quad (7.11)$$

$$\dots = w^T \bar{h}_T \phi'(z_1) \phi'(z_2) \dots \phi'(z_T) \quad (7.12)$$

$$(7.13)$$

Now remembering the definition of  $\bar{h} = \frac{\partial L}{\partial h}$  we can write:

$$\bar{h}_o = \bar{h}_T \frac{\partial h_T}{\partial h_o} \quad (7.14)$$

which implies:

$$\frac{\partial h_T}{\partial h_o} = w^T \phi'(z_1) \phi'(z_2) \dots \phi'(z_T) \quad (7.15)$$

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:

```
In [65]: 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.

```
In [66]: 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')

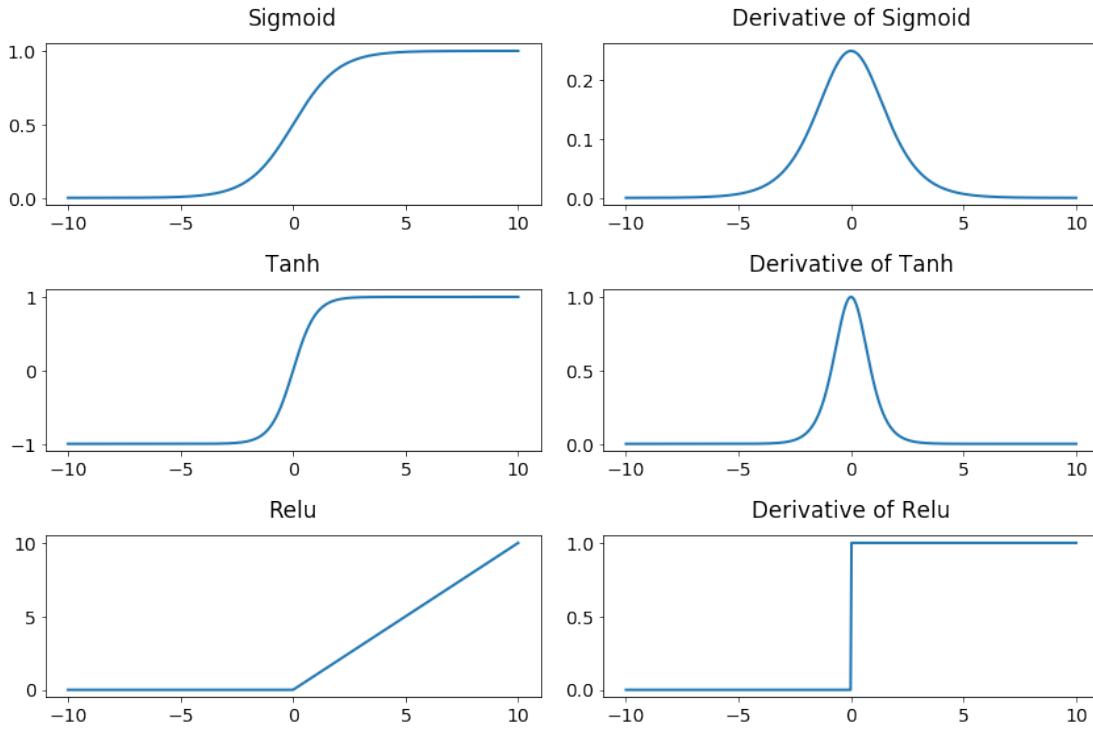
plt.subplot(323)
plt.plot(x, np.tanh(x))
plt.title('Tanh')

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:

$$\frac{\partial h_T}{\partial h_o} = w^T \phi'(z_1) \phi'(z_2) \dots \phi'(z_T) \leq w^T \quad (7.16)$$

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.

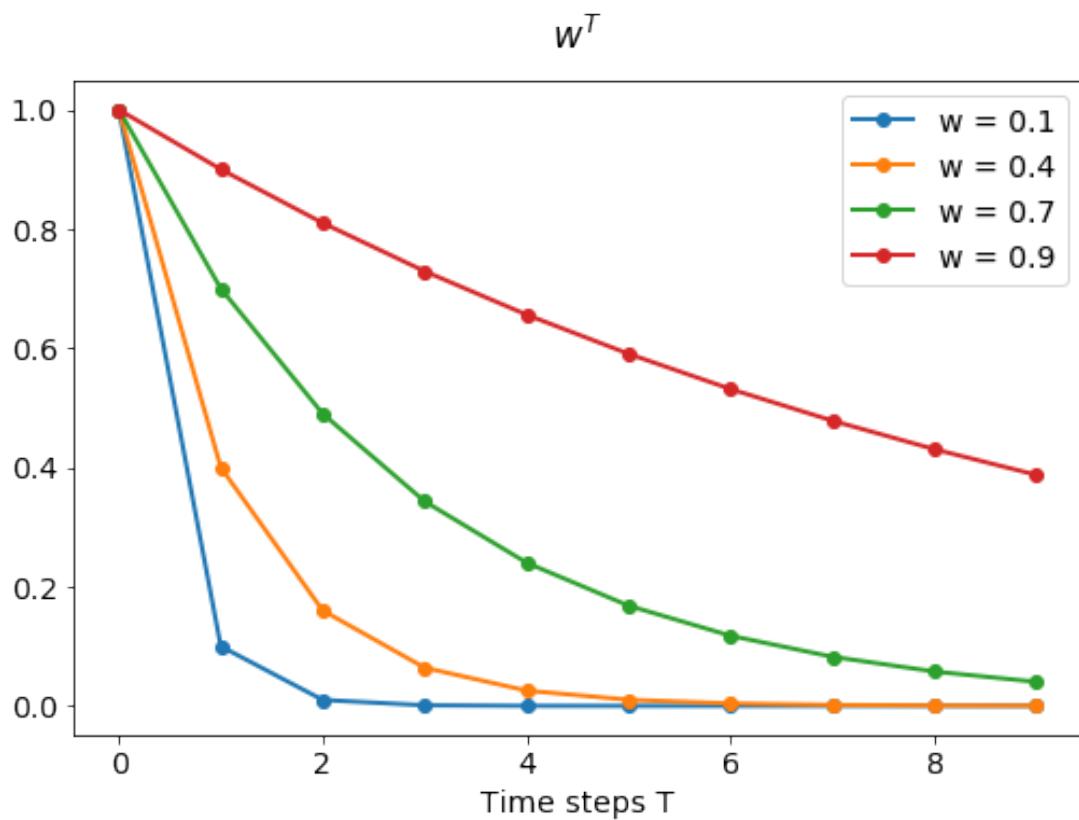
Let's take a peek at what this looks like visually. First, let's create a decay function that we'll use to create our plots.

```
In [67]: 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  $w$ :

```
In [68]: 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]);
```



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  $x_t$
- the previous output at time  $t - 1$  which we indicate as  $h_{t-1}$ .

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.

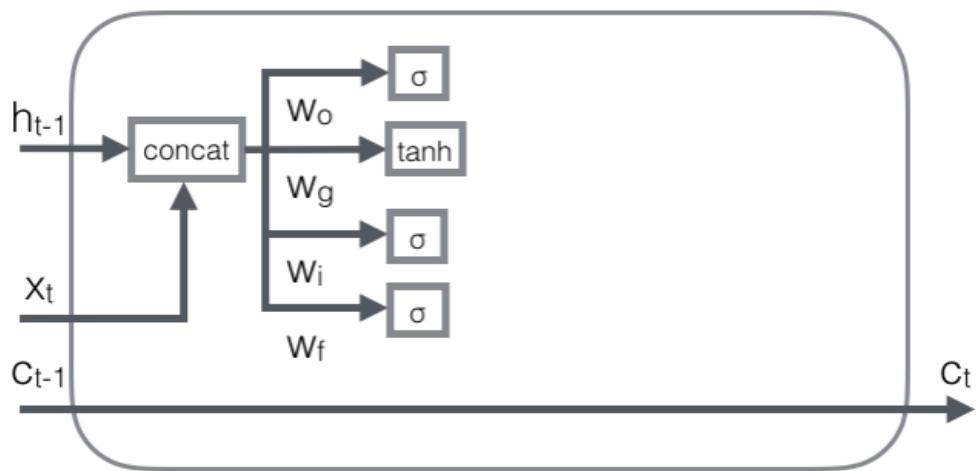
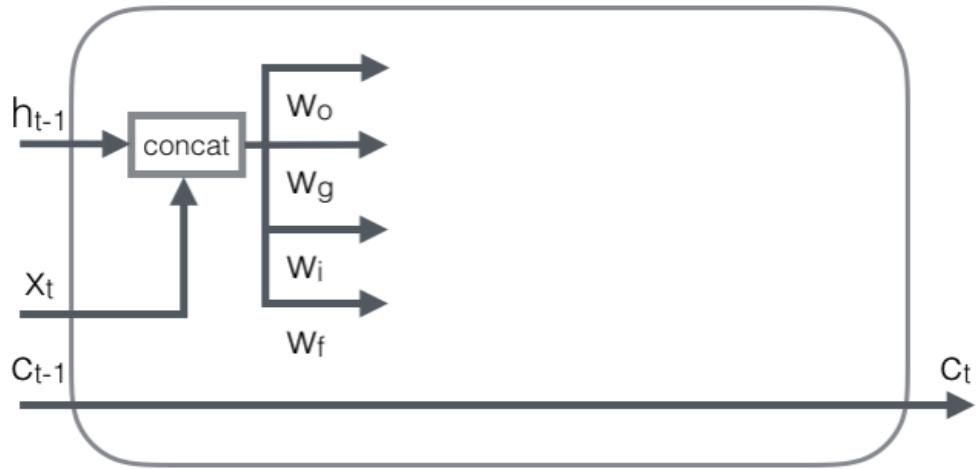
**TIP:** You may be wondering why 4 and not 3 or 5. The reason is simple: one branch is the one that will process the data similarly to the Vanilla RNN, i.e. it will take the past and the present, weight them and send them through a  $\tanh$  activation function. The other three branches will control 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.

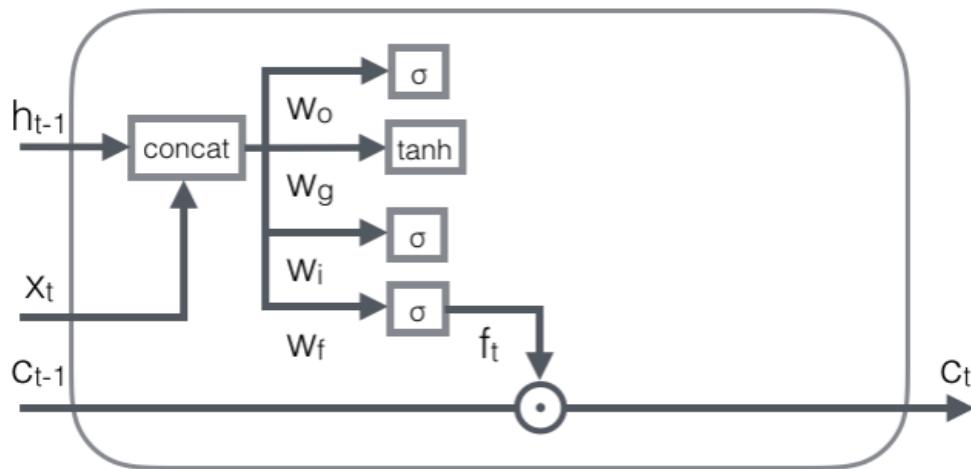
Notice that the weights here are actually weight matrices. The number of rows in the weight matrix corresponds to the number of features, while the number of columns corresponds to the number of output features, i.e. the number of nodes in the LSTM layer.

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**.

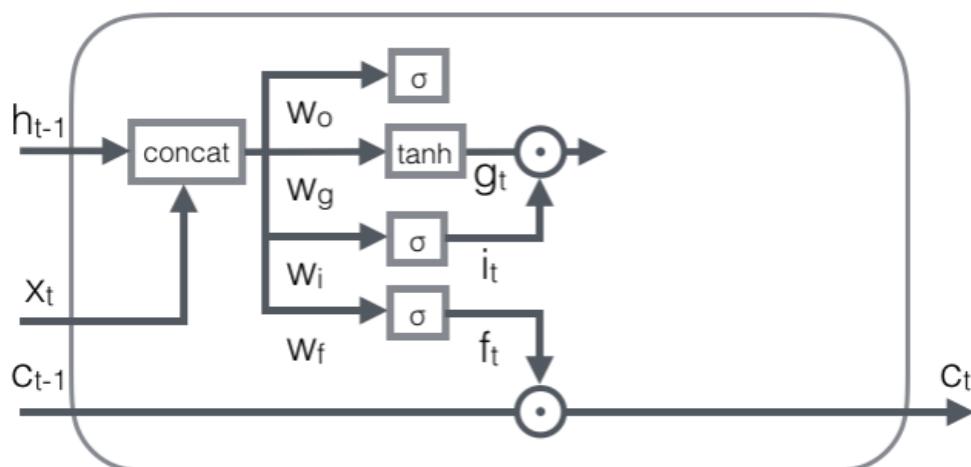




The role of the forget gate is to mediate how much of the internal state vector will be kept and passed through to the future times. Since the value of this gate comes from a dense layer followed by a sigmoid, the LSTM node is learning which fraction of the past data to retain and which fraction to forget.

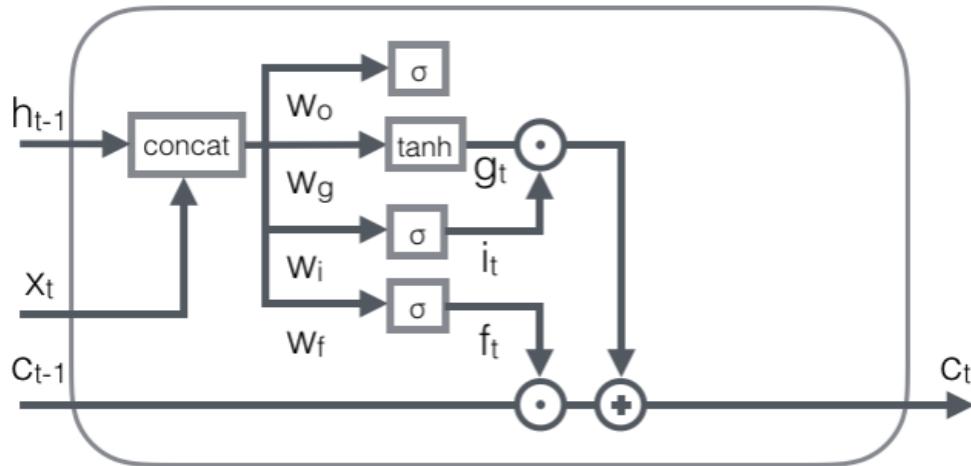
Notice that the  $\odot$  operator implies we are multiplying  $f_t$  and  $c_t$  elementwise. This fact also means they are vectors of the same length.

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  $g_t$ .



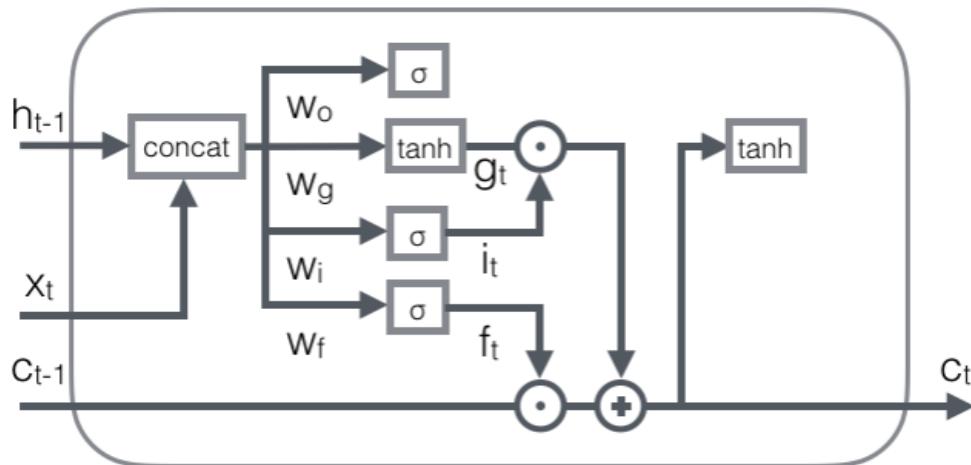
This gating operation is also performed elementwise on  $\mathbf{g}_t$ .

The resulting vector  $\mathbf{i}_t \odot \mathbf{g}_t$  is added to the fraction of internal state that had been retained through the forget gate.



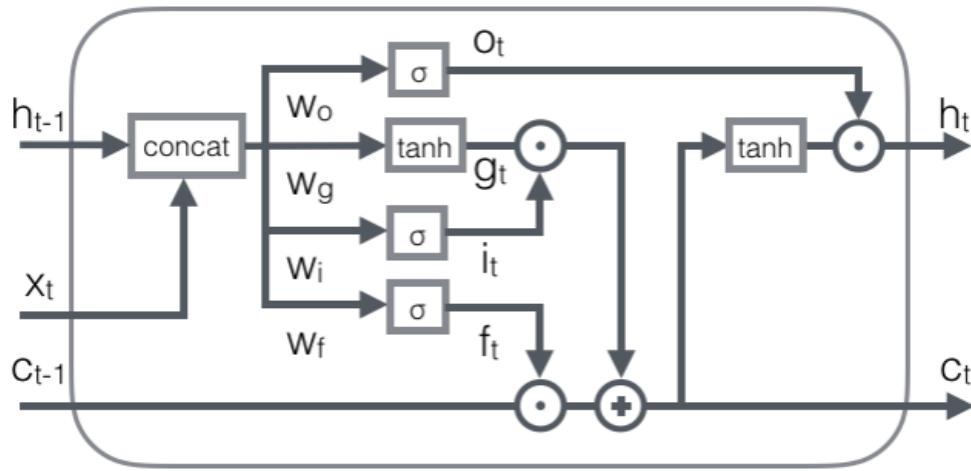
The new internal state  $c_t$  is the result of these two operations: forgetting a bit of the past state and adding some new elements coming from the input and the past output.

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.



The complete LSTM network graph

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!

$$\mathbf{x}_t = [\mathbf{x}_t, \mathbf{h}_{t-1}] \quad (7.17)$$

$$\mathbf{i}_t = \sigma(\mathbf{x}_t \cdot \mathbf{W}_i) \quad (7.18)$$

$$\mathbf{f}_t = \sigma(\mathbf{x}_t \cdot \mathbf{W}_f) \quad (7.19)$$

$$\mathbf{o}_t = \sigma(\mathbf{x}_t \cdot \mathbf{W}_o) \quad (7.20)$$

$$\mathbf{g}_t = \tanh(\mathbf{x}_t \cdot \mathbf{W}_g) \quad (7.21)$$

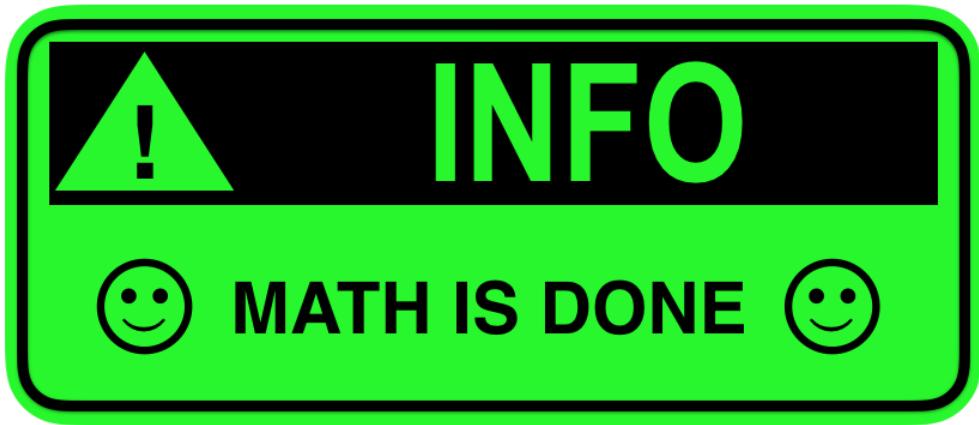
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \quad (7.22)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (7.23)$$

$$(7.24)$$

## 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:

```
In [69]: from keras.layers import LSTM
```

Let's clear the backend memory again to build our model:

```
In [70]: 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.

Like we did above, we will use the Adam optimizer with a small learning rate and initialize the input weights to one:

```
In [71]: 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.

```
In [72]: model.fit(X_train_t, y_train_t,
                  epochs=2,
                  batch_size=1,
                  verbose=1,
                  shuffle=False);
```

```

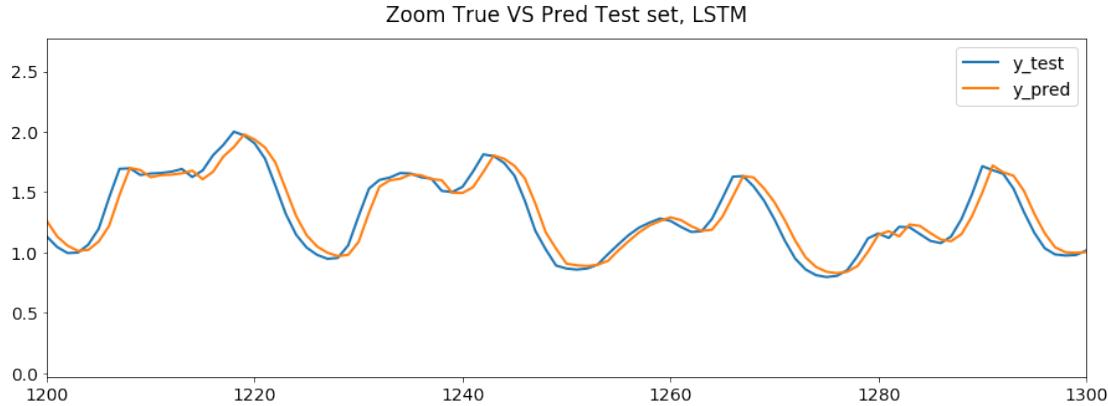
Epoch 1/2
93551/93551 [=====] - 349s 4ms/step - loss: 0.0407
Epoch 2/2
74185/93551 [=====>...] - ETA: 1:12 - loss: 0.0156

```

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:

```
In [73]: 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)
plt.title("Zoom True VS Pred Test set, LSTM");
```



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:

```
In [74]: 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: 0.013
R2: 0.942
```

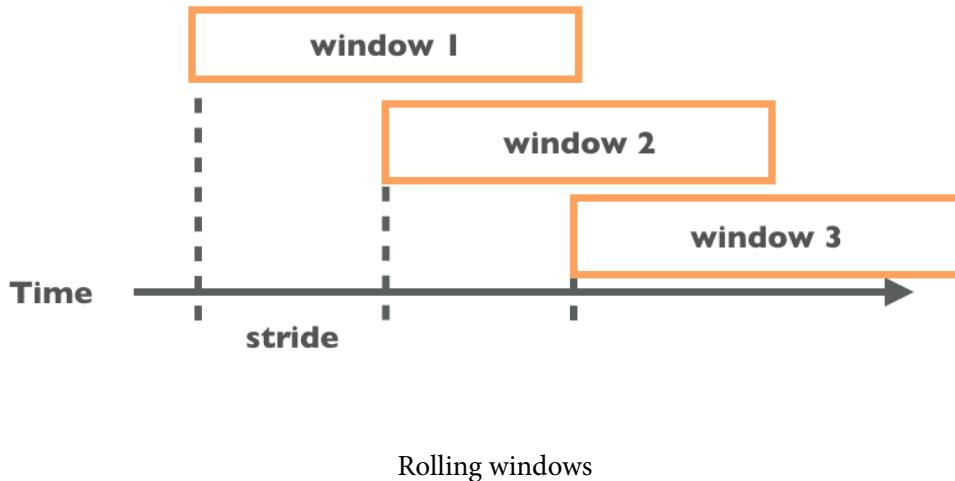
## Improving forecasting

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:



```
In [75]: 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:

```
In [76]: train_sc.head()
```

Out[76] :

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:

```
In [77]: 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):
        col_ = 'T_{}'.format(start_lag + s)
        X[col_] = 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:

```

In [78]: 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:

```
In [79]: X_train.head()
```

Out[79] :

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_10	T_11	T_12	T_13	T_14	T_15	T_16	T_17	T_18	T_19	T_20	T_21	T_22	T_23	T_24
2003-05-02 01:00:00	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096	1.6290	1.6036	1.5976	1.6236	1.6242	1.6342	1.6382	1.6074	1.5536	1.3524	1.0226	0.8020	0.7002	0.6822	0.7156	
2003-05-02 02:00:00	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096	1.6290	1.6036	1.5976	1.6236	1.6242	1.6342	1.6382	1.6074	1.5536	1.3524	1.0226	0.8020	0.7002	0.6822	0.7156
2003-05-02 03:00:00	0.7218	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096	1.6290	1.6036	1.5976	1.6236	1.6242	1.6342	1.6382	1.6074	1.5536	1.3524	1.0226	0.8020	0.7002	0.6822
2003-05-02 04:00:00	0.6914	0.7218	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096	1.6290	1.6036	1.5976	1.6236	1.6242	1.6342	1.6382	1.6074	1.5536	1.3524	1.0226	0.8020	0.7002
2003-05-02 05:00:00	0.7018	0.6914	0.7218	0.7742	0.8694	1.0414	1.3096	1.5408	1.6008	1.5228	1.5486	1.6096	1.6290	1.6036	1.5976	1.6236	1.6242	1.6342	1.6382	1.6074	1.5536	1.3524	1.0226	0.8020

```
In [80]: y_train.head()
```

Out[80] :

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:

```
In [81]: 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:

```
In [82]: X_train_t.shape
```

```
Out[82]: (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.

```
In [83]: 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!

```
In [84]: model.fit(X_train_t, y_train_t,
                  epochs=5,
                  batch_size=256,
                  verbose=1);
```

```

Epoch 1/5
93528/93528 [=====] - 13s 134us/step - loss: 0.0709
Epoch 2/5
93528/93528 [=====] - 12s 128us/step - loss: 0.0595
Epoch 3/5
93528/93528 [=====] - 12s 128us/step - loss: 0.0594
Epoch 4/5
93528/93528 [=====] - 12s 129us/step - loss: 0.0593
Epoch 5/5
93528/93528 [=====] - 12s 129us/step - loss: 0.0588

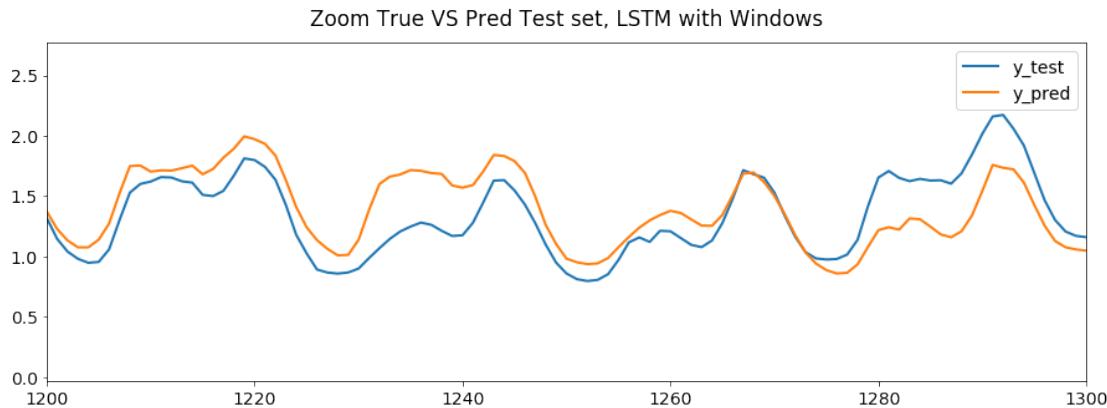
```

Let's generate the predictions and compare them with the actual values:

```

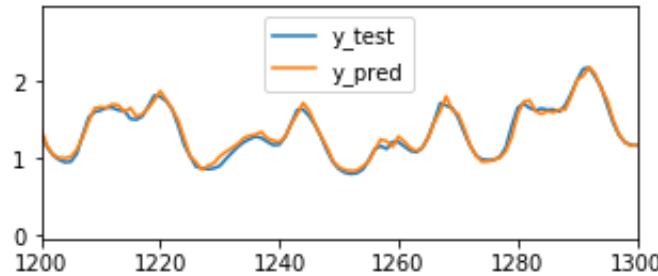
In [85]: y_pred = model.predict(X_test_t, batch_size=256)
plt.figure(figsize=(15,5))
plt.plot(y_test_t, label='y_test')
plt.plot(y_pred, label='y_pred')
plt.legend()
plt.xlim(1200,1300)
plt.title("Zoom True VS Pred Test set, LSTM with Windows");

```



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.

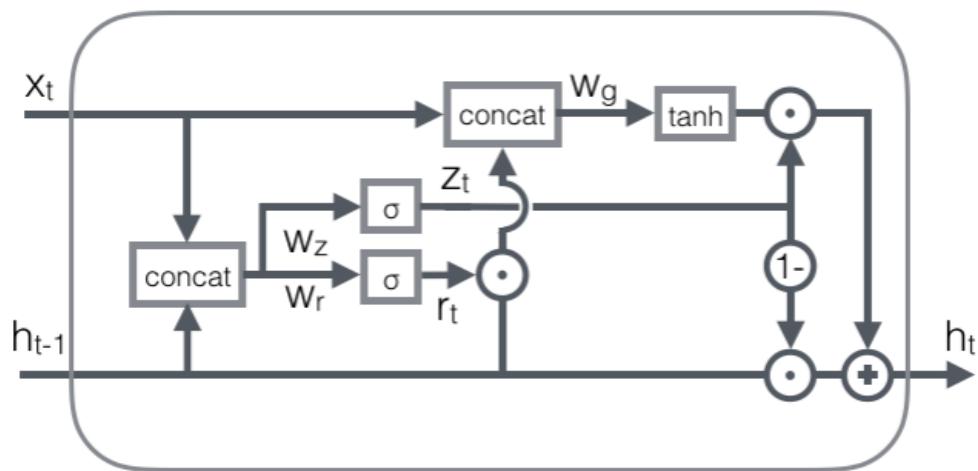
- Go ahead and use the helper function `create_lagged_Xy_win` we created above to generate new X and y pairs where the `start_lag` is 36 hours or even further. You may want to extend the window size to a little longer than a day.
- Train your best model on this data. You may have to use more than one 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?



GRU network graph

**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.
- 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:

$$\nu_t = 100 \times \frac{x_t - x_{t-1}}{x_{t-1}} \quad (7.25)$$

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.



# 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. 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.

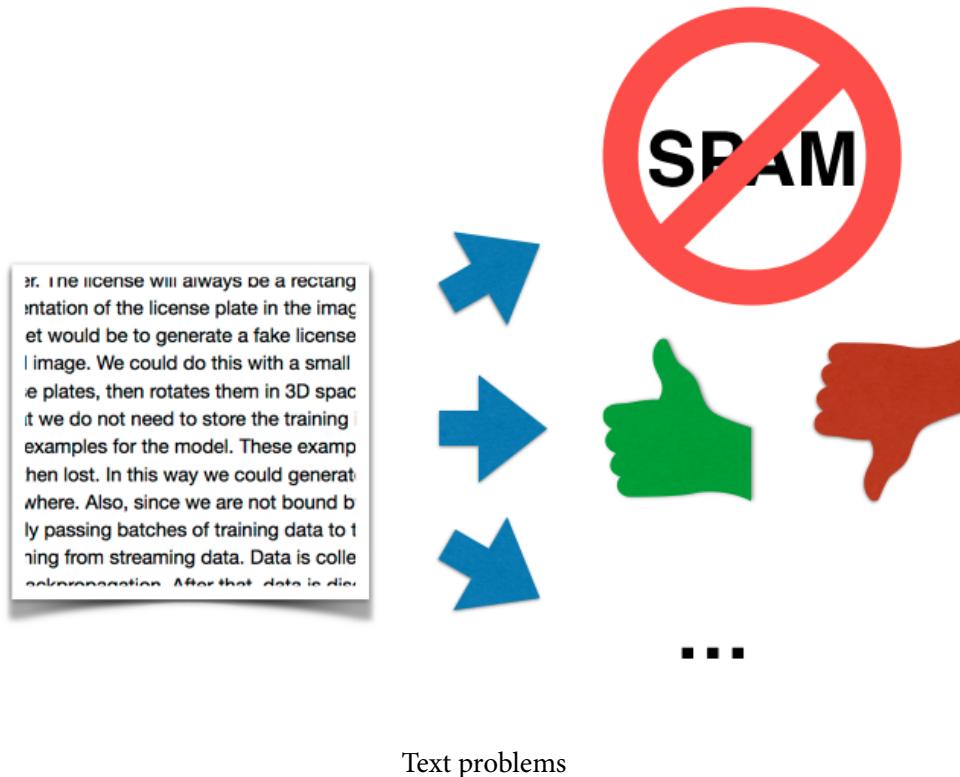
Spam detection can be applied to email spam, sms spam, im spam and in general any corpus of messages. The problem is usually presented as a binary classification where one has two sets of documents: the spam messages and the “ham” messages, i.e. the legitimate messages that we would like to keep.

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.



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:

```
In [1]: 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:

```
In [2]: 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 many functions to interact with the host system. Let's import it:

```
In [3]: import os
```

And let's use the command `os.listdir` to look at the content of the folder:

```
In [4]: os.listdir('../data/sms')
```

```
Out[4]: ['ham', 'spam']
```

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.

```
In [5]: from os.path import isfile, join
```

```
In [6]: def count_files(path):
    files_list = [name for name in os.listdir(path)
                  if isfile(join(path, name))]
    return len(files_list)
```

Let's use this function to count the number of files in the folders:

```
In [7]: ham_count = count_files('../data/sms/ham/')
ham_count
```

```
Out[7]: 4825
```

```
In [8]: spam_count = count_files('../data/sms/spam/')
spam_count
```

```
Out[8]: 747
```

We have 4825 ham files and 747 spam files. We can use these numbers to establish a baseline for our classification efforts:

```
In [9]: baseline_acc = ham_count / (ham_count + spam_count)
        print("Baseline accuracy: {:.3f}".format(baseline_acc))
```

```
Baseline 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:

```
In [10]: def read_file(path):
            with open(path) as fin:
                msg = fin.read()
            return msg
```

```
In [11]: read_file('../data/sms/ham/msg_000.txt')
```

```
Out[11]: 'Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Ci
```

```
In [12]: read_file('../data/sms/ham/msg_001.txt')
```

```
Out[12]: 'Ok lar... Joking wif u oni...'
```

```
In [13]: read_file('../data/sms/spam/msg_002.txt')
```

```
Out[13]: "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 t
```

```
In [14]: read_file('../data/sms/spam/msg_005.txt')
```

```
Out[14]: "FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun y
```

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:

```
In [15]: from sklearn.datasets import load_files
```

```
In [16]: data = load_files('../data/sms/', encoding='utf-8')
```

`data` is an object of type:

```
In [17]: type(data)
```

```
Out[17]: 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:

```
In [18]: data.keys()
```

```
Out[18]: 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`.

```
In [19]: docs = data.data
```

The first five text examples in our `docs` variable are:

```
In [20]: docs[:5]
```

```
Out[20]: ['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 friend",
           '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.']}
```

```
In [21]: y = data.target
```

The first five entries in our y variable:

```
In [22]: y[:5]
```

```
Out[22]: 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 common files:

```
In [23]: with open('common.py') as fin:
    exec(fin.read())
```

```
In [24]: with open('matplotlibconf.py') as fin:
    exec(fin.read())
```

and then create a Dataframe with all the documents

```
In [25]: df = pd.DataFrame(docs, columns=['message'])
df['spam'] = y
df.head()
```

```
Out[25]:
```

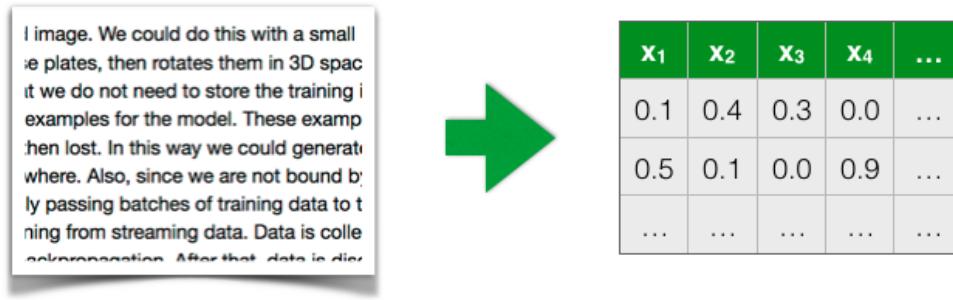
	message	spam
0	Hi Princess! Thank you for the pics. You are v...	0
1	Hello my little party animal! I just thought I...	0
2	And miss vday the parachute and double coins??...	0
3	Maybe you should find something else to do ins...	0
4	What year. And how many miles.	0

Pandas allows to save a dataframe to a variety of different formats, including Excel, CSV and SAS. We will export to CSV:

```
In [26]: 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!



Feature extraction from text

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.

```
In [27]: docs_lower = [d.lower() for d in docs]
```

The first five entries in our `docs_lower` variable are:

```
In [28]: docs_lower[:5]
```

```
Out[28]: ['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 friend"]
```

```
'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.]
```

We can define a simple helper function that counts the occurrences of a particular word in a sentence:

```
In [29]: def count_word(word, sentence):  
    tokens = sentence.split()  
    return len([w for w in tokens if w == word])
```

and apply it to each document:

```
In [30]: free_counts = [count_word('free', d) for d in docs_lower]  
df = pd.DataFrame(free_counts, columns=['free'])
```

```
In [31]: df.head()
```

```
Out[31]:
```

	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:

```
In [32]: import re
```

```
In [33]: def count_numbers(sentence):  
    return len(re.findall('[0-9]', sentence))
```

```
In [34]: df['num_char'] = [count_numbers(d) for d in docs_lower]
```

```
In [35]: df.head()
```

```
Out[35]:
```

free	num_char
0	0
1	0
2	0
3	0
4	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:

```
In [36]: from sklearn.model_selection import train_test_split
         from keras.models import Sequential
         from keras.layers import Dense
```

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:

```
In [37]: 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=0)

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`:

```
In [38]: res = split_fit_eval(df.values, y)
```

```
1393/1393 [=====] - 0s 42us/step
```

Let's check the accuracy of our model:

```
In [39]: print("Simple model accuracy: {:.3f}".format(res[1]))
```

```
Simple model accuracy: 0.971
```

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 baseline, 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`:

```
In [40]: 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**:

```
In [41]: 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:

```
In [42]: 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. Let's look at a few of them:

```
In [43]: stop_words = list(vect.get_stop_words())
```

```
In [44]: stop_words[:10]
```

```
Out[44]: ['ourselves',
          'seemed',
          'cannot',
          'across',
          'those',
          'not',
          'twenty',
          'whole',
          'were',
          'with']
```

Now that we have created the vectorizer, let's apply it to our corpus:

```
In [45]: X = vect.fit_transform(docs)
X
```

```
Out[45]: <5572x3000 sparse matrix of type '<class 'numpy.int64'>'  
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:

```
In [46]: 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:

```
In [47]: vocab = vect.get_feature_names()
```

The features are listed in alphabetical order:

```
In [48]: vocab[:10]
```

```
Out[48]: ['00',
'000',
'02',
'0207',
'02073162414',
'03',
'04',
'05',
'06',
'07123456789']
```

In [49]: `vocab[-10:]`

Out[49]: `['yogasana', 'yor', 'yr', 'yrs', 'yummy', 'yun', 'unny', 'yuo', 'yup', 'zed']`

Let's use the helper function we've defined above to train a model on the new features:

In [50]: `res = split_fit_eval(Xd, y)`

`1393/1393 [=====] - 0s 57us/step`

In [51]: `print("Test set accuracy:\t{:0.3f}\n".format(res[1]))`

`Test set accuracy: 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:

In [52]: `model = res[2]`

Then let's put the weights in a Pandas Series, indexed by the vocabulary:

In [53]: `w_ = model.get_weights()[0].ravel()  
vocab_weights = pd.Series(w_, index=vocab)`

Let's look at the top 20 words with positive weights:

In [54]: `vocab_weights.sort_values(ascending=False).head(20)`

Out[54]:

	o
txt	0.567846
claim	0.529718
free	0.529599
18	0.520037
uk	0.514894
mobile	0.508967
reply	0.499052
www	0.497820
15op	0.491184
service	0.486691
1000	0.440858
prize	0.437020
stop	0.414112
com	0.406652
50	0.405576
urgent	0.393387
rate	0.387784
16	0.386934
ringtone	0.380585
text	0.369661

Not surprisingly we find here words like www, claim, prize, cash etc. Similarly we can look at the bottom 20 words:

```
In [55]: vocab_weights.sort_values(ascending=False).tail(20)
```

Out [55] :

---

	o
lt	-0.448238
lol	-0.451866
tell	-0.455773
doing	-0.459791
lor	-0.470848
need	-0.476604
home	-0.487474
later	-0.488070
yeah	-0.490425
think	-0.491266
like	-0.503484
oh	-0.504350
good	-0.512226
sorry	-0.516341
got	-0.517523
going	-0.522831
come	-0.550252
da	-0.552738
ll	-0.581713
ok	-0.634840

---

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_word_in_corpus
```

or using maths:

$$\text{tf-idf}(w, d) = \frac{\text{tf}(w, d)}{\text{df}(w, d)} \quad (8.1)$$

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](#):

$$\text{tf-idf}(w, d) = \text{tf}(w, d) \times \log\left(\frac{1 + n_d}{1 + \text{df}(w, d)}\right) + 1 \quad (8.2)$$

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](#):

```
In [56]: df = pd.read_csv('../data/movie_reviews.csv')
df.head()
```

Out[56] :

	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.

In [57]: `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.

In [58]: `df['vote'].value_counts() / len(df)`

Out[58] :

vote
fresh 0.612067
rotten 0.386299
none 0.001634

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`:

In [59]: `df = df[df.vote != 'none'].copy()`

In [60]: `df['vote'].value_counts() / len(df)`

```
Out[60] :
```

vote	
fresh	0.613069
rotten	0.386931

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`:

```
In [61]: from sklearn.preprocessing import LabelEncoder
```

Let's instantiate the `LabelEncoder`:

```
In [62]: le = LabelEncoder()
```

Finally, let's create a vector of 0s and 1s that represent the features:

```
In [63]: y = le.fit_transform(df['vote'])
```

`y` is now a vector of 0s and 1s. Let's look at the first 10 entries:

```
In [64]: y[:10]
```

```
Out[64]: 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:

```
In [65]: 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:

```
In [66]: vocab_size = 10000
```

```
vect = TfidfVectorizer(decode_error='ignore',
                      stop_words='english',
                      max_features=vocab_size)
```

We can use the vectorizer to transform our reviews:

```
In [67]: X = vect.fit_transform(df['review'])
```

```
In [68]: X
```

```
Out[68]: <14049x10000 sparse matrix of type '<class 'numpy.float64'>'  
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:

```
In [69]: 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:

```
In [70]: res = split_fit_eval(Xd, y, epochs=30)
```

```
3513/3513 [=====] - 0s 60us/step
```

```
In [71]: print("Test set accuracy:\t{:0.3f}".format(res[1]))
```

```
Test set accuracy: 0.751
```

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:

```
In [72]: 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:

```
In [73]: vocab_size
```

```
Out[73]: 10000
```

```
In [74]: 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:

```
In [75]: 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:

In [76]: `tokenizer.document_count`

Out[76]: 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:

In [77]: `tokenizer.num_words`

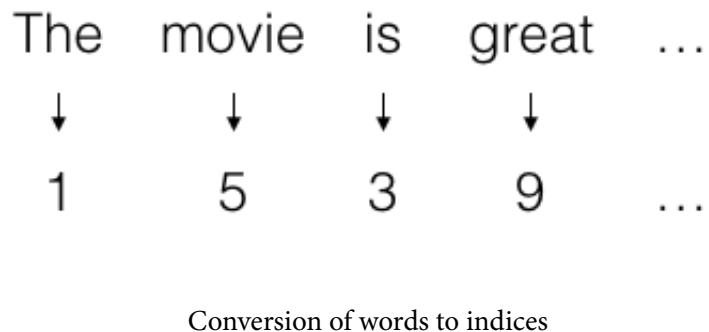
Out[77]: 10000

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:

In [78]: `list(tokenizer.word_index)[:10]`

Out[78]: `['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:



In [79]: `sequences = tokenizer.texts_to_sequences(df['review'])`

`sequences` is a list of lists. Each of the inner lists is one of the reviews:

In [80]: `sequences[:3]`

```
Out[80]: [[36,
 1764,
 7,
 1058,
 800,
 3,
 1765,
 9,
 27,
 151,
 268,
 8,
 21,
 2,
 9088,
 3879,
 5881,
 115,
 3,
 101,
 20,
 22,
 17,
 360],
 [1, 610, 38, 801, 49],
 [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  $\rightarrow$  word map:

```
In [81]: tok_items = tokenizer.word_index.items()
idx_to_word = {i:w for w, i in tok_items}
```

The first review is:

```
In [82]: df.loc[0, 'review']
```

```
Out[82]: 'So ingenious in concept, design and execution that you could watch it on a postage sta
```

The first sequence is:

```
In [83]: ' '.join([idx_to_word[i] for i in sequences[0]])
```

```
Out[83]: '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:

```
In [84]: maxlen = max([len(seq) for seq in sequences])
maxlen
```

```
Out[84]: 49
```

The longest review contains 49 words. Let's pad every other review to 49 using the `pad_sequences` function from Keras:

```
In [85]: 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='pre', value=0.0)`

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:

```
In [86]: X = pad_sequences(sequences, maxlen=maxlen)
```

In [87]: `X.shape`

Out[87]: (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:

In [88]: `X[:4]`

```
Out[88]: array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
   0,  0,  0, 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,  0,  0,  0,  0,  0,  0,  0,  0,
  1, 610, 38, 801, 49], [
  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,  2, 1012, 347, 225,  9, 24, 107,
 564, 21, 1, 354, 7122], [
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,
 16, 1190, 2, 822, 3, 485, 42, 121, 144, 285,
1678, 4, 13, 742, 724]], dtype=int32)
```

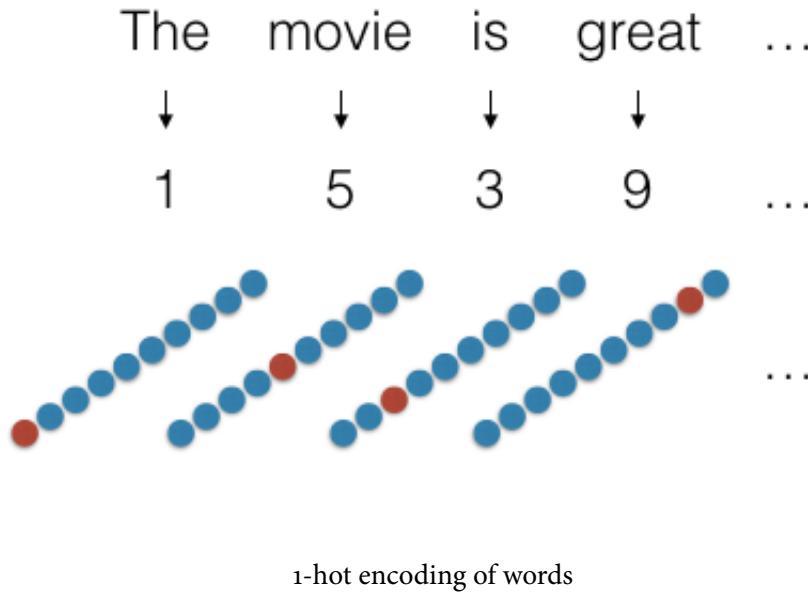
Can we feed this matrix to a Machine Learning model? Let's think about it for a second. If we treat this matrix as tabular data, it would mean each column represents a feature. But what feature? Columns in the matrix correspond to the position of the word in a sentence and so there's absolutely no reason 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.



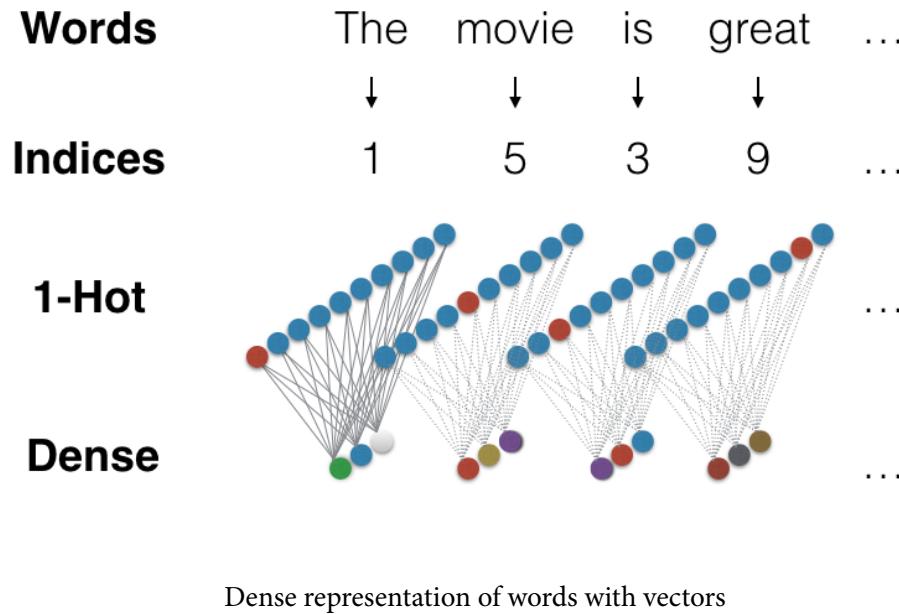
It looks like one way to feed this data to a Neural Network would be to expand the X matrix to a 1-hot encoded order-3 tensor with 0s and 1s, and then feed this tensor to our network, for example to a [Recurrent layer](#) with `input_shape=(49, 10000)`. This would be the equivalent of feeding a dataset 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.

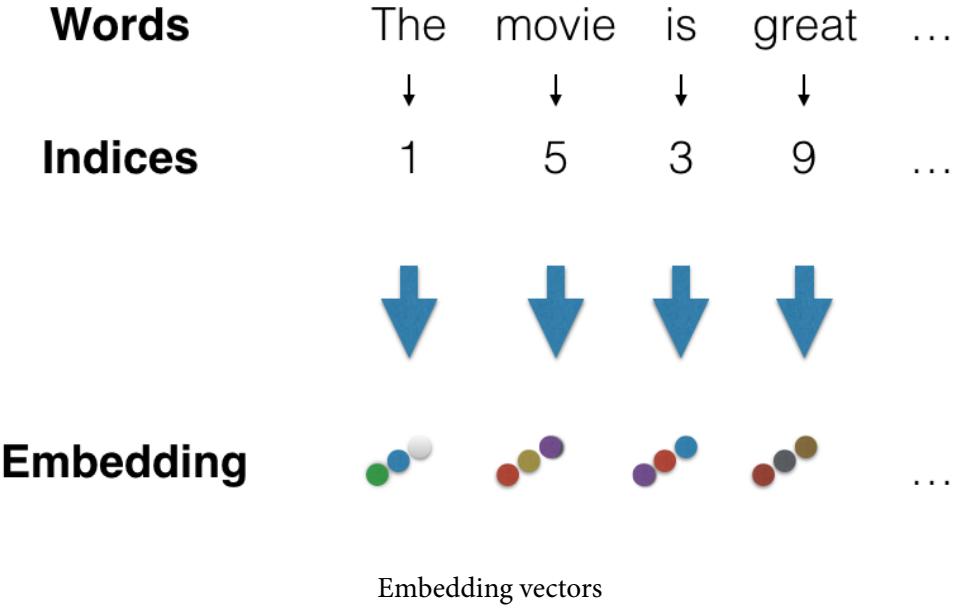
Let's see how to include this in our own network. Let's load the **Embedding** layers from Keras:

```
In [89]: 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:

```
In [90]: 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:

```
In [91]: model.predict(np.array([[ 0, 81, 1, 0, 79]]))
```

```
Out[91]: array([[-0.01208742,  0.02399189],
   [ 0.00883657, -0.00948893],
   [ 0.04218039,  0.00517732],
   [-0.01208742,  0.02399189],
   [-0.02936612, -0.03825488]], 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 0 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 0 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):

```
In [92]: model.predict(np.array([[ 0, 81, 1, 96, 79],
   [ 4, 17, 47, 69, 50],
   [15, 49, 3, 12, 88]]))
```

```
Out[92]: array([[-0.01208742,  0.02399189],  
                 [ 0.00883657, -0.00948893],  
                 [ 0.04218039,  0.00517732],  
                 [ 0.00621601,  0.04060371],  
                 [-0.02936612, -0.03825488]],  
  
                [[-0.02983077, -0.01654588],  
                 [-0.02862899, -0.04024762],  
                 [ 0.03696802,  0.01382831],  
                 [ 0.04779704, -0.03472906],  
                 [ 0.00352927, -0.00599594]],  
  
                [[-0.03610412,  0.0394091 ],  
                 [ 0.03755711, -0.01945177],  
                 [-0.01093823, -0.01441556],  
                 [-0.04975789, -0.00622205],  
                 [-0.0464707 ,  0.03933306]], 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.

```
In [93]: X_train, X_test, y_train, y_test = \  
          train_test_split(X, y, random_state=0)
```

```
In [94]: X.shape
```

```
Out[94]: (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:

```
In [95]: 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:

```
In [96]: 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:

```
In [97]: 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 [=====] - 6s 684us/step - loss: 0.6667 - acc: 0.6157 - val_loss: 0.6572 - val_acc: 0.6067
Epoch 2/8
8428/8428 [=====] - 5s 579us/step - loss: 0.5833 - acc: 0.6688 - val_loss: 0.5679 - val_acc: 0.7092
Epoch 3/8
8428/8428 [=====] - 5s 580us/step - loss: 0.3961 - acc: 0.8340 - val_loss: 0.5833 - val_acc: 0.7453
Epoch 4/8
8428/8428 [=====] - 5s 580us/step - loss: 0.2714 - acc: 0.8974 - val_loss: 0.5965 - val_acc: 0.7476
Epoch 5/8
8428/8428 [=====] - 5s 579us/step - loss: 0.1995 - acc: 0.9277 - val_loss: 0.7437 - val_acc: 0.7334
Epoch 6/8
8428/8428 [=====] - 5s 580us/step - loss: 0.1439 - acc: 0.9514 - val_loss: 0.7884 - val_acc: 0.7400
Epoch 7/8
```

```
8428/8428 [=====] - 5s 579us/step - loss: 0.1033 - acc:  
0.9680 - val_loss: 0.9223 - val_acc: 0.7362  
Epoch 8/8  
8428/8428 [=====] - 5s 580us/step - loss: 0.0761 - acc:  
0.9765 - val_loss: 0.9755 - val_acc: 0.7225
```

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:

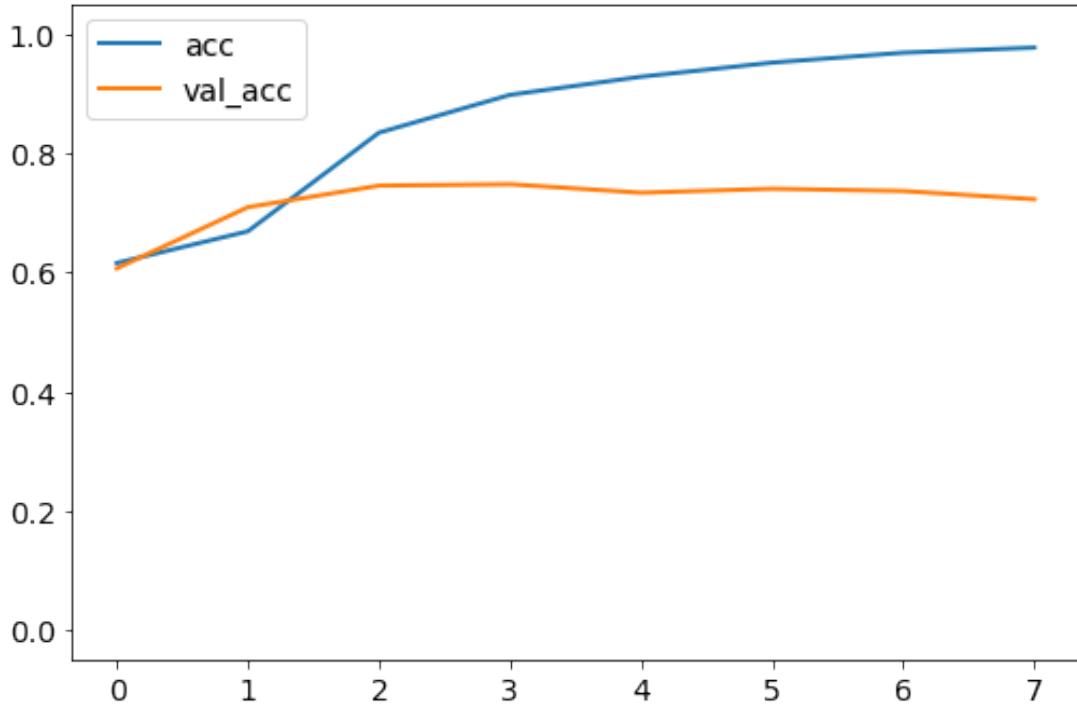
```
In [98]: loss, acc = model.evaluate(X_test, y_test, batch_size=32)  
acc
```

```
3513/3513 [=====] - 3s 712us/step
```

```
Out[98]: 0.7386848847648204
```

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:

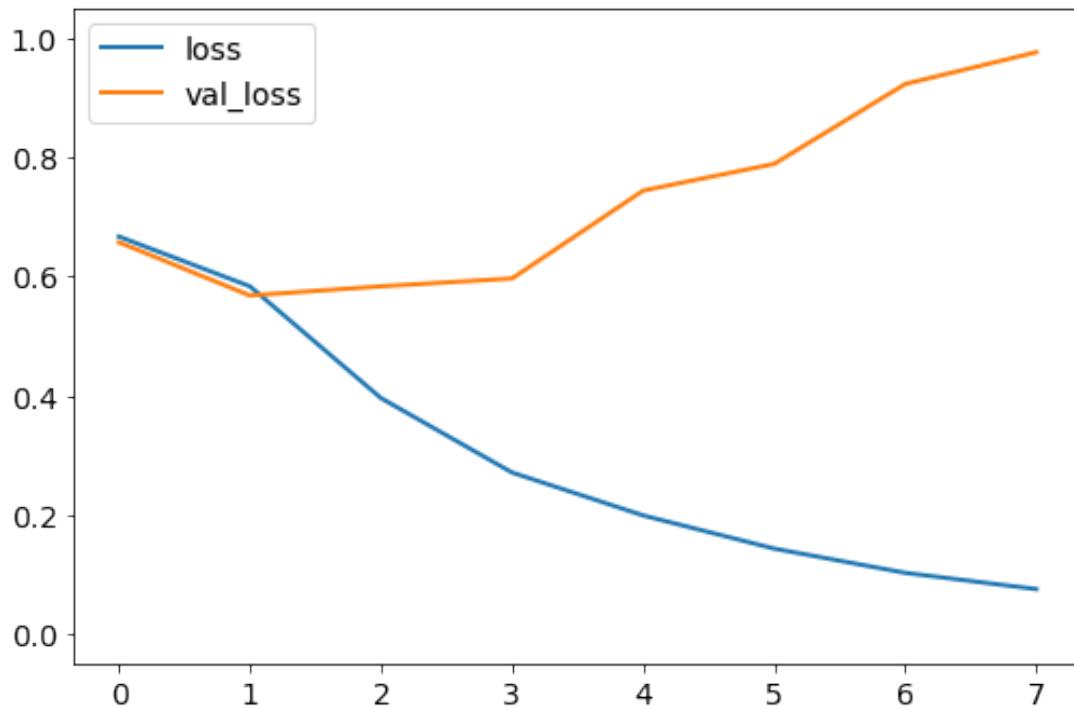
```
In [99]: dfhistory = pd.DataFrame(h.history)  
dfhistory[['acc', 'val_acc']].plot(ylim=(-0.05, 1.05));
```



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.

```
In [100]: dfhistory[['loss', 'val_loss']].plot(ylim=(-0.05, 1.05));
```



How many weights are there in our model? Is it too big?

```
In [101]: model.summary()
```

```
Layer (type)          Output Shape         Param #  
=====          ======         ======-----  
embedding_2 (Embedding)    (None, 49, 16)      160000  
lstm_1 (LSTM)           (None, 32)          6272  
dense_4 (Dense)          (None, 1)           33  
=====-----  
Total params: 166,305  
Trainable params: 166,305  
Non-trainable params: 0  
=====-----
```

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

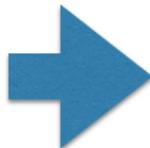
As mentioned at the beginning of this chapter and in the previous one, Neural Networks are not only suited to deal with textual input data but also to generate text output. In fact, text can be viewed as a **sequence of words** or as a **sequence of characters** and we can use a model to predict the 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
...	...

Windows of text

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.

```
In [102]: with open('../data/names.txt') as f:
```

```
names = f.readlines()
names = [n.lower().strip() + '\n' for n in names]

print('Loaded %d names' % len(names))
```

```
Loaded 7939 names
```

Let's have a look at the first three of them:

```
In [103]: names[:3]
```

```
Out[103]: ['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:

```
In [104]: chars = set()

for name in names:
    chars.update(name)

vocab_size = len(chars)
```

Let's look at the number of chars we've saved:

```
In [105]: vocab_size
```

```
Out[105]: 28
```

```
In [106]: chars
```

```
Out[106]: {'\n',
           '_',
           'a',
           'b',
           'c',
           'd',
```

```
'e',
'f',
'g',
'h',
'i',
'j',
'k',
'l',
'm',
'n',
'o',
'p',
'q',
'r',
's',
't',
'u',
'v',
'w',
'x',
'y',
'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.

```
In [107]: char_to_idx = 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:

```
In [108]: int_names = [[char_to_idx[c] for c in n] for n in names]
```

Now each name has been converted to a sequence of integers, for example, the first name:

```
In [109]: names[0]
```

```
Out[109]: 'aamir\n'
```

Was converted to:

```
In [110]: int_names[0]
```

```
Out[110]: [2, 2, 19, 0, 27, 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:

```
In [111]: 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:

```
In [112]: name_parts[:4]
```

```
Out[112]: [[2, 2, 19], [2, 19, 0], [19, 0, 27], [2, 2, 27]]
```

next\_chars is a list with single entries, each representing the next character:

```
In [113]: next_chars[:4]
```

```
Out[113]: [0, 27, 12, 17]
```

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:

```
In [114]: X = pad_sequences(name_parts, maxlen=maxlen)
```

The final shape of our input is:

```
In [115]: X.shape
```

```
Out[115]: (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`:

```
In [116]: 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:

```
In [117]: y = to_categorical(next_chars, vocab_size)
```

The shape of our labels is:

```
In [118]: y.shape
```

```
Out[118]: (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:

```
In [119]: 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.

```
In [120]: inputs = Input(shape=(3, ))
```

Let's look at the inputs variable we have just defined:

```
In [121]: inputs
```

```
Out[121]: <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.

```
In [122]: 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).

```
In [123]: 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:

```
In [124]: 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:

```
In [125]: 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:

```
In [126]: 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:

```
In [127]: 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:

```
In [128]: model.fit(X, y, epochs=20)
```

```
Epoch 1/20
32016/32016 [=====] - 8s 265us/step - loss: 2.6411 -
acc: 0.2466
Epoch 2/20
32016/32016 [=====] - 8s 240us/step - loss: 2.4984 -
```

```
acc: 0.2535
Epoch 3/20
32016/32016 [=====] - 8s 240us/step - loss: 2.3988 -
acc: 0.2724
Epoch 4/20
32016/32016 [=====] - 8s 241us/step - loss: 2.3453 -
acc: 0.2787
Epoch 5/20
32016/32016 [=====] - 8s 242us/step - loss: 2.3163 -
acc: 0.2814
Epoch 6/20
32016/32016 [=====] - 8s 242us/step - loss: 2.2936 -
acc: 0.2885
Epoch 7/20
32016/32016 [=====] - 8s 243us/step - loss: 2.2736 -
acc: 0.2958
Epoch 8/20
32016/32016 [=====] - 8s 242us/step - loss: 2.2565 -
acc: 0.2992
Epoch 9/20
32016/32016 [=====] - 8s 243us/step - loss: 2.2419 -
acc: 0.3010
Epoch 10/20
32016/32016 [=====] - 8s 242us/step - loss: 2.2291 -
acc: 0.3030
Epoch 11/20
32016/32016 [=====] - 8s 243us/step - loss: 2.2172 -
acc: 0.3046
Epoch 12/20
32016/32016 [=====] - 8s 240us/step - loss: 2.2056 -
acc: 0.3081
Epoch 13/20
32016/32016 [=====] - 8s 241us/step - loss: 2.1948 -
acc: 0.3106
Epoch 14/20
32016/32016 [=====] - 8s 241us/step - loss: 2.1850 -
acc: 0.3158
Epoch 15/20
32016/32016 [=====] - 8s 240us/step - loss: 2.1756 -
acc: 0.3189
Epoch 16/20
32016/32016 [=====] - 8s 239us/step - loss: 2.1676 -
acc: 0.3223
Epoch 17/20
32016/32016 [=====] - 8s 240us/step - loss: 2.1603 -
acc: 0.3254
Epoch 18/20
32016/32016 [=====] - 8s 240us/step - loss: 2.1537 -
acc: 0.3295
Epoch 19/20
32016/32016 [=====] - 8s 241us/step - loss: 2.1478 -
acc: 0.3320
Epoch 20/20
32016/32016 [=====] - 8s 240us/step - loss: 2.1420 -
acc: 0.3348
```

Out[128]: <keras.callbacks.History at 0x7f67e69f2c88>

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  $\mathbf{p}$ . 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:

$$\text{sample}(p) := \text{argmax}(\text{multinomial}(1, p, 1)) \quad (8.3)$$

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:

```
In [129]: 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.

```
In [130]: 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:

```
In [131]: draws = [sample(probs) for i in range(100)]
```

and let's use a Counter to count how many of each we drew:

```
In [132]: from collections import Counter
```

```
In [133]: Counter(draws)
```

```
Out[133]: Counter({1: 60, 0: 29, 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.

```
In [134]: 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:

```
In [135]: 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`:

```
In [136]: stop = False
```

Finally let's loop until we have to stop:

```
In [137]: while not stop:
    for i, c in enumerate(out[-maxlen:]):
        x[0, i] = char_to_idx[c]

    preds = model.predict(x, verbose=0)[0]

    c = inds_to_char[sample(preds)]
    out += c

    if c == '\n':
        stop = True
out
```

```
Out[137]: 'alie\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.

```
In [138]: 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 '.'
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_idx[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:

```
In [139]: for i in range(10):
    print(complete_name('jen'))
```

jenaann

jene

```
jenie
jenne
jenn
jen
jenta
jenanl
jena
jenen
```

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:

```
In [140]: for i in range(10):
    print(complete_name('jen', diversity=10,
                         max_name_len=20))
```

```
jenzycmz-oapqwoeqliw
jenclwypkloqvtkpbsgi
jentspmwikqfxojlopdf
jenjwdi

jenwhyrq

jenjooogynah-yzk-qjl
jenkkdpakbi-

jenemsfy-lwkcrutvzsy
jen

jensyletawbk
```

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.

```
In [141]: for i in range(10):
    print(complete_name('jen', diversity=0.01,
                         max_name_len=20))
```

```
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.

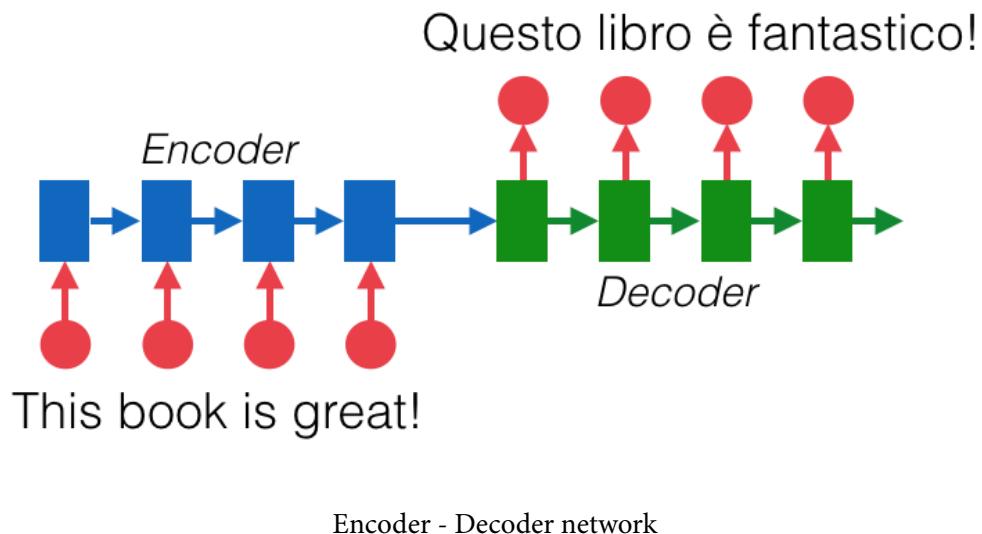
In the next section we will extend our skills to build a language translation model.

### 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.

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.



Encoder - Decoder network

## 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 datafram
- 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:

# 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.



NVIDIA Graphics cards

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*.



## CPU versus GPU

### 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**

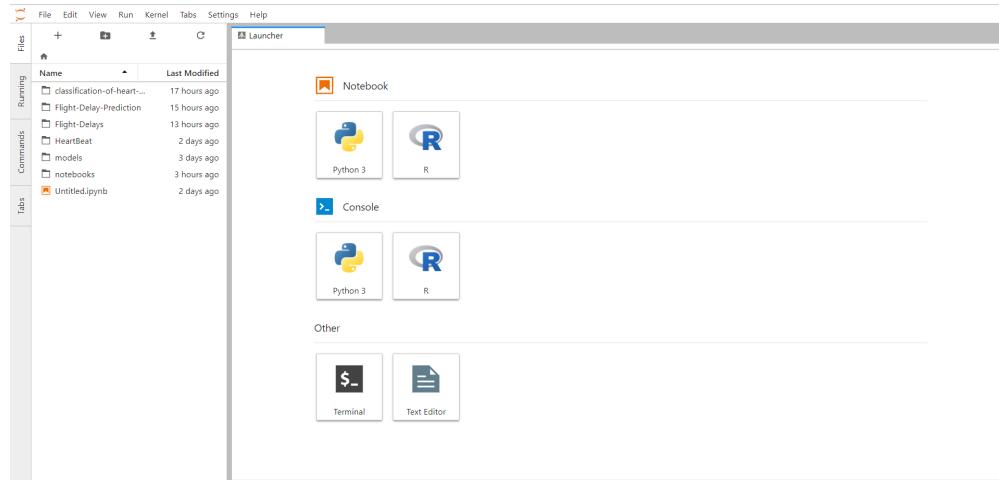
Once you've done that, you can run this code to verify that GPU is available

```
import tensorflow as tf
tf.test.gpu_device_name()
```

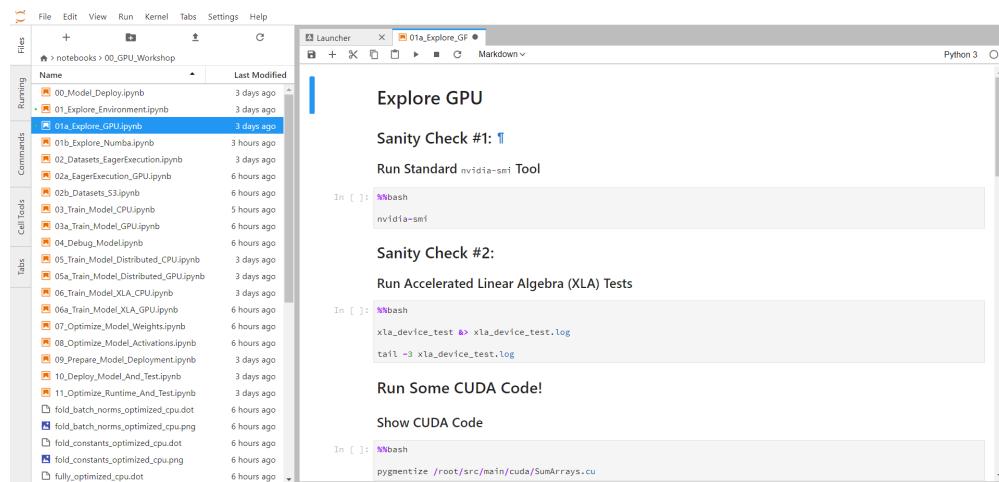
## Pipeline AI

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](#).

The figure consists of three screenshots of the PipelineAI platform:

- Monitor Prediction Traffic:** A dashboard showing two monitoring panels for 'mnist-a' and 'mnist-b'. Each panel includes a bar chart for 'Error %' (0.0%), a table for 'Hosts' (1), 'Mean' (0.0s), '99.5th' (0.0s), and 'Circuit Closed' (0.0s/p). Below the charts are 'Hosts' (1), 'Mean' (0ms), '99.5th' (0ms), and 'Circuit Closed' (0ms).
- AWS S3 Explorer:** A file browser interface titled 'AWS S3 Explorer' showing a list of objects under 'datapalooza'. The list includes 'airbnb/', 'census/', 'dating/', 'dogs\_and\_cats/', 'eigenface/', 'email/', 'geo/', 'graph/', 'inception/', 'ldr/', 'meetup/', 'mnist/', 'movielens/', and '...'. There are 50 entries shown.
- Pipeline DB > Projects:** A list of projects. The first project, 'mnist\_gridsearch\_cross\_validation', was created on '05/05/2018' and has a 'Default' status. It includes a table of metrics: 'Metrics' (multiclass), 'min' (0.938), 'max' (1), and 'average' (0.98). It is associated with '60 models'. The second project, 'basic\_workflow', was created on '05/05/2018' and has a 'Default' status. It includes a table of metrics: 'Metrics' (accuracy), 'min' (0.8), 'max' (0.9), and 'average' (0.85). It is associated with '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](#)

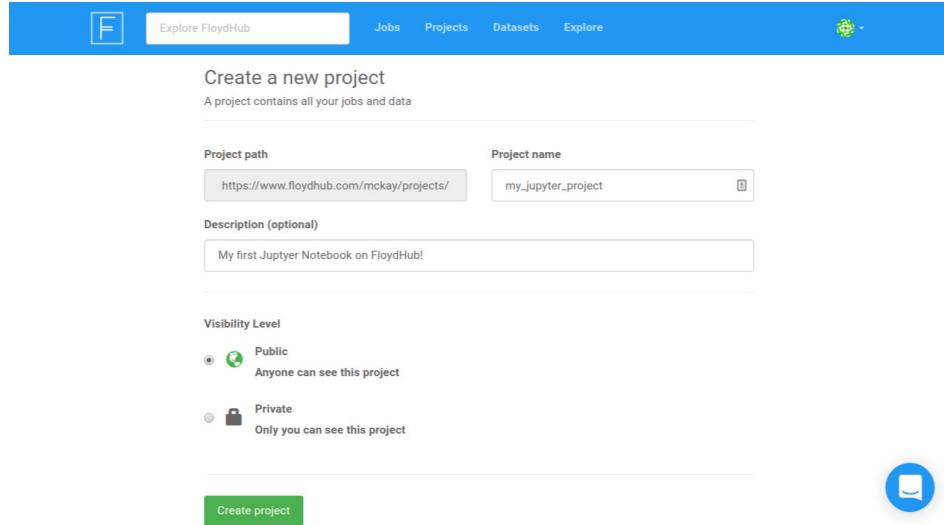
## Floydhub

[Floydhub](#) is another 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:



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:

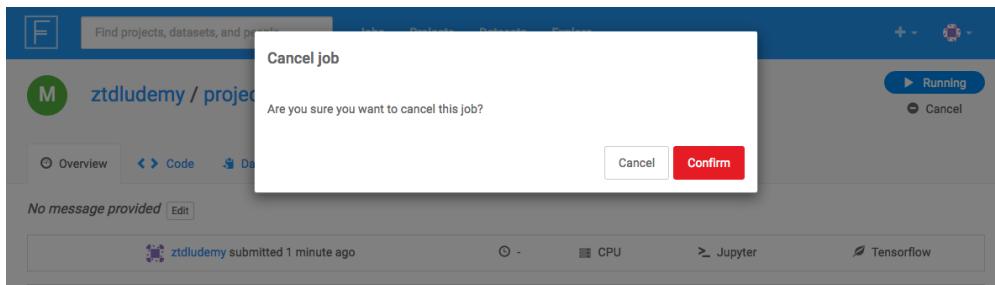
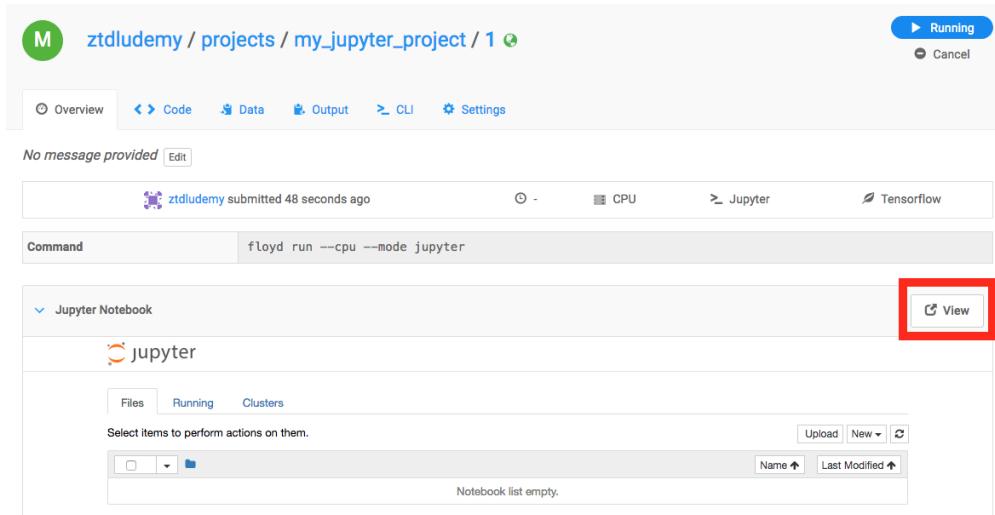
```
Creating project run. Total upload size: 224.0B
Syncing code ...
[=====] 1008/1008 - 00:00:01

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.

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:



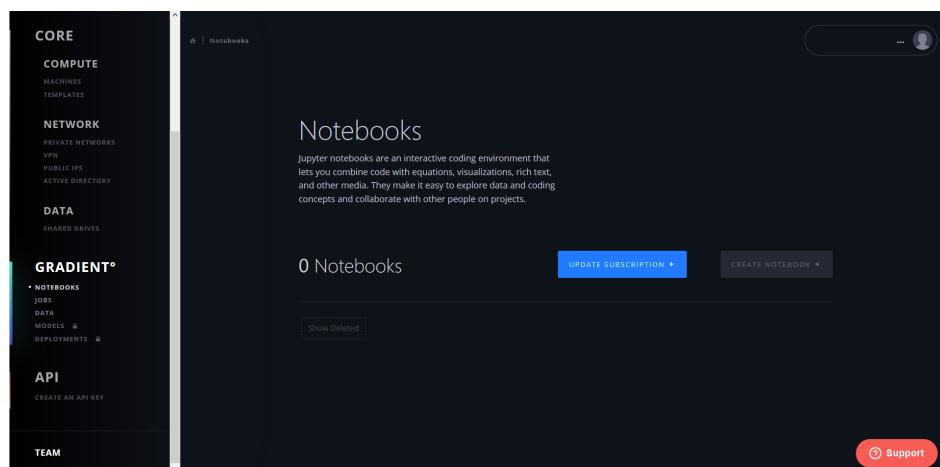
## Paperspace

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](#).

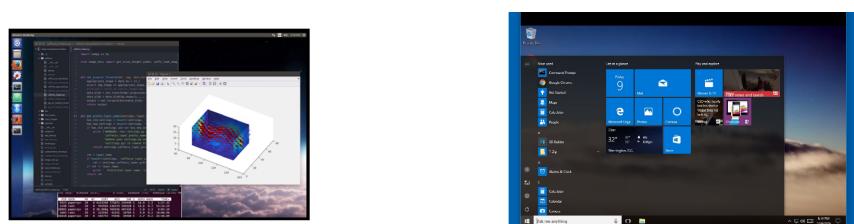
The procedure to run a Jupyter Notebooks within Paperspace is similar to what we have seen so far for different GPU services:

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.



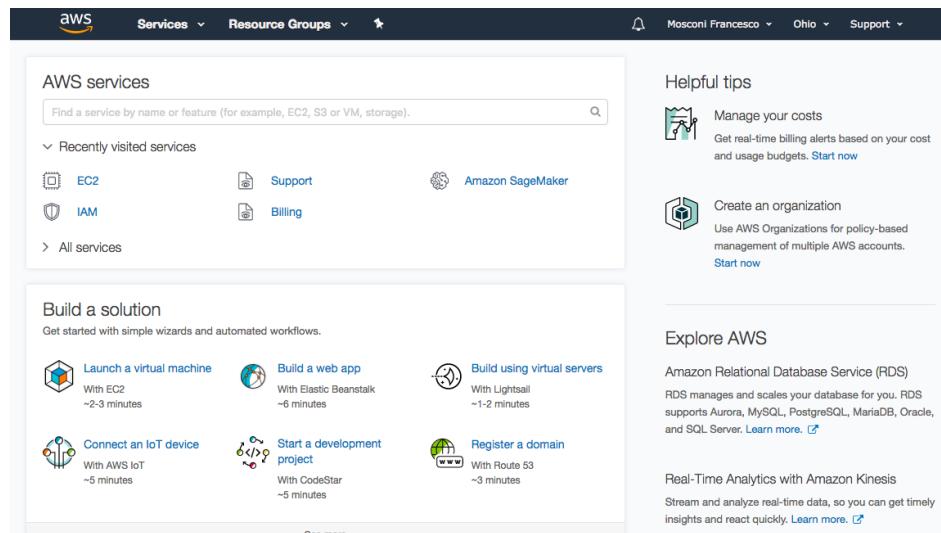
## AWS EC2 Deep Learning AMI

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.



2. Click on the *Launch Instance* button.

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with links for Events, Tags, Reports, Limits, Instances, Launch Templates, Spot Requests, Reserved Instances, and Dedicated Hosts. Below that are IMAGEs, AMIs, and Bundle Tasks. Further down are ELASTIC BLOCK STORE Volumes and Snapshots, and NETWORK & SECURITY Security Groups, Elastic IPs, and Placement Groups. The main area displays resource counts: 0 Running Instances, 0 Dedicated Hosts, 0 Volumes, 0 Key Pairs, 0 Placement Groups, 0 Elastic IPs, 0 Snapshots, 0 Load Balancers, and 1 Security Groups. A callout box provides information about the latest in AWS Compute from AWS re:Invent 2017. To the right, 'Account Attributes' include Supported Platforms (VPC), Default VPC (vpc-fff94096), and Resource ID length management. Below that is 'Additional Information' with links to Getting Started Guide, Documentation, All EC2 Resources, Forums, Pricing, and Contact Us. At the bottom, there's a 'Create Instance' section with a 'Launch Instance' button, service health status, scheduled events, and marketplace information.

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.

The screenshot shows the 'Step 1: Choose an Amazon Machine Image (AMI)' screen. It lists three options: 'Deep Learning AMI (Ubuntu) Version 7.0 - ami-1be7d77e', 'Deep Learning AMI (Amazon Linux) Version 7.0 - ami-e0f7c785', and 'Deep Learning Base AMI (Ubuntu) Version 4.0 - ami-10457475'. Each item includes a 'Select' button and a '64-bit' link. The first item is highlighted with a blue border. The base AMI is marked as 'Free tier eligible'. Descriptions for each AMI mention they come with pre-installed deep learning frameworks like MXNet, TensorFlow, and PyTorch. The base AMI also mentions it includes system libraries for deploying a custom deep learning environment.

4. Chose an instance type from the menu. Roughly speaking, instance types are ordered in ascending order considering the computational power and the storage space.

Compute optimized	Instance Type	vCPUs	Memory (GiB)	Number of GPUs	GPU cores	GPU Memory (GiB)	Total GPU cores	1U Gigabit	Yes
<input checked="" type="checkbox"/>	GPU graphics	g3.4xlarge	16	122	EBS only	Yes	Up to 10 Gigabit	Yes	
<input type="checkbox"/>	GPU graphics	g3.8xlarge	32	244	EBS only	Yes	10 Gigabit	Yes	
<input type="checkbox"/>	GPU graphics	g3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes	
<input type="checkbox"/>	GPU compute	p2.xlarge	4	61	EBS only	Yes	High	Yes	
<input type="checkbox"/>	GPU compute	p2.8xlarge	32	488	EBS only	Yes	10 Gigabit	Yes	
<input type="checkbox"/>	GPU compute	p2.16xlarge	64	732	EBS only	Yes	25 Gigabit	Yes	
<input type="checkbox"/>	GPU compute	p3.2xlarge	8	61	EBS only	Yes	Up to 10 Gigabit	Yes	
<input type="checkbox"/>	GPU compute	p3.8xlarge	32	244	EBS only	Yes	10 Gigabit	Yes	
<input type="checkbox"/>	GPU compute	p3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes	

Cancel Previous Review and Launch Next: Configure Instance Details

Feedback English (US) © 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Here's a summary table of AWS GPU instances. Read the [documentation](#) for a detailed description of every instance type.

Name	GPU model	vCPU	Memory (GiB)	Number of GPUs	GPU cores	GPU Memory (GiB)	Total GPU cores	Total GPU Memory (GiB)	Cost per 1000 cores	Hourly Cost
g2.2xlarge	Grid K80	8	15	1	1536	4	1536	4	\$0.42	\$0.65
p2.xlarge	K80	4	61	1	2496	12	2496	12	\$0.36	\$0.90
g3.4xlarge	M60	16	122	1	2048	8	2048	8	\$0.56	\$1.14
g3.8xlarge	M60	32	244	2	2048	8	4096	16	\$0.56	\$2.28
g2.8xlarge	Grid K80	32	60	4	1536	4	6144	16	\$0.42	\$2.60
p3.2xlarge	V100	8	61	1	5120	16	5120	16	\$0.60	\$3.06
g3.16xlarge	M60	64	488	4	2048	8	8192	32	\$0.56	\$4.56
p2.8xlarge	K80	32	488	8	2496	12	19968	96	\$0.36	\$7.20
p3.8xlarge	V100	32	244	4	5120	16	20480	64	\$0.60	\$12.24
p2.16xlarge	K80	64	732	16	2496	12	39936	192	\$0.36	\$14.40
p3.16xlarge	V100	64	488	8	5120	16	40960	128	\$0.60	\$24.48

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.

The screenshot shows the AWS EC2 Instances page. On the left, there's a sidebar with navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances (selected), Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, AMIs, Bundle Tasks, Elastic Block Store (Volumes, Snapshots), and Network & Security (Security Groups, Elastic IPs). The main content area has a search bar at the top. Below it is a table with columns: Name, Instance ID, Instance Type, Availability Zone, Instance State, Status Checks, and Alarm Status. A single row is selected for an instance named 'Book' with the ID i-07db3a7b63ac50626, type g3.8xlarge, in us-east-2c, pending, and no status checks or alarms. At the bottom of the page, there are tabs for Description, Status Checks, Monitoring, and Tags, along with instance details like Instance ID, Public DNS, Instance state, and IPv4 Public IP.

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_64v)

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 Python3 (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 are buttons for 'Logout', 'Upload', 'New', and a refresh icon. Below the header is a search bar with placeholder text 'Select items to perform actions on them.' Underneath the search bar is a file list table with columns for 'Name' and 'Last Modified'. The file list includes:

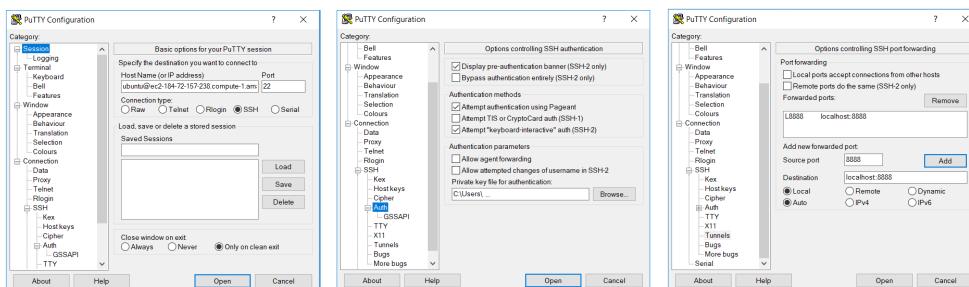
- anaconda3 (5 days ago)
- src (5 days ago)
- tutorials (5 days ago)
- nohup.out (seconds ago)
- Nvidia\_Cloud\_EULA.pdf (7 days ago)

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:

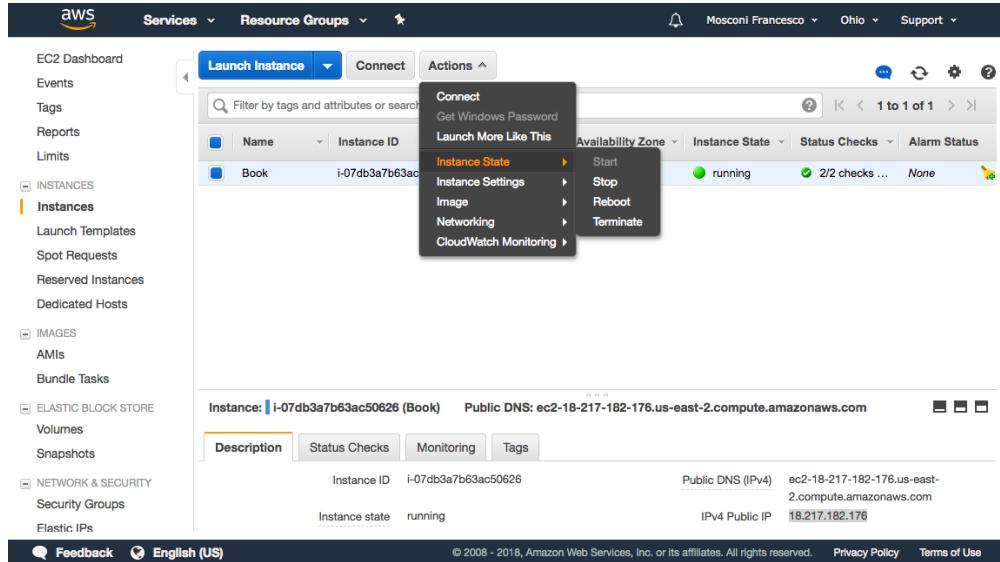
1. In the *Session* palette:
  - Host Name (or IP address): `ubuntu@<your-ip>`
  - Port: 22
  - Connection type: SSH
2. In the *Connection > SSH > Auth* palette:
  - Private key file for authentication: browse the key generated by [PuttyGen](#)
3. In the *Conenction > SSH > Tunnels* palette:
  - Source 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:



### 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:

or text:

```
{
  "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-fff94096",
          "Architecture": "x86_64",
          "BlockDeviceMappings": [

```

```

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-fff94096
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-fff94096
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

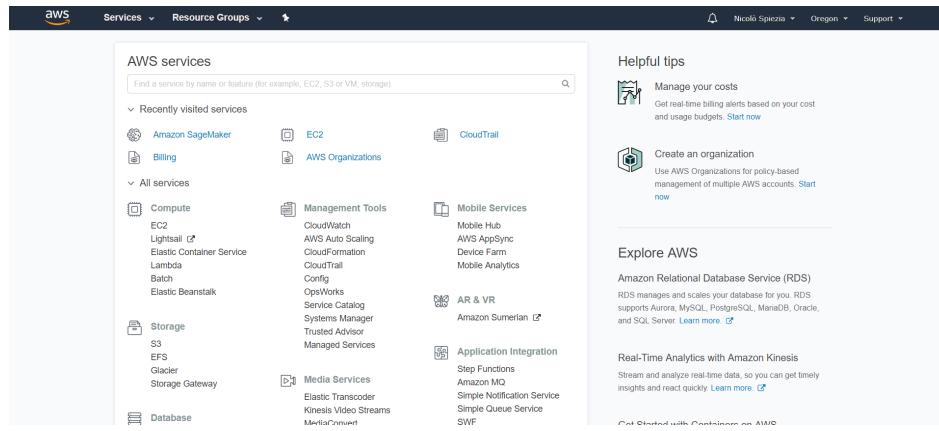
[AWS Sagemaker](#) is an AWS managed solution that allows to perform all the steps involved in a Deep Learning pipeline. In fact, on Sagemaker you can define, train and deploy a Machine Learning model in just a few steps.

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.

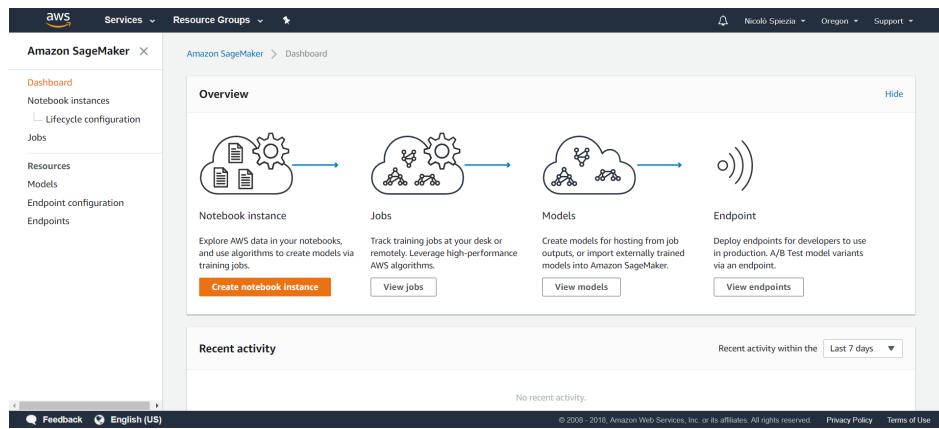
Detailed information about this service can be found in the official [documentation](#).

The procedure to spin up a notebook is similar to what previously seen:

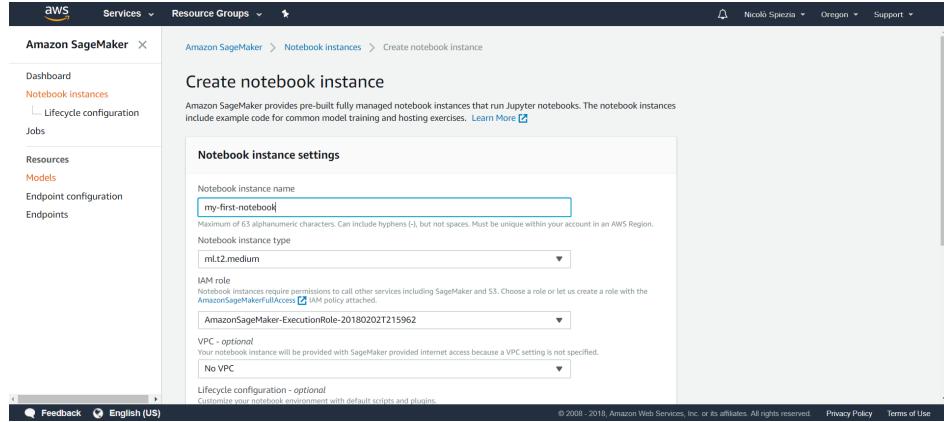
1. [Create an AWS account](#) and access the console.
2. From the AWS console, select the *Amazon SageMaker* service, under the Machine Learning group.



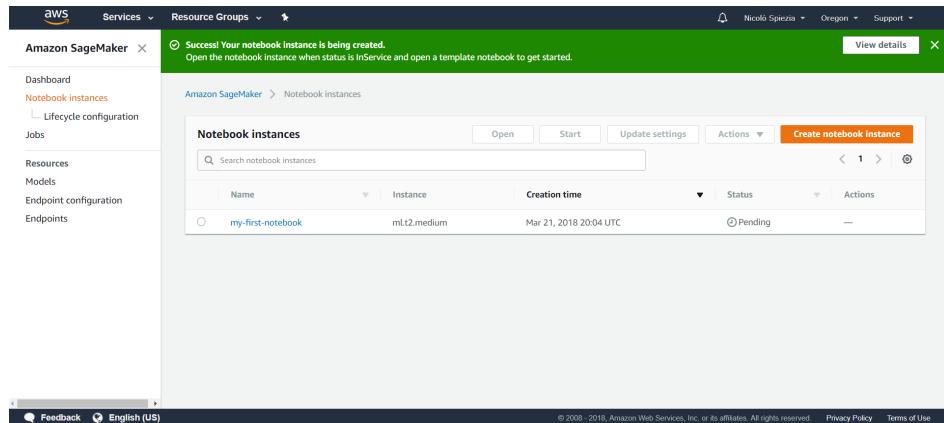
3. Click the button *Create notebook instance*.



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.



## 5. Start working on the newly created notebook



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:

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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.

```
In [3]: from keras.datasets import cifar10  
        from keras.utils import to_categorical
```

Using TensorFlow backend.

```
In [4]: 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:

```
In [5]: from keras.models import Sequential
        from keras.layers import Flatten, Dense
        from keras.layers import Conv2D, MaxPooling2D
        from time import time
```

```
In [6]: 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:

In [7]: `import tensorflow as tf`

And then let's create a model on the CPU:

In [8]: `with tf.device('cpu:0'):`  
 `model = convolutional_model()`

```

Defining convolutional model
0.067 seconds.
Compiling the model...
0.039 seconds.

```

Now let's train the CPU model for 2 epochs:

```
In [9]: 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 615us/step - loss: 2.0819 -
acc: 0.2793
Epoch 2/2
50000/50000 [=====] - 29s 587us/step - loss: 1.6724 -
acc: 0.4206
60.376140832901 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'):`

```
In [10]: with tf.device('gpu:0'):
    model = convolutional_model()

Defining convolutional model
0.063 seconds.
Compiling the model...
0.037 seconds.
```

And then we train the model on the GPU:

```
In [11]: 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 [=====] - 6s 111us/step - loss: 2.1398 -
acc: 0.2552
Epoch 2/2
50000/50000 [=====] - 4s 70us/step - loss: 1.7468 -
```

```
acc: 0.3874
9.329 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`:

```
In [12]: 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):

```
In [13]: with tf.device("/cpu:0"):
    model = convolutional_model()
```

```
Defining convolutional model
0.066 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):

```
In [14]: #adjust this to the number of gpus in your machine
        NGPU = 2
```

```
In [15]: model = multi_gpu_model(model, NGPU)
```

Once the model has been parallelized, we need to re-compile it:

```
In [16]: 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`.

```
In [17]: 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))
```

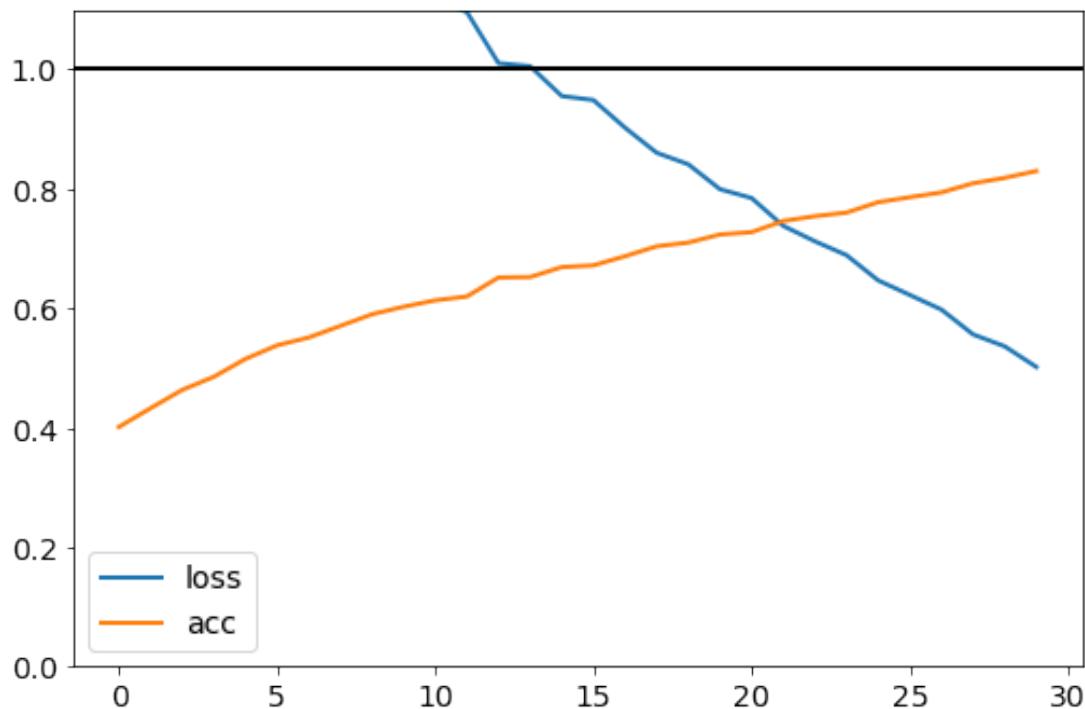
```
Training recurrent GPU model on 2 GPUs ...
Epoch 1/2
50000/50000 [=====] - 4s 84us/step - loss: 2.2684 -
acc: 0.2150
Epoch 2/2
50000/50000 [=====] - 2s 45us/step - loss: 1.9415 -
acc: 0.3312
6.800 seconds.
```

Since with 2 GPUs each epoch takes only a few seconds, let's run the training for a few more epochs:

```
In [18]: 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:

```
In [19]: pd.DataFrame(h.history).plot()
plt.ylim(0, 1.1)
plt.axhline(1, color='black');
```



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!

## Exercises

### 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.

# 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:

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())  
  
In [2]: with open('matplotlibconf.py') as fin:  
    exec(fin.read())
```

## 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:

```
In [3]: from sklearn.datasets import load_digits
```

Now let's create a variable called `digits` we'll fill as the result of calling `load_digits()`:

```
In [4]: digits = load_digits()
```

Then we assign `digits.data` and `digits.target` to `X` and `y` respectively:

```
In [5]: X, y = digits.data, digits.target
```

Let's look at the shape of the `X` data:

```
In [6]: X.shape
```

```
Out[6]: (1797, 64)
```

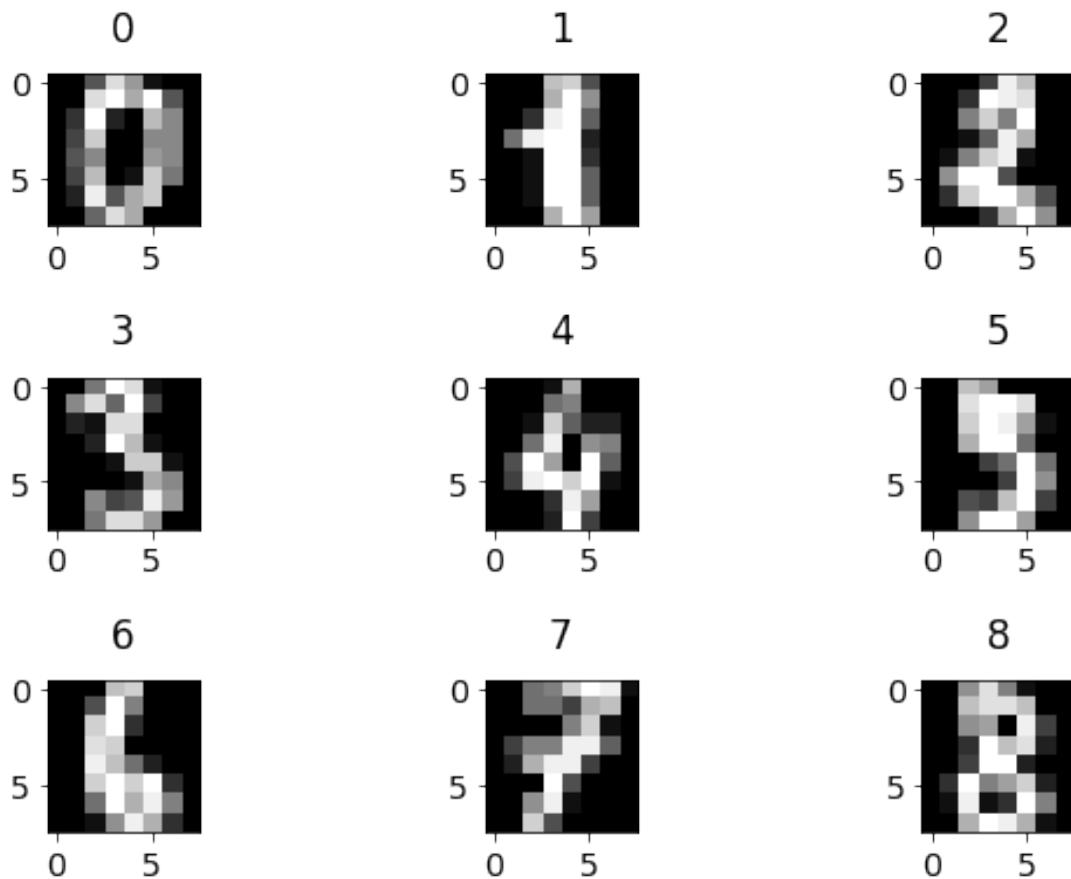
`X` is an array of 1797 images that have been unrolled as feature vectors of length 64.

```
In [7]: y.shape
```

```
Out[7]: (1797,)
```

In order to see the images we can always reshape them to the original 8x8 format. Let's plot a few digits:

```
In [8]: 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:

```
In [9]: 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.  
 :Missing Attribute Values: None  
 :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)  
 :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets  
<http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits>

The data set contains images of hand-written digits: 10 classes where each class refers to a digit.

Preprocessing programs made available by NIST were used to extract normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G. T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469, 1994.

## References

-----

- C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their Applications to Handwritten Digit Recognition, MSc Thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University.
- E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin. Linear dimensionality reduction using relevance weighted LDA. School of Electrical and Electronic Engineering Nanyang Technological University. 2005.
- Claudio Gentile. A New Approximate Maximal Margin Classification Algorithm. NIPS. 2000.

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:

In [10]: X.min()

Out[10]: 0.0

```
In [11]: X.max()
```

```
Out[11]: 16.0
```

Let's also check the data type of X:

```
In [12]: X.dtype
```

```
Out[12]: 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):

```
In [13]: X_sc = X / 16.0
```

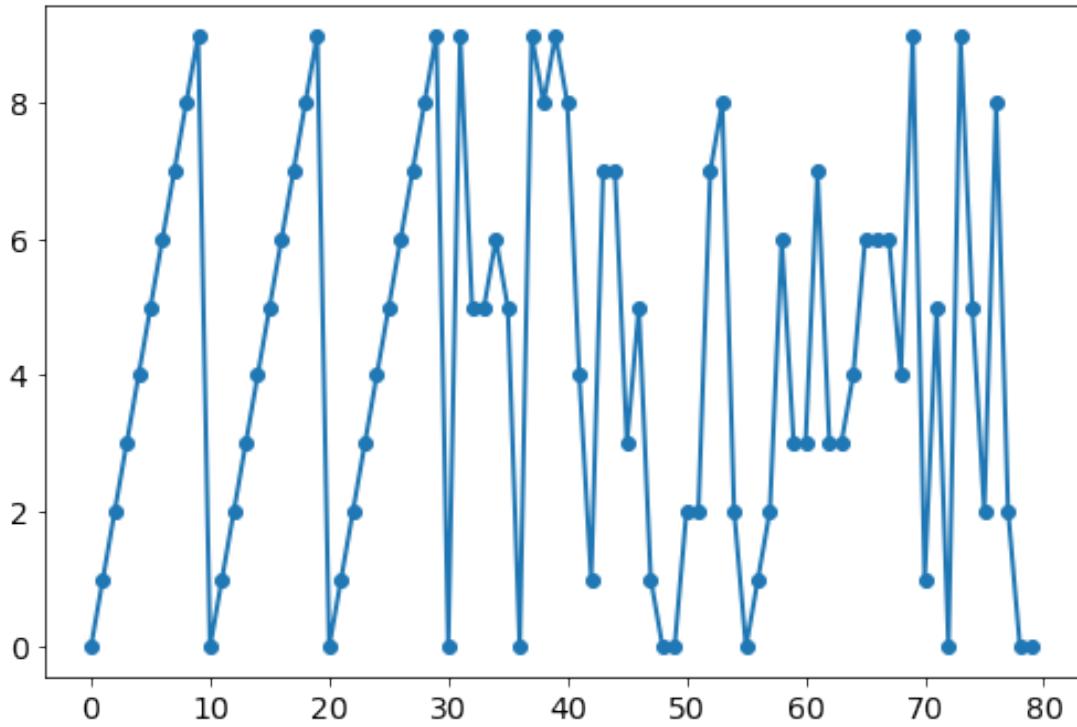
y contains the labels as a list of digits:

```
In [14]: y[:20]
```

```
Out[14]: 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:

```
In [15]: plt.plot(y[:80], 'o-');
```



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`:

```
In [16]: from keras.utils import to_categorical
```

Using TensorFlow backend.

Then let's set the variable of `y_cat` to these categories:

```
In [17]: 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:

```
In [18]: 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).

```
In [19]: 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`:

```
In [20]: y_test_classes = np.argmax(y_test, axis=1)
```

`y_test_classes` is an array of digits:

```
In [21]: y_test_classes
```

```
Out[21]: 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,
   3, 4, 1, 0, 7, 8, 7, 5, 0, 6, 0, 1, 8, 7, 0, 0, 3, 4, 8, 9, 4, 4,
   1, 1, 2, 1, 9, 2, 7, 7, 6, 9, 2, 9, 6, 0, 5, 2, 4, 4, 4, 6, 4, 0,
   1, 8, 3, 4, 0, 5, 9, 0, 2, 0, 0, 1, 3, 2, 8, 1, 6, 1, 1, 9, 2, 7,
   8, 3, 8, 2, 1, 3, 3, 0, 7, 8, 6, 7, 1, 4, 8, 2, 1, 4, 2, 6, 0, 6,
   0, 1, 0, 8, 0, 6, 5, 1, 6, 6, 9, 2, 9, 2, 8, 5, 9, 4, 3, 9, 2, 9,
   7, 9, 1, 3, 0, 3, 9, 2, 6, 1, 0, 0, 6, 3, 5, 0, 0, 3, 8, 0, 3, 0,
   7, 7, 6, 1, 8, 8, 7, 2, 7, 5, 8, 5, 3, 7, 8, 2, 5, 4, 5, 1, 5, 7,
   5, 6, 4, 0, 6, 7, 1, 1, 6, 4, 0, 4, 0, 1, 3, 4, 4, 4, 5, 4, 5, 5,
   4, 3, 7, 9, 1, 1, 4, 7, 2, 0, 2, 9, 7, 8, 4, 8, 2, 4, 8, 7, 9, 4,
   8, 0, 7, 0, 6, 5, 4, 2, 3, 5, 3, 5, 7, 7, 4, 1, 3, 0, 1, 1, 8, 6,
   5, 1, 8, 0, 0, 3, 7, 7, 4, 9, 0, 4, 6, 9, 0, 7, 9, 2, 9, 2, 9, 6,
   6, 5, 4, 5, 7, 3, 7, 7, 5, 2, 2, 7, 8, 9, 3, 3, 2, 6, 3, 6, 2, 1,
   7, 4, 8, 0, 8, 2, 4, 3, 7, 6, 3, 5, 7, 9, 3, 7, 9, 5, 3, 7, 7, 6,
   4, 8, 0, 8, 4, 6, 8, 4, 1, 7, 6, 5, 9, 3, 4, 5, 9, 8, 2, 3, 2, 5,
   6, 4, 9, 1, 5, 9, 8, 2, 6, 1, 3, 1, 0, 7, 5, 2, 8, 1, 5, 2, 2, 3,
   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:

```
In [22]: pd.Series(y_test_classes).value_counts()
```

Out [22] :

	o
5	55
3	55
1	55
9	54
7	54
6	54
4	54
0	54
2	53
8	52

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:

```
In [23]: 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:

```
In [24]: model = Sequential()
        model.add(Dense(16, input_shape=(64, ),
                      activation='relu'))
        model.add(Dense(10, activation='softmax'))

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

Let's also save the initial weights so that we can always re-start from the same initial configuration:

```
In [25]: initial_weights = model.get_weights()
```

Now we fit the model on the training data for 100 epochs:

```
In [26]: model.fit(X_train, y_train, epochs=100, verbose=0)
```

```
Out[26]: <keras.callbacks.History at 0x7fbc7b2f8ba8>
```

The model converged and we can evaluate the final training performance and test accuracies:

```
In [27]: _, 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:

```
In [28]: print("Train accuracy: {:.4f}".format(train_acc))
          print("Test accuracy: {:.4f}".format(test_acc))
```

```
Train accuracy: 0.9952
Test accuracy: 0.9741
```

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?



Do we need more data or a better model?

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.

```
In [29]: fracs = np.linspace(0.1, 0.90, 5)
fracs
```

```
Out[29]: array([0.1, 0.3, 0.5, 0.7, 0.9])
```

```
In [30]: train_sizes = list((len(X_train) * fracs).astype(int))
train_sizes
```

```
Out[30]: [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:

```
In [31]: train_scores = []
          test_scores = []
```

Now let's break up the test data using the `train_test_split` function as we usually would:

```
In [32]: 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:

```
In [33]: model.set_weights(initial_weights)
```

Now we can train our model using the `fit` function, as normal:

```
In [34]: 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:

```
In [35]: 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:

```
In [36]: 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:

```
In [37]: 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])

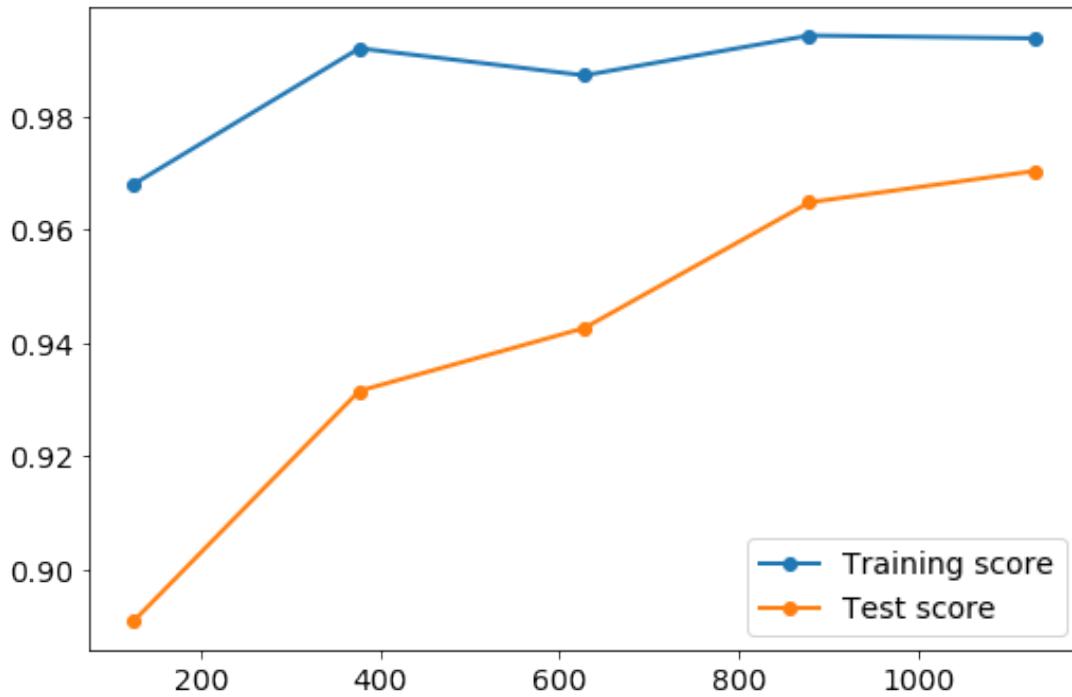
    e = model.evaluate(X_test, y_test, verbose=0)
    test_scores.append(e[-1])

    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:

```
In [38]: plt.plot(train_sizes, train_scores, 'o-', label="Training score")
plt.plot(train_sizes, test_scores, 'o-', label="Test score")
plt.legend(loc="best");
```



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:

```
In [39]: 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.

```
In [40]: 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 val accuracy to list
    histories.append([h.history['acc'],
                      h.history['val_acc']])

```

```

print(repeat, end=" ")

histories = np.array(histories)
print()

# calculate mean and std 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:

```
In [41]: 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:

```
In [42]: ((m_train_base, m_test_base),
      (s_train_base, s_test_base)) = \
      repeat_train(base_model, repeats=5)
```

```
0 1 2 3 4
```

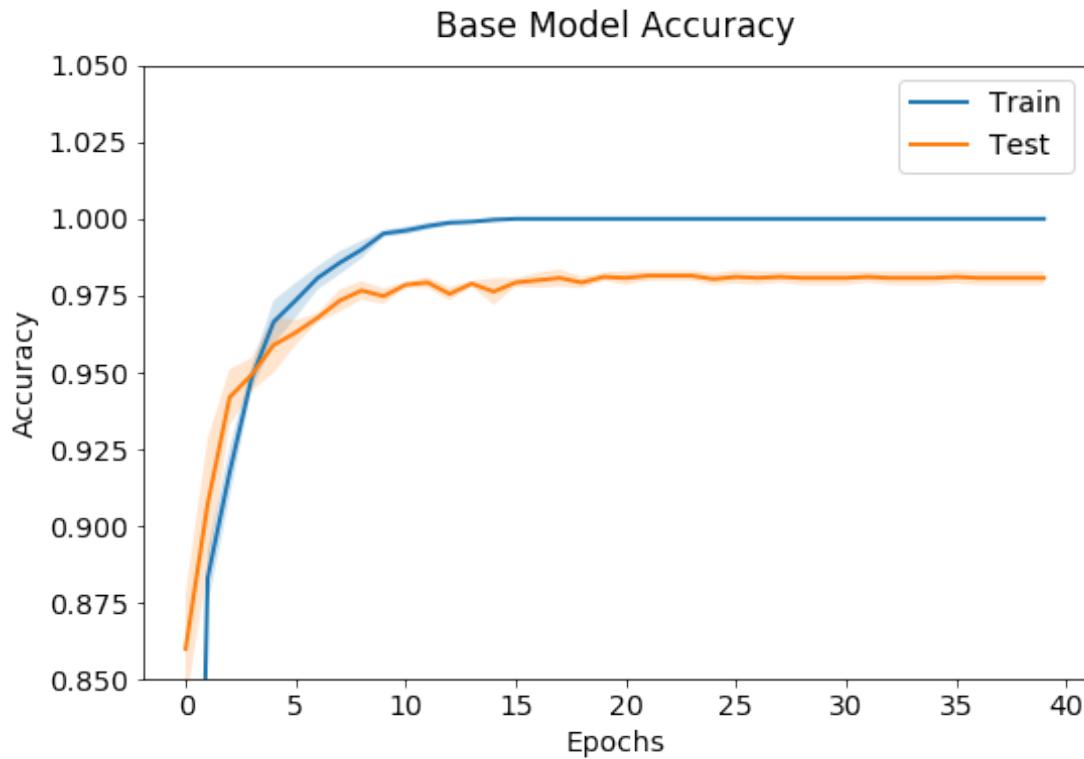
We can plot the histories for training and test. First, let's define an additional helper function `plot_mean_std()`, which plots the average history as a line and add a colored area around it corresponding to  $+/- 1$  standard deviation:

```
In [43]: 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:

```
In [44]: 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);
```



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) is a common procedure in Machine Learning and it has been used to improve the performance of complex models with many parameters.

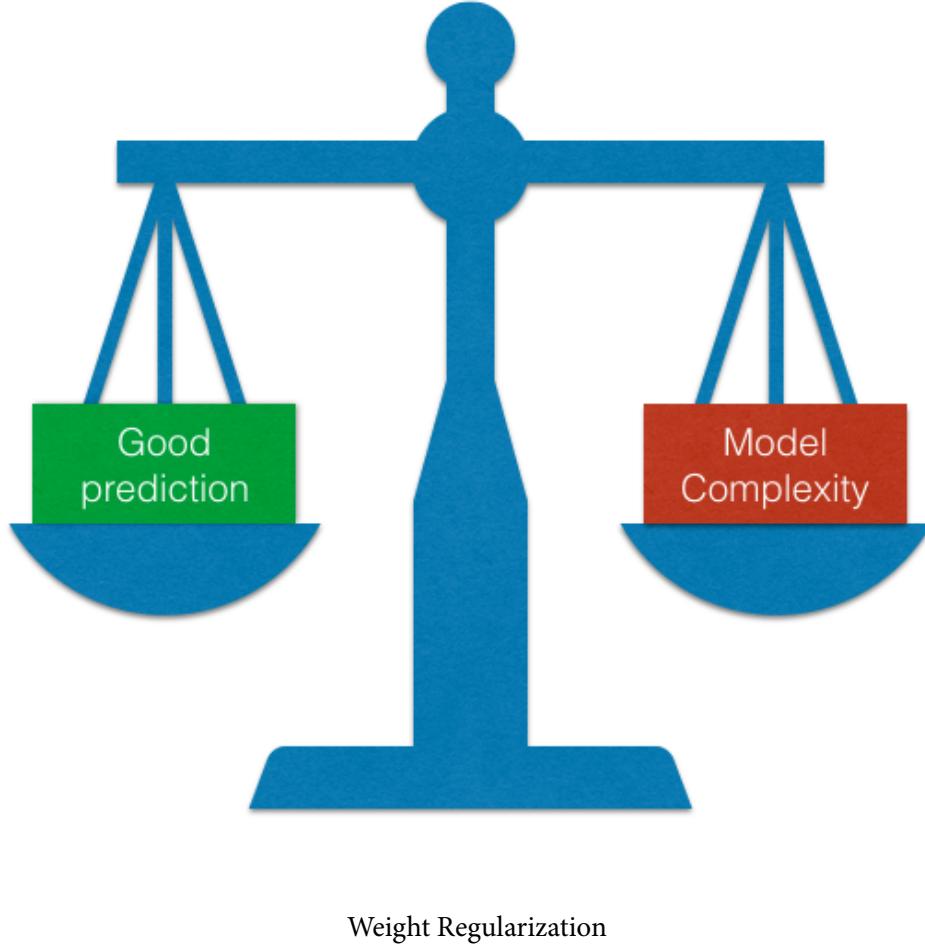
Remember the [Cost Function](#) we have introduced in [Chapter 3](#)? The main goal of the cost function is to make sure that the predictions of the model are close to the correct labels.

Regularization works by modifying the original cost function  $C$  with an additional term  $\lambda C_r$ , that somehow penalizes the complexity of the model:

$$C' = C + \lambda C_r \quad (10.1)$$

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  $\lambda$  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:

$$C_w = \sum_w |w| \quad (10.2)$$

or:

$$C_w = \sum_w w^2 \quad (10.3)$$

The first one is called **l1-regularization** and it is the sum of the absolute values of each weight. The second one is called **l2-regularization** and it is the sum of the square values of each weight. While they both suppress complexity, their effect is different.

l1-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 l1-regularization is to make the weight matrix *sparse*.

l2-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:

```
In [45]: from keras.regularizers import l2
```

```
In [46]: def regularized_model():
    """
    Return an l2-weight-regularized, fully connected
    model with 3 inner layers with 1024 nodes each
    and relu activation function.
    """
    reg = l2(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 l<sub>2</sub>-regularization. Let's repeat the training 3 times.

```
In [47]: (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:

```

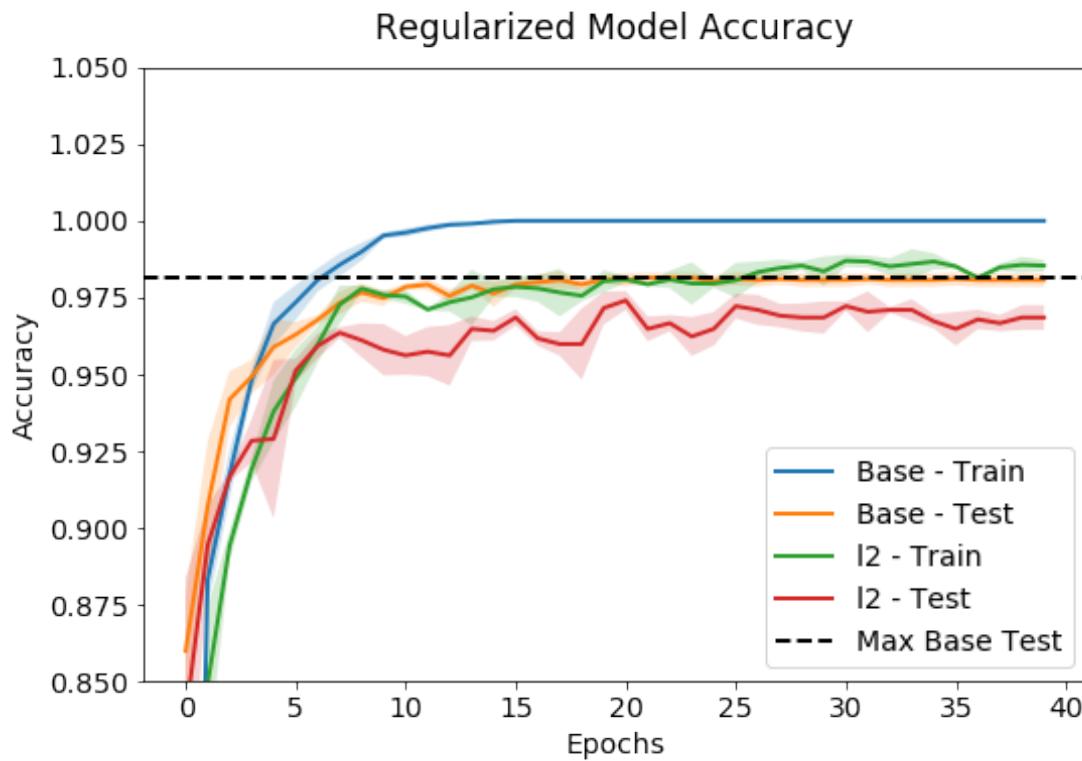
In [48]: plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

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',
           'Max Base Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim(0.85, 1.05);

```



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.

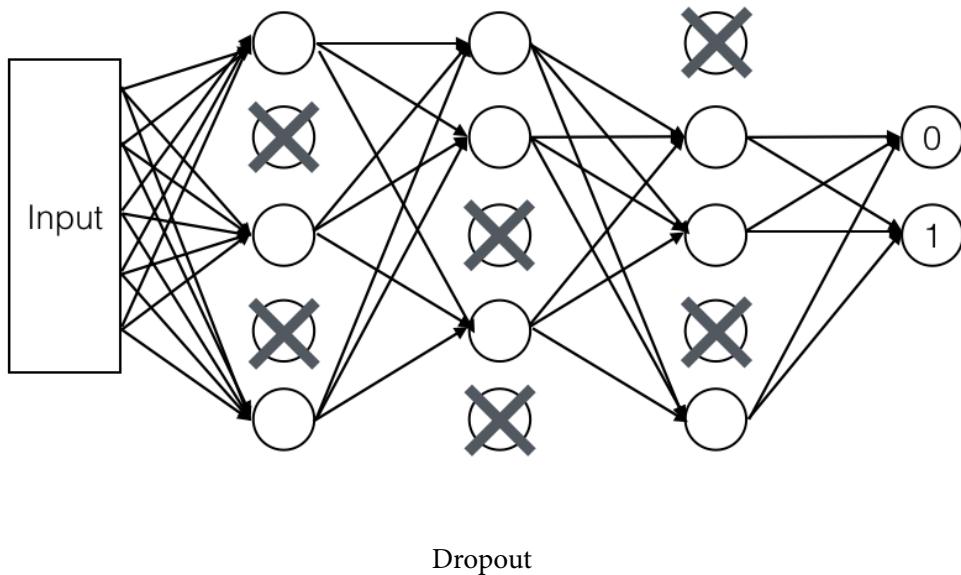
It's however good to know that this technique exists and keep it in mind as one of the options to try. In practice, weight regularization has been superseded by more modern regularization techniques such as **Dropout** and **Batch Normalization**.

## Dropout

[Dropout]([https://en.wikipedia.org/wiki/Dropout\\_\(neural\\_networks\)](https://en.wikipedia.org/wiki/Dropout_(neural_networks))) 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 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:

```
In [49]: 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.

```
In [50]: 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:

```
In [51]: (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:

```
In [52]: 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)

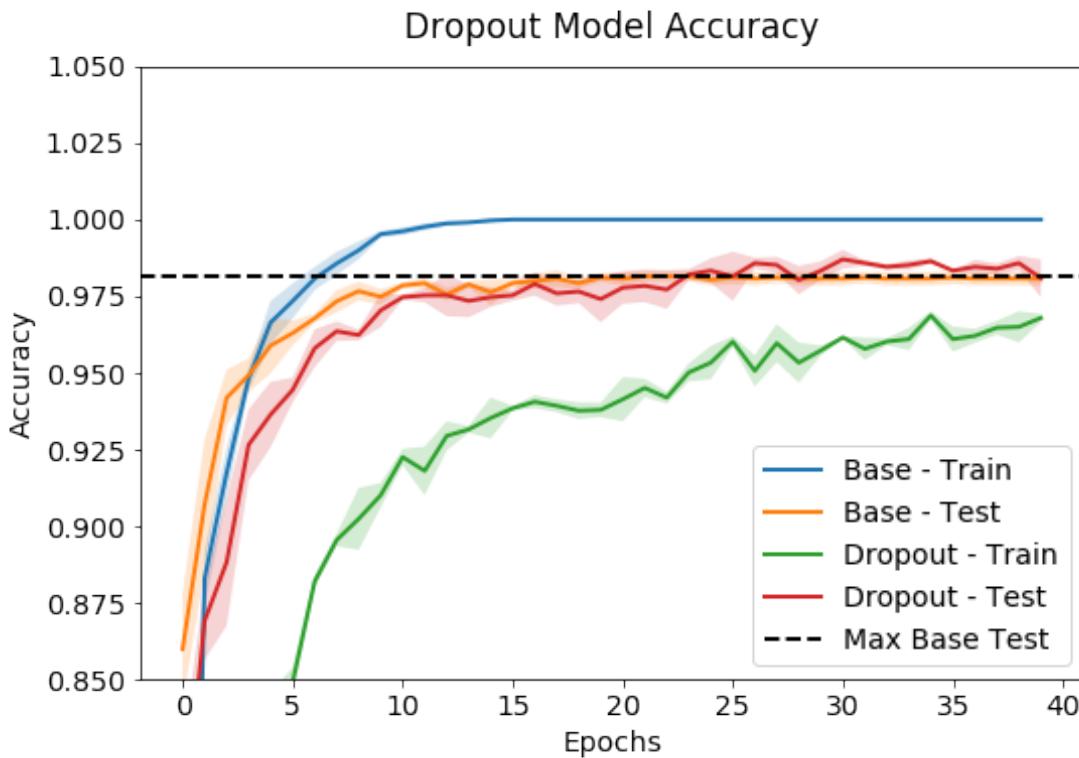
plt.axhline(m_test_base.max(),
            linestyle='dashed',
            color='black')

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);

```



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:

```
In [53]: 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:

```
In [54]: 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:

```
In [55]: (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:

```
In [56]: plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

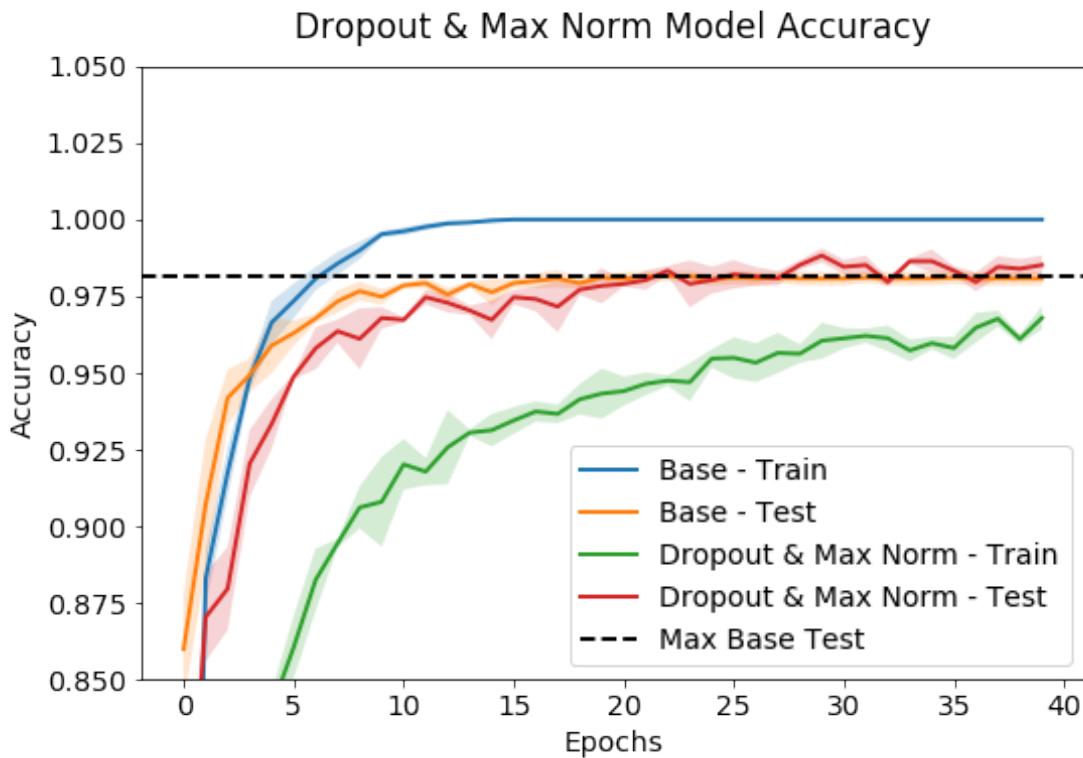
plot_mean_std(m_train_dmn, s_train_dmn)
plot_mean_std(m_test_dmn, s_test_dmn)
```

```

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);

```



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  $\gamma$  and  $\beta$  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  $B = \{x_{1\dots m}\}$

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad (10.4)$$

$$\sigma_B = \sqrt{\frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2} \quad (10.5)$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (10.6)$$

$$y_i = \gamma \hat{x}_i + \beta \quad (10.7)$$

$$\quad \quad \quad (10.8)$$

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:

```
In [57]: 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`:

```
In [58]: 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.

```
In [59]: (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
```

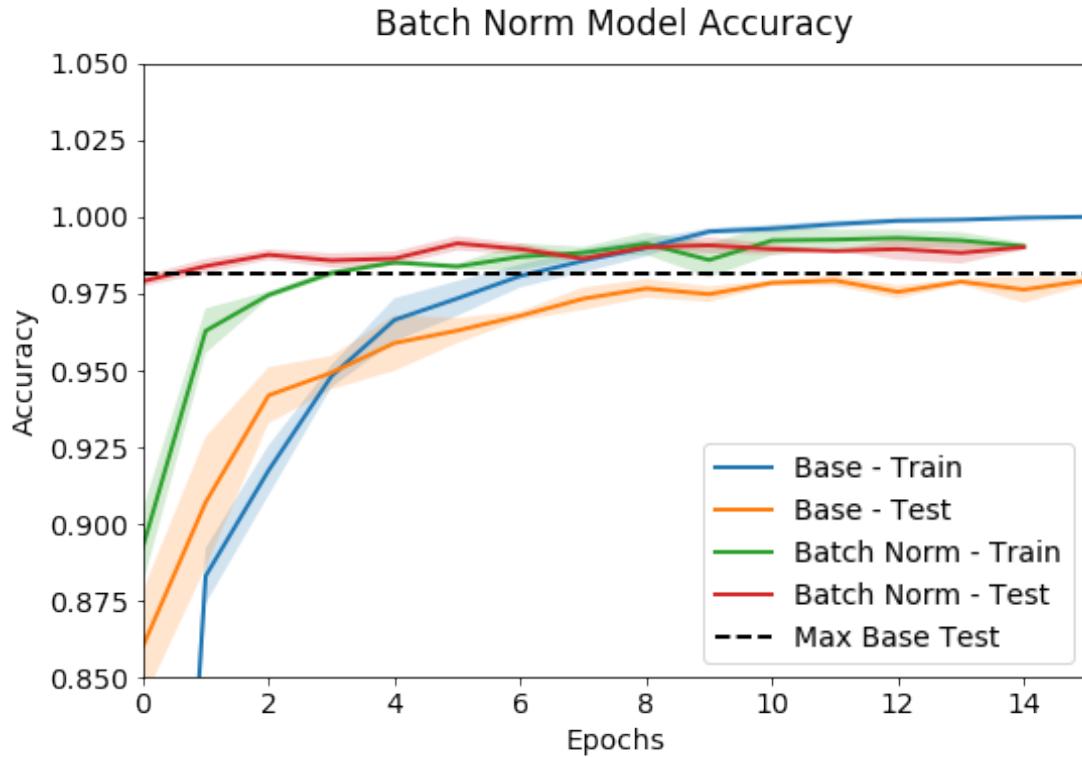
Let's plot the results and compare with the base model:

```
In [60]: 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);
```



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:

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



Image of a squirrel

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:

```
In [61]: 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  $\pm 20$  degrees - We'll set the `shear_range` to  $\pm 0.3$  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.

```
In [62]: 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.

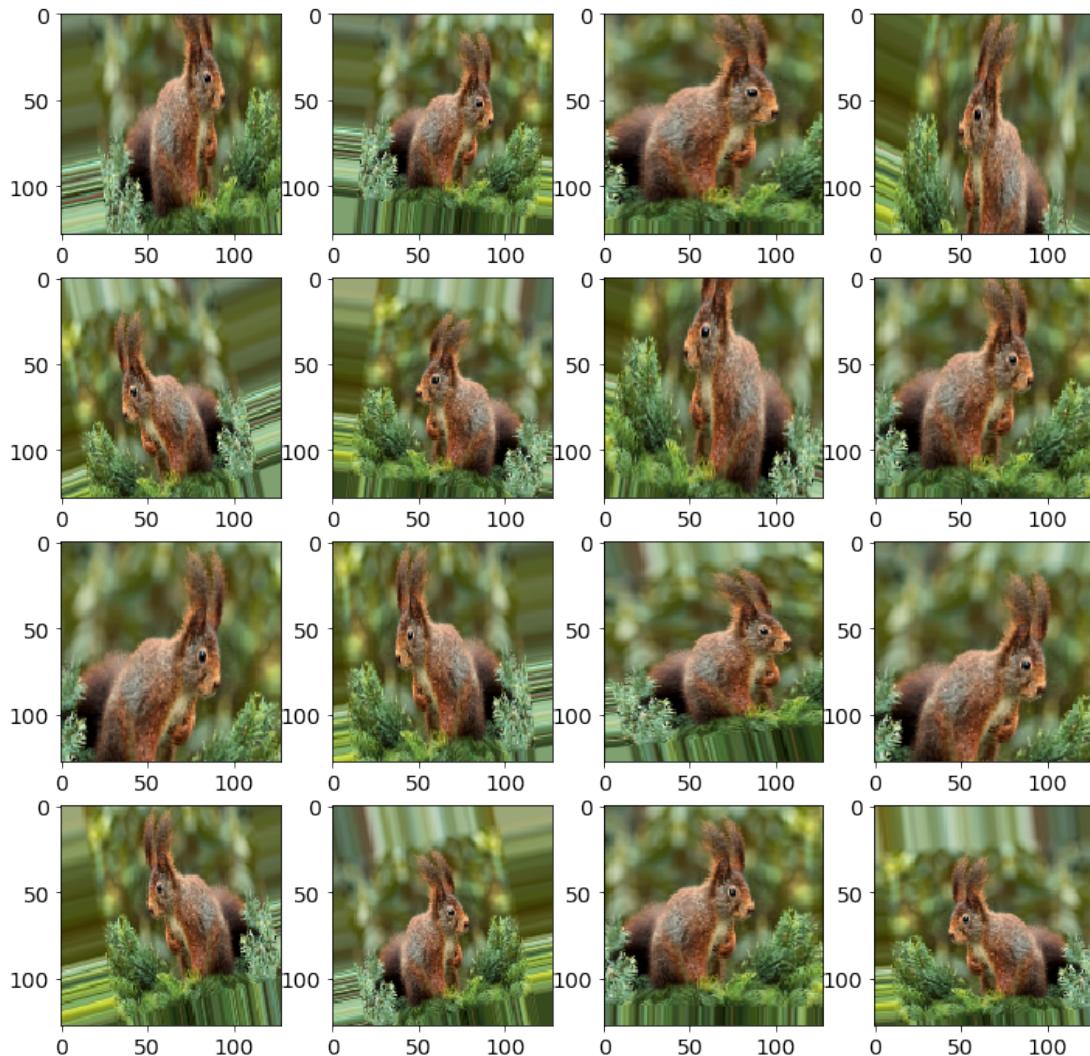
```
In [63]: train_gen = idg.flow_from_directory(
            '../data/generator',
            target_size = (128, 128),
            class_mode = 'binary')
```

Found 1 images belonging to 1 classes.

Let's generate a few images and display them:

```
In [64]: 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.

```
In [65]: 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:

```
In [66]: 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.

```
In [67]: 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:

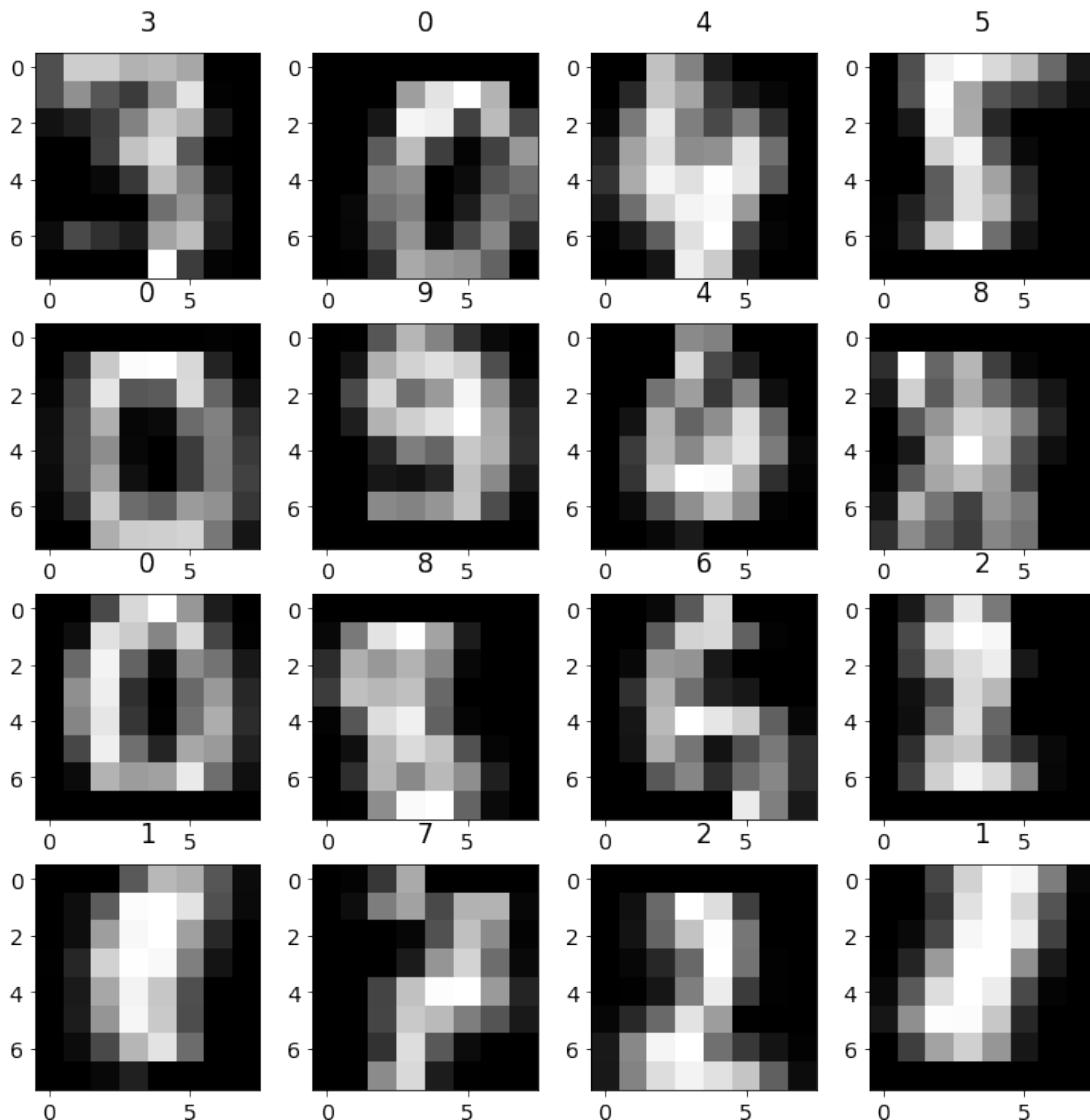
```
In [68]: imgs, labels = train_gen.next()
```

```
In [69]: imgs.shape
```

```
Out[69]: (32, 8, 8, 1)
```

Let's display a few of them:

```
In [70]: 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:

```
In [71]: from keras.layers import Flatten
```

```
In [72]: 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.

```
In [73]: 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 val accuracy to list
    histories.append([h.history['acc'],
```

```

        h.history['val_acc'])
print(repeat, end=" ")

histories = np.array(histories)
print()

# calculate mean and std 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:

```
In [74]: (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:

```

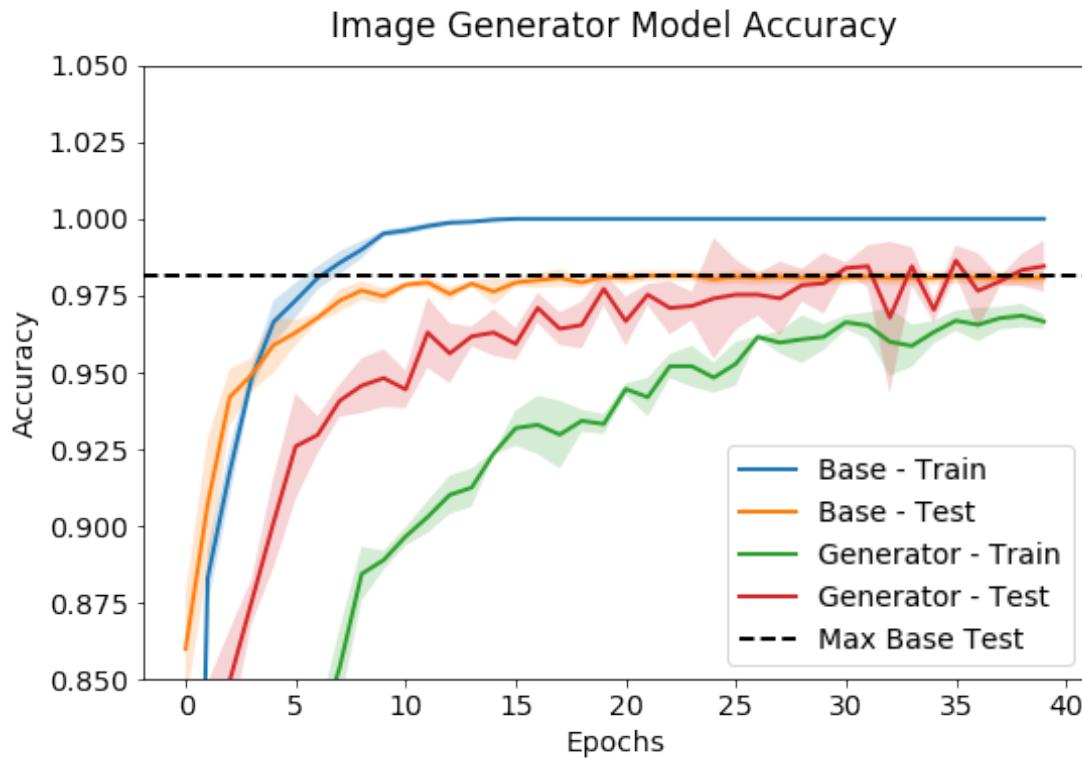
In [75]: plot_mean_std(m_train_base, s_train_base)
plot_mean_std(m_test_base, s_test_base)

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);

```



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  $64 \times 64$  as input and return a binary output ( $\text{female}=0/\text{male}=1$ )
- 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  $64 \times 64$ .
- 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](#)

# 11

## Pretrained 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](#), 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](#) on [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, you 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 the common files:

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:
    exec(fin.read())
```

## 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:

```
In [3]: df = pd.read_csv('../data/sports.csv')
```

and let's inspect it with the `.head()` command:

```
In [4]: df.head()
```

Out [4] :

	image_url	class	label:confidence
0	https://multimedia-commons.s3-us-west-2.amazonaws.com/1003/1003_001.jpg	Formula racing	1.0000
1	https://multimedia-commons.s3-us-west-2.amazonaws.com/1003/1003_002.jpg	Cross-country skiing	0.9594
2	https://multimedia-commons.s3-us-west-2.amazonaws.com/1003/1003_003.jpg	Formula racing	1.0000
3	https://multimedia-commons.s3-us-west-2.amazonaws.com/1003/1003_004.jpg	Formula racing	1.0000
4	https://multimedia-commons.s3-us-west-2.amazonaws.com/1003/1003_005.jpg	Formula racing	0.6646

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:

```
In [5]: df['class'].value_counts()
```

Out [5] :

	class
Cross-country skiing	1003
Beach volleyball	1002
Formula racing	1001

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:

```
In [6]: 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.

```
In [7]: 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:

```
In [8]: import os
```

We also load the `urlretrieve` (from the `urllib.request` library) function that allows us to download an image from a url:

```
In [9]: from urllib.request import urlretrieve
```

Then let's define a `maybe_download_image` function:

```
In [10]: 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
    """

    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: Path where images are saved*

```

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:

```
In [11]: image_url = df['image_url'][0]
image_url
```

```
Out[11]: 'https://multimedia-commons.s3-us-west-2.amazonaws.com/data/images/7ad/a7b/7ada7b21d671
```

and the first label:

```
In [12]: label = df['class'][0]
label
```

```
Out[12]: 'Formula racing'
```

Now let's download the image using our helper function:

```
In [13]: maybe_download_image('/tmp/ztdlbook/', image_url, label)
```

```
Out[13]: 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:

```
In [14]: maybe_download_image('/tmp/ztdlbook/', image_url, label)
```

```
Out[14]: 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:

```
In [15]: from concurrent.futures import ThreadPoolExecutor
        from concurrent.futures import 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.

```
In [16]: def get_images(save_dir, image_urls,
                  image_labels, max_workers=20):
    """
    Download list of labeled images using threads.

    Args:
        save_dir: Path where images are saved
        image_urls: A list of image urls
        image_labels: A list of image labels
        max_workers: Concurrent threads (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 = {}
        for url, label in zip(image_urls, image_labels):
            k = executor.submit(maybe_download_image,
                                save_dir, url, label)
            future_to_url[k] = url

        # 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 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`:

```
In [17]: train_path = '../data/sports/train/'
```

```
In [18]: 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`:

```
In [19]: test_path = '../data/sports/test/'
```

```
In [20]: 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	1.0
VGG16	528	0.715	0.901	138	23	6.5	1.0
VGG19	549	0.727	0.910	144	26	7.3	1.1
ResNet50	99	0.759	0.929	26	168	10.9	1.3
InceptionV3	92	0.788	0.944	24	159	17.4	2.0
InceptionResNetV2	215	0.804	0.953	56	572	47.6	3.6
MobileNet	17	0.665	0.871	4	88	9.7	2.5
DenseNet121	33	0.745	0.918	8	121	32.6	4.4
DenseNet169	57	0.759	0.928	14	169	57.6	5.9
DenseNet201	80	0.770	0.933	20	201	80.3	8.6

## Keras pre-trained models comparison table

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:

- 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:

```
In [21]: from keras.preprocessing import image
```

```
Using TensorFlow backend.
```

Now let's load an image from the ones we have downloaded previously. Let's define the `input_path`:

```
In [22]: dir_ = '../data/sports/train/Beach volleyball/'  
fname_ = '1d8a1f53f36487ac4f10e47e6937308.jpg'  
input_path = dir_ + fname_
```

and then load the image:

```
In [23]: img = image.load_img(input_path, target_size=(299, 299))
```

Jupyter notebook can display images inline, so let's have a look at it:

```
In [24]: img
```

```
Out[24] :
```



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.

```
In [25]: type(img)
```

```
Out[25]: 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:

```
In [26]: 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:

```
In [27]: img_array.shape
```

```
Out[27]: (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:

```
In [28]: 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:

```
In [29]: img_tensor.shape
```

```
Out[29]: (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:

```
In [30]: 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.

```
In [31]: model = Xception(weights='imagenet')
```

Let's have a look at the model architecture by printing the summary:

```
In [32]: model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
--------------	--------------	---------	--------------

```
=====
=====
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]
-----
block1_conv1_act (Activation)  (None, None, None, 3 0
block1_conv1_bn[0][0]
-----
block1_conv2 (Conv2D)      (None, None, None, 6 18432
block1_conv1_act[0][0]
-----
block1_conv2_bn (BatchNormaliza (None, None, None, 6 256
block1_conv2[0][0]
-----
block1_conv2_act (Activation)  (None, None, None, 6 0
block1_conv2_bn[0][0]
-----
block2_sepconv1 (SeparableConv2 (None, None, None, 1 8768
block1_conv2_act[0][0]
-----
block2_sepconv1_bn (BatchNormal (None, None, None, 1 512
block2_sepconv1[0][0]
-----
block2_sepconv2_act (Activation (None, None, None, 1 0
block2_sepconv1_bn[0][0]
-----
block2_sepconv2 (SeparableConv2 (None, None, None, 1 17536
block2_sepconv2_act[0][0]
-----
block2_sepconv2_bn (BatchNormal (None, None, None, 1 512
block2_sepconv2[0][0]
-----
conv2d_1 (Conv2D)          (None, None, None, 1 8192
block1_conv2_act[0][0]
-----
block2_pool (MaxPooling2D)   (None, None, None, 1 0
block2_sepconv2_bn[0][0]
-----
batch_normalization_1 (BatchNor (None, None, None, 1 512      conv2d_1[0][0]
-----
```

```
add_1 (Add)           (None, None, None, 1 0
block2_pool[0] [0]
batch_normalization_1[0] [0]
-----
block3_sepconv1_act (Activation (None, None, None, 1 0           add_1[0] [0]
-----
block3_sepconv1 (SeparableConv2 (None, None, None, 2 33920
block3_sepconv1_act[0] [0]
-----
block3_sepconv1_bn (BatchNormal (None, None, None, 2 1024
block3_sepconv1[0] [0]
-----
block3_sepconv2_act (Activation (None, None, None, 2 0           block3_sepconv1_bn[0] [0]
-----
block3_sepconv2 (SeparableConv2 (None, None, None, 2 67840
block3_sepconv2_act[0] [0]
-----
block3_sepconv2_bn (BatchNormal (None, None, None, 2 1024
block3_sepconv2[0] [0]
-----
conv2d_2 (Conv2D)           (None, None, None, 2 32768           add_1[0] [0]
-----
block3_pool (MaxPooling2D)   (None, None, None, 2 0
block3_sepconv2_bn[0] [0]
-----
batch_normalization_2 (BatchNor (None, None, None, 2 1024           conv2d_2[0] [0]
-----
add_2 (Add)           (None, None, None, 2 0
block3_pool[0] [0]
batch_normalization_2[0] [0]
-----
block4_sepconv1_act (Activation (None, None, None, 2 0           add_2[0] [0]
-----
block4_sepconv1 (SeparableConv2 (None, None, None, 7 188672
block4_sepconv1_act[0] [0]
-----
block4_sepconv1_bn (BatchNormal (None, None, None, 7 2912
block4_sepconv1[0] [0]
-----
block4_sepconv2_act (Activation (None, None, None, 7 0           block4_sepconv1_bn[0] [0]
-----
block4_sepconv2 (SeparableConv2 (None, None, None, 7 536536
```

```
block4_sepconv2_act[0][0]
-----
block4_sepconv2_bn (BatchNormal (None, None, None, 7 2912
block4_sepconv2[0][0]
-----
conv2d_3 (Conv2D)           (None, None, None, 7 186368      add_2[0][0]
-----
block4_pool (MaxPooling2D)   (None, None, None, 7 0
block4_sepconv2_bn[0][0]
-----
batch_normalization_3 (BatchNor (None, None, None, 7 2912      conv2d_3[0][0]
-----
add_3 (Add)                 (None, None, None, 7 0
block4_pool[0][0]
batch_normalization_3[0][0]
-----
block5_sepconv1_act (Activation (None, None, None, 7 0          add_3[0][0]
-----
block5_sepconv1 (SeparableConv2 (None, None, None, 7 536536
block5_sepconv1_act[0][0]
-----
block5_sepconv1_bn (BatchNormal (None, None, None, 7 2912
block5_sepconv1[0][0]
-----
block5_sepconv2_act (Activation (None, None, None, 7 0
block5_sepconv1_bn[0][0]
-----
block5_sepconv2 (SeparableConv2 (None, None, None, 7 536536
block5_sepconv2_act[0][0]
-----
block5_sepconv2_bn (BatchNormal (None, None, None, 7 2912
block5_sepconv2[0][0]
-----
block5_sepconv3_act (Activation (None, None, None, 7 0
block5_sepconv2_bn[0][0]
-----
block5_sepconv3 (SeparableConv2 (None, None, None, 7 536536
block5_sepconv3_act[0][0]
-----
block5_sepconv3_bn (BatchNormal (None, None, None, 7 2912
block5_sepconv3[0][0]
-----
add_4 (Add)                 (None, None, None, 7 0
block5_sepconv3_bn[0][0]
```

```
add_3[0][0]
-----
block6_sepconv1_act (Activation (None, None, None, 7 0           add_4[0][0]
-----
block6_sepconv1 (SeparableConv2 (None, None, None, 7 536536
block6_sepconv1_act[0][0]
-----
block6_sepconv1_bn (BatchNormal (None, None, None, 7 2912
block6_sepconv1[0][0]
-----
block6_sepconv2_act (Activation (None, None, None, 7 0
block6_sepconv1_bn[0][0]
-----
block6_sepconv2 (SeparableConv2 (None, None, None, 7 536536
block6_sepconv2_act[0][0]
-----
block6_sepconv2_bn (BatchNormal (None, None, None, 7 2912
block6_sepconv2[0][0]
-----
block6_sepconv3_act (Activation (None, None, None, 7 0
block6_sepconv2_bn[0][0]
-----
block6_sepconv3 (SeparableConv2 (None, None, None, 7 536536
block6_sepconv3_act[0][0]
-----
block6_sepconv3_bn (BatchNormal (None, None, None, 7 2912
block6_sepconv3[0][0]
-----
add_5 (Add)           (None, None, None, 7 0
block6_sepconv3_bn[0][0]           add_4[0][0]
-----
block7_sepconv1_act (Activation (None, None, None, 7 0           add_5[0][0]
-----
block7_sepconv1 (SeparableConv2 (None, None, None, 7 536536
block7_sepconv1_act[0][0]
-----
block7_sepconv1_bn (BatchNormal (None, None, None, 7 2912
block7_sepconv1[0][0]
-----
block7_sepconv2_act (Activation (None, None, None, 7 0
block7_sepconv1_bn[0][0]
-----
block7_sepconv2 (SeparableConv2 (None, None, None, 7 536536
```

```
block7_sepconv2_act[0][0]
-----
block7_sepconv2_bn (BatchNormal (None, None, None, 7 2912
block7_sepconv2[0][0]
-----
block7_sepconv3_act (Activation (None, None, None, 7 0
block7_sepconv2_bn[0][0]
-----
block7_sepconv3 (SeparableConv2 (None, None, None, 7 536536
block7_sepconv3_act[0][0]
-----
block7_sepconv3_bn (BatchNormal (None, None, None, 7 2912
block7_sepconv3[0][0]
-----
add_6 (Add) (None, None, None, 7 0
block7_sepconv3_bn[0][0] add_5[0][0]
-----
block8_sepconv1_act (Activation (None, None, None, 7 0 add_6[0][0]
-----
block8_sepconv1 (SeparableConv2 (None, None, None, 7 536536
block8_sepconv1_act[0][0]
-----
block8_sepconv1_bn (BatchNormal (None, None, None, 7 2912
block8_sepconv1[0][0]
-----
block8_sepconv2_act (Activation (None, None, None, 7 0
block8_sepconv1_bn[0][0]
-----
block8_sepconv2 (SeparableConv2 (None, None, None, 7 536536
block8_sepconv2_act[0][0]
-----
block8_sepconv2_bn (BatchNormal (None, None, None, 7 2912
block8_sepconv2[0][0]
-----
block8_sepconv3_act (Activation (None, None, None, 7 0
block8_sepconv2_bn[0][0]
-----
block8_sepconv3 (SeparableConv2 (None, None, None, 7 536536
block8_sepconv3_act[0][0]
-----
block8_sepconv3_bn (BatchNormal (None, None, None, 7 2912
block8_sepconv3[0][0]
```

```
add_7 (Add)           (None, None, None, 7 0
block8_sepconv3_bn[0] [0]
                                add_6[0][0]
-----
block9_sepconv1_act (Activation (None, None, None, 7 0           add_7[0][0]
-----
block9_sepconv1 (SeparableConv2 (None, None, None, 7 536536
block9_sepconv1_act[0][0]
-----
block9_sepconv1_bn (BatchNormal (None, None, None, 7 2912
block9_sepconv1[0][0]
-----
block9_sepconv2_act (Activation (None, None, None, 7 0
block9_sepconv1_bn[0][0]
-----
block9_sepconv2 (SeparableConv2 (None, None, None, 7 536536
block9_sepconv2_act[0][0]
-----
block9_sepconv2_bn (BatchNormal (None, None, None, 7 2912
block9_sepconv2[0][0]
-----
block9_sepconv3_act (Activation (None, None, None, 7 0
block9_sepconv2_bn[0][0]
-----
block9_sepconv3 (SeparableConv2 (None, None, None, 7 536536
block9_sepconv3_act[0][0]
-----
block9_sepconv3_bn (BatchNormal (None, None, None, 7 2912
block9_sepconv3[0][0]
-----
add_8 (Add)           (None, None, None, 7 0
block9_sepconv3_bn[0][0]
                                add_7[0][0]
-----
block10_sepconv1_act (Activatio (None, None, None, 7 0           add_8[0][0]
-----
block10_sepconv1 (SeparableConv (None, None, None, 7 536536
block10_sepconv1_act[0][0]
-----
block10_sepconv1_bn (BatchNorma (None, None, None, 7 2912
block10_sepconv1[0][0]
-----
block10_sepconv2_act (Activatio (None, None, None, 7 0
block10_sepconv1_bn[0][0]
```

```
-----  
block10_sepconv2 (SeparableConv (None, None, None, 7 536536  
block10_sepconv2_act[0] [0]  
-----  
-----  
block10_sepconv2_bn (BatchNorma (None, None, None, 7 2912  
block10_sepconv2[0] [0]  
-----  
-----  
block10_sepconv3_act (Activatio (None, None, None, 7 0  
block10_sepconv2_bn[0] [0]  
-----  
-----  
block10_sepconv3 (SeparableConv (None, None, None, 7 536536  
block10_sepconv3_act[0] [0]  
-----  
-----  
block10_sepconv3_bn (BatchNorma (None, None, None, 7 2912  
block10_sepconv3[0] [0]  
-----  
-----  
add_9 (Add) (None, None, None, 7 0  
block10_sepconv3_bn[0] [0] add_8[0] [0]  
-----  
-----  
block11_sepconv1_act (Activatio (None, None, None, 7 0 add_9[0] [0]  
-----  
-----  
block11_sepconv1 (SeparableConv (None, None, None, 7 536536  
block11_sepconv1_act[0] [0]  
-----  
-----  
block11_sepconv1_bn (BatchNorma (None, None, None, 7 2912  
block11_sepconv1[0] [0]  
-----  
-----  
block11_sepconv2_act (Activatio (None, None, None, 7 0  
block11_sepconv1_bn[0] [0]  
-----  
-----  
block11_sepconv2 (SeparableConv (None, None, None, 7 536536  
block11_sepconv2_act[0] [0]  
-----  
-----  
block11_sepconv2_bn (BatchNorma (None, None, None, 7 2912  
block11_sepconv2[0] [0]  
-----  
-----  
block11_sepconv3_act (Activatio (None, None, None, 7 0  
block11_sepconv2_bn[0] [0]  
-----  
-----  
block11_sepconv3 (SeparableConv (None, None, None, 7 536536  
block11_sepconv3_act[0] [0]  
-----  
-----  
block11_sepconv3_bn (BatchNorma (None, None, None, 7 2912  
block11_sepconv3[0] [0]
```

```
-----  
add_10 (Add)           (None, None, None, 7 0  
block11_sepconv3_bn[0][0]      add_9[0][0]  
-----  
  
block12_sepconv1_act (Activatio (None, None, None, 7 0      add_10[0][0]  
-----  
  
block12_sepconv1 (SeparableConv (None, None, None, 7 536536  
block12_sepconv1_act[0][0]  
-----  
  
block12_sepconv1_bn (BatchNorma (None, None, None, 7 2912  
block12_sepconv1[0][0]  
-----  
  
block12_sepconv2_act (Activatio (None, None, None, 7 0  
block12_sepconv1_bn[0][0]  
-----  
  
block12_sepconv2 (SeparableConv (None, None, None, 7 536536  
block12_sepconv2_act[0][0]  
-----  
  
block12_sepconv2_bn (BatchNorma (None, None, None, 7 2912  
block12_sepconv2[0][0]  
-----  
  
block12_sepconv3_act (Activatio (None, None, None, 7 0  
block12_sepconv2_bn[0][0]  
-----  
  
block12_sepconv3 (SeparableConv (None, None, None, 7 536536  
block12_sepconv3_act[0][0]  
-----  
  
block12_sepconv3_bn (BatchNorma (None, None, None, 7 2912  
block12_sepconv3[0][0]  
-----  
  
add_11 (Add)           (None, None, None, 7 0  
block12_sepconv3_bn[0][0]      add_10[0][0]  
-----  
  
block13_sepconv1_act (Activatio (None, None, None, 7 0      add_11[0][0]  
-----  
  
block13_sepconv1 (SeparableConv (None, None, None, 7 536536  
block13_sepconv1_act[0][0]  
-----  
  
block13_sepconv1_bn (BatchNorma (None, None, None, 7 2912  
block13_sepconv1[0][0]  
-----  
  
block13_sepconv2_act (Activatio (None, None, None, 7 0
```

```
block13_sepconv1_bn[0] [0]
-----
block13_sepconv2 (SeparableConv (None, None, None, 1 752024
block13_sepconv2_act[0] [0]
-----
block13_sepconv2_bn (BatchNorma (None, None, None, 1 4096
block13_sepconv2[0] [0]
-----
conv2d_4 (Conv2D)           (None, None, None, 1 745472      add_11[0] [0]
-----
block13_pool (MaxPooling2D)  (None, None, None, 1 0
block13_sepconv2_bn[0] [0]
-----
batch_normalization_4 (BatchNor (None, None, None, 1 4096      conv2d_4[0] [0]
-----
add_12 (Add)               (None, None, None, 1 0
block13_pool[0] [0]
batch_normalization_4[0] [0]
-----
block14_sepconv1 (SeparableConv (None, None, None, 1 1582080      add_12[0] [0]
-----
block14_sepconv1_bn (BatchNorma (None, None, None, 1 6144
block14_sepconv1[0] [0]
-----
block14_sepconv1_act (Activatio (None, None, None, 1 0
block14_sepconv1_bn[0] [0]
-----
block14_sepconv2 (SeparableConv (None, None, None, 2 3159552
block14_sepconv1_act[0] [0]
-----
block14_sepconv2_bn (BatchNorma (None, None, None, 2 8192
block14_sepconv2[0] [0]
-----
block14_sepconv2_act (Activatio (None, None, None, 2 0
block14_sepconv2_bn[0] [0]
-----
avg_pool (GlobalAveragePooling2 (None, 2048)          0
block14_sepconv2_act[0] [0]
=====
predictions (Dense)           (None, 1000)            2049000      avg_pool[0] [0]
=====
=====
Total params: 22,910,480
Trainable params: 22,855,952
Non-trainable params: 54,528
```

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:

```
In [33]: 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.

```
In [34]: 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:

```
In [35]: 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:

```
In [36]: preds.shape
```

```
Out[36]: (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:

```
In [37]: 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.

```
In [38]: decode_predictions(preds, top=3)[0]
```

```
Out[38]: [('n04540053', 'volleyball', 0.9431327),  
          ('n09421951', 'sandbar', 0.021305429),  
          ('n04371430', 'swimming_trunks', 0.0068396116)]
```

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.

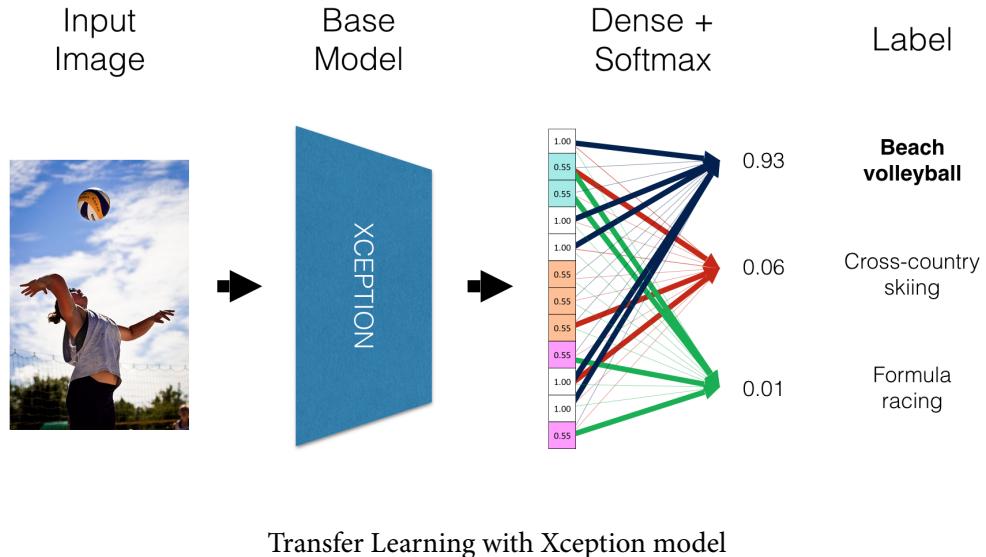
First of all we're going to set a value for the `img_size = 299`. This is the correct input size for Xception and it corresponds to the size of images it was originally trained on.

```
In [39]: 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 will 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.



Transfer Learning with Xception model

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`:

```
In [40]: 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:

```
In [41]: 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:

```
In [42]: 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:

```
In [43]: model.summary()
```

Layer (type)	Output Shape	Param #
xception (Model)	(None, 2048)	20861480
dense_1 (Dense)	(None, 256)	524544
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 3)	771

```
Total params: 21,386,795
Trainable params: 21,332,267
Non-trainable params: 54,528
```

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:

```
In [44]: model.layers[0].trainable = False
```

Let's check the model summary again.

```
In [45]: model.summary()
```

```

Layer (type)          Output Shape         Param #
=====
xception (Model)     (None, 2048)        20861480
dense_1 (Dense)      (None, 256)         524544
dropout_1 (Dropout)  (None, 256)         0
dense_2 (Dense)      (None, 3)           771
=====
Total params: 21,386,795
Trainable params: 525,315
Non-trainable params: 20,861,480
=====
```

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:

```
In [46]: 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`:

```
In [47]: 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:

```
In [48]: 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 ±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

```
In [49]: train_datagen = ImageDataGenerator(  
    preprocessing_function=preprocess_input,  
    rotation_range=15,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=5,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

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:

```
In [50]: 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:

```
In [51]: test_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input)
```

We will flow test images from the `test_path` directory:

```
In [52]: 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:

```
In [53]: model.fit_generator(
    train_generator,
    steps_per_epoch=65,
    epochs=1)
```

```
Epoch 1/1
65/65 [=====] - 48s 733ms/step - loss: 0.5263 - acc: 0.7945
```

```
Out[53]: <keras.callbacks.History at 0x7f57bfb58d0>
```

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:

```
In [54]: model.evaluate_generator(test_generator)
```

```
Out[54]: [0.22299607986885797, 0.9046563192904656]
```

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:

```
In [55]: model.predict_classes(img_tensor)
```

```
Out[55]: 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`:

```
In [56]: train_generator.class_indices
```

```
Out[56]: {'Beach volleyball': 0, 'Cross-country skiing': 1, 'Formula racing': 2}
```

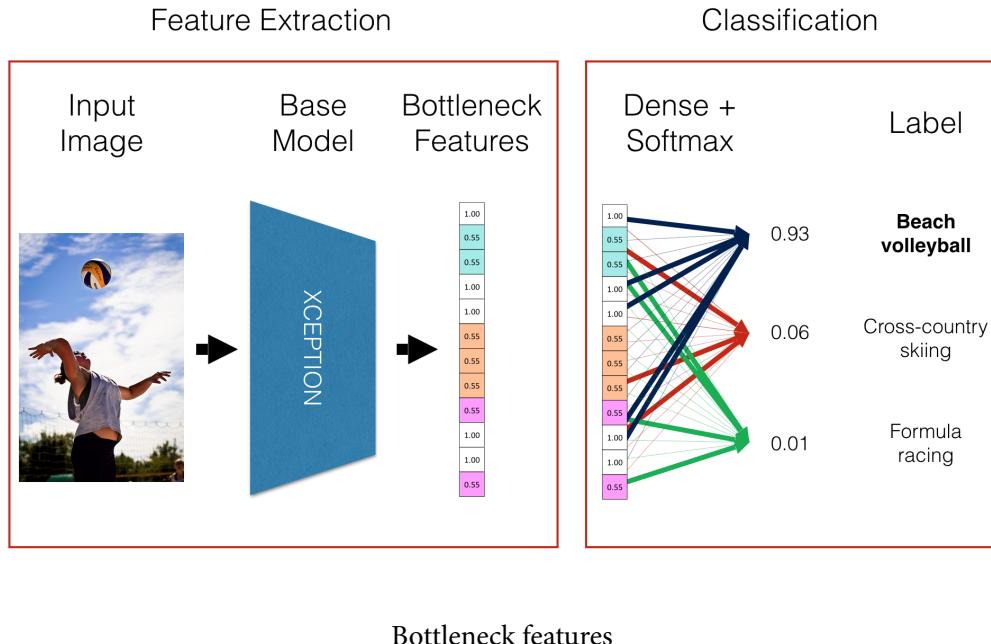
Awesome! We've performed transfer learning for the first time and we've reused a giant pre-trained model for our goal. This was really good, but it did take a bit of a long time to train. Can we speed that up? The answer is yes! Let's introduce bottleneck features.

## 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:

```
In [57]: 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:

```
In [58]: 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 [=====] - 28s 429ms/step
```

Let's also recover the training labels from the same generator.

```
In [59]: 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:

```
In [60]: bottlenecks_train.shape
```

```
Out[60]: (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:

```
In [61]: 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:

```
In [62]: label_map = list(train_generator.class_indices.keys())
label_map
```

```
Out[62]: ['Beach volleyball', 'Cross-country skiing', 'Formula racing']
```

Then let's use the `label_map` to convert the labels into the corresponding label names:

```
In [63]: label_names = [label_map[i] for i in labels.argmax(axis=1)]
```

```
label_names[:10]
```

```
Out[63]: ['Formula racing',
 'Beach volleyball',
 'Beach volleyball',
 'Cross-country skiing',
 'Formula racing',
 'Beach volleyball',
 'Beach volleyball',
 'Formula racing',
 'Formula racing',
 'Beach volleyball']
```

Great. Now let's generate bottleneck features for the images in the batch:

```
In [64]: bottlenecks = base_model.predict(images, verbose=1)
```

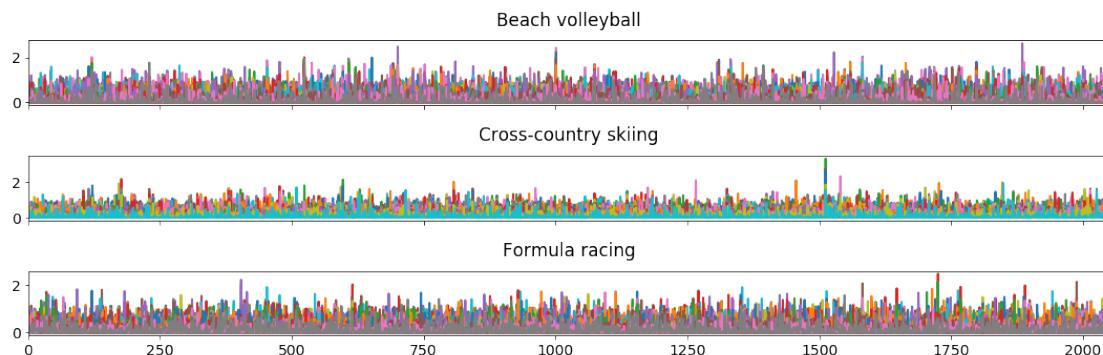
```
256/256 [=====] - 3s 13ms/step
```

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:

```
In [65]: 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()
```

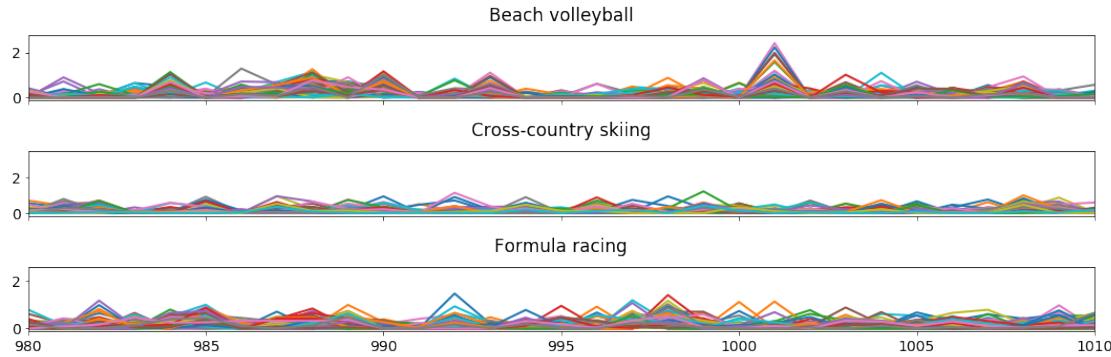


Hmm, although the 3 plots look somewhat different, it's hard to tell if there's anything interesting. Let's zoom in to the range 980-1010:

```
In [66]: 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.

```
In [67]: import gzip
```

```
In [68]: fname_ = '../data/sports/bottlenecks_train.npy.gz'
np.save(gzip.open(fname_, 'wb'), bottlenecks_train)
```

```
In [69]: fname_ = '../data/sports/labels_train.npy'
np.save(open(fname_, 'wb'), labels_train)
```

Let's also generate the bottlenecks for the test set:

```
In [70]: 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 [=====] - 12s 419ms/step
```

and the test labels:

```
In [71]: labels_test = test_generator.classes
```

and let's save them too:

```
In [72]: fname_ = '../data/sports/bottlenecks_test.npy.gz'  
np.save(gzip.open(fname_, 'wb'), bottlenecks_test)
```

```
In [73]: fname_ = '../data/sports/labels_test.npy'  
np.save(open(fname_, '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`.

```
In [74]: fname_ = '../data/sports/bottlenecks_train.npy.gz'  
X_train = np.load(gzip.open(fname_, 'rb'))  
  
fname_ = '../data/sports/bottlenecks_test.npy.gz'  
X_test = np.load(gzip.open(fname_, 'rb'))
```

We can check the shape of the train data, and verify that it's a matrix:

```
In [75]: X_train.shape
```

```
Out[75]: (2100, 2048)
```

Similarly, let's load the labels:

```
In [76]: fname_ = '../data/sports/labels_train.npy'  
y_train = np.load(open(fname_, 'rb'))  
  
fname_ = '../data/sports/labels_test.npy'  
y_test = np.load(open(fname_, 'rb'))
```

and check the shape as well:

```
In [77]: y_train.shape
```

```
Out[77]: (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:

```
In [78]: from keras.utils import to_categorical
```

```
In [79]: 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.

```
In [80]: 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:

```
In [81]: 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:

```
In [82]: fc_model.compile(optimizer='rmsprop',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
```

Here's a summary of the model:

```
In [83]: fc_model.summary()
```

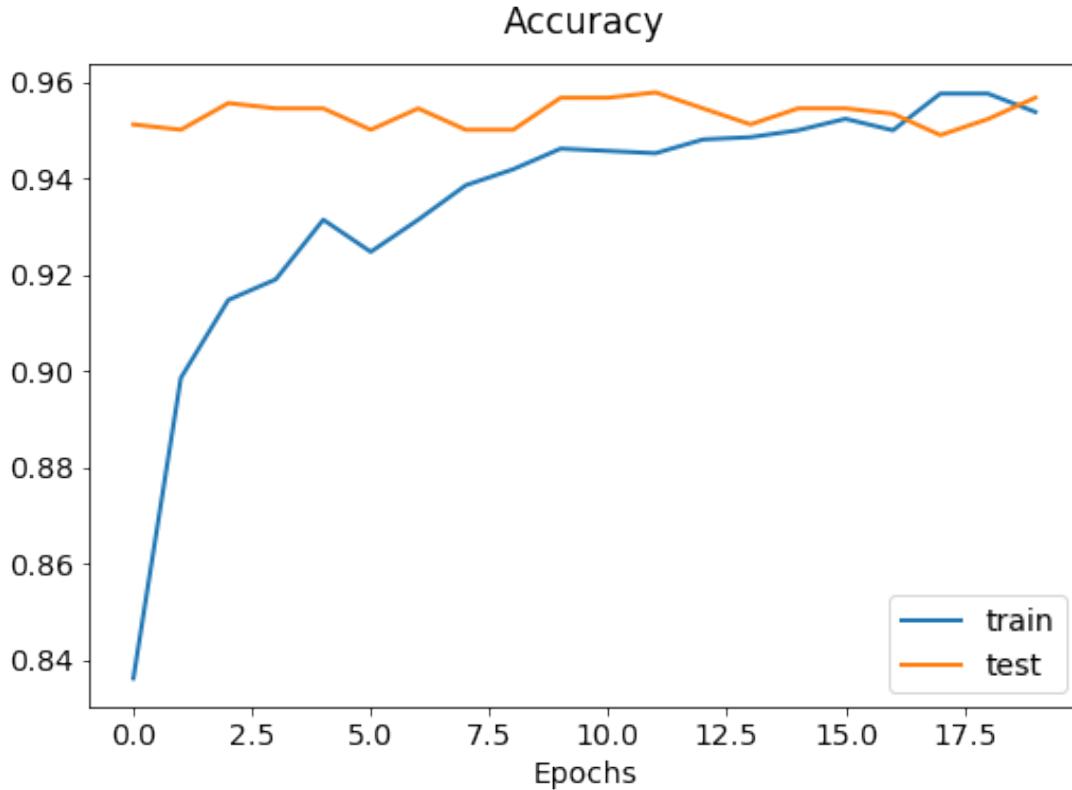
```
-----  
Layer (type)          Output Shape         Param #  
-----  
dropout_2 (Dropout)    (None, 2048)          0  
-----  
dense_3 (Dense)        (None, 16)           32784  
-----  
dropout_3 (Dropout)    (None, 16)           0  
-----  
dense_4 (Dense)        (None, 3)            51  
-----  
Total params: 32,835  
Trainable params: 32,835  
Non-trainable params: 0  
-----
```

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:

```
In [84]: history = fc_model.fit(X_train, y_train_cat,  
                           epochs=20,  
                           verbose=0,  
                           batch_size=batch_size,  
                           validation_data=(X_test, y_test_cat))
```

And let's plot the accuracy:

```
In [85]: plt.plot(history.history['acc'])  
plt.plot(history.history['val_acc'])  
plt.title('Accuracy')  
plt.legend(['train', 'test'])  
plt.xlabel('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:

```
In [86]: from sklearn.neighbors import DistanceMetric
```

and let's load an instance of the Euclidean metric, which is the usual vector difference distance:

```
In [87]: 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:

$$d(\mathbf{x}', \mathbf{x}) = \sqrt{(\mathbf{x}' - \mathbf{x})^2} = \sqrt{\sum_i (x'_i - x_i)^2} \quad (11.1)$$

and given the two vectors:

```
In [88]: a = np.array([1, 2])
```

```
In [89]: b = np.array([2, -1])
```

it is calculated as:

```
In [90]: np.sqrt(np.square(a - b).sum())
```

```
Out[90]: 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:

```
In [91]: images, labels = bottleneck_generator(
    train_path, batch_size=2100).next()
```

```
Found 2100 images belonging to 3 classes.
```

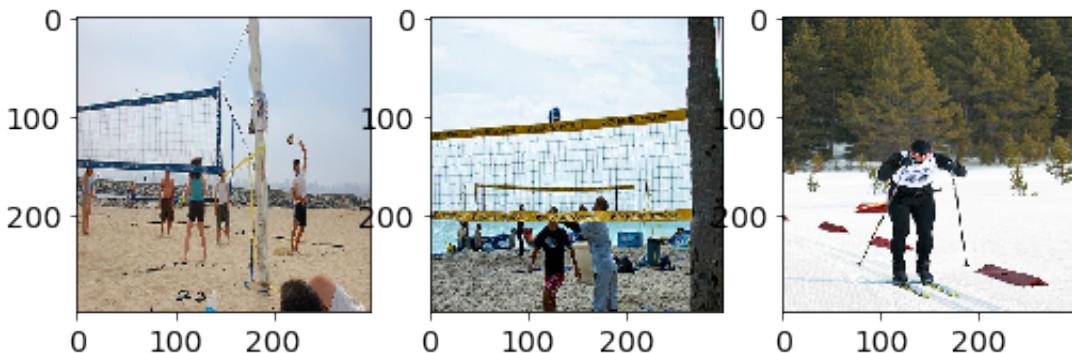
And let's display a few of them. Note that since our images have been normalized during pre-processing, their pixels are now values between -1 and 1. `plt.imshow` requires floating point images to have values between 0 and 1 so we will need to add 1 and divide by 2 each pre-processed image in order to display it correctly. Let's define a helper function that does that.

```
In [92]: def imshow_scaled(img):
    plt.imshow((img + 1) / 2)
```

```
In [93]: 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:

```
In [94]: np.sqrt(np.square(X_train[0] - X_train[1]).sum())
```

```
Out[94]: 6.608993
```

while the distance between the first and the third is:

```
In [95]: np.sqrt(np.square(X_train[0] - X_train[900]).sum())
```

```
Out[95]: 10.53551
```

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:

```
In [96]: bn_dist = dist.pairwise(X_train)
```

Let's check the shape of the matrix we have obtained:

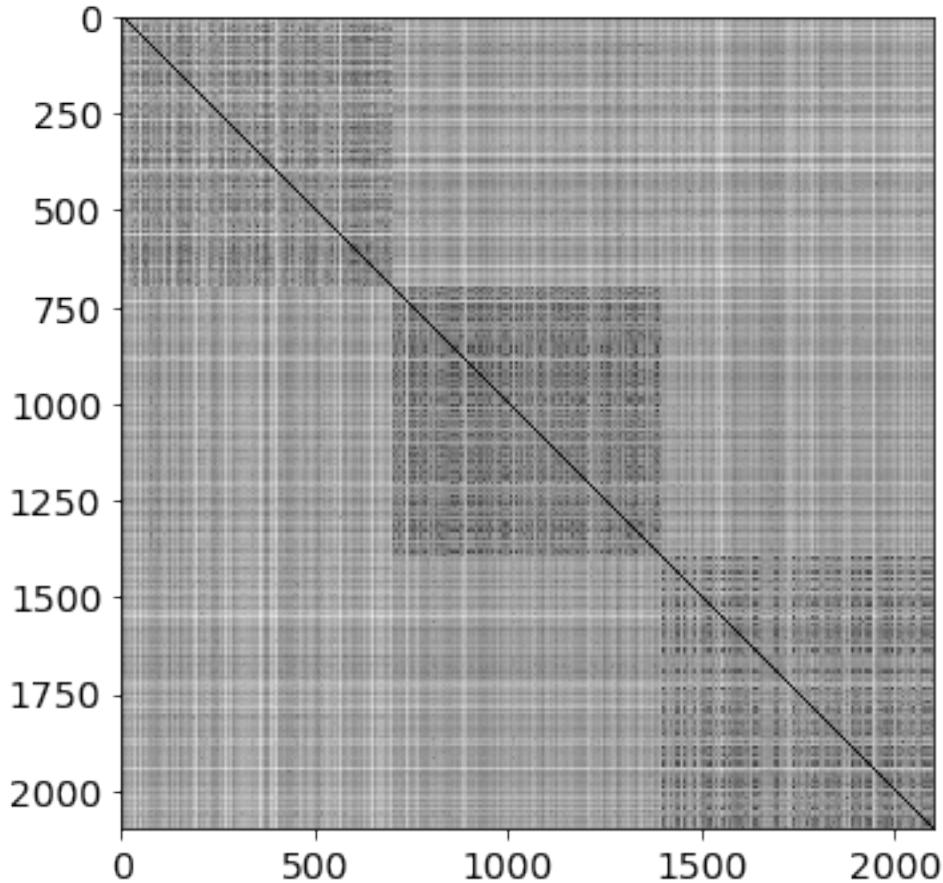
```
In [97]: bn_dist.shape
```

```
Out[97]: (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:

```
In [98]: plt.imshow(bn_dist, cmap='gray')
```

```
Out[98]: <matplotlib.image.AxesImage at 0x7f57b70a2ba8>
```

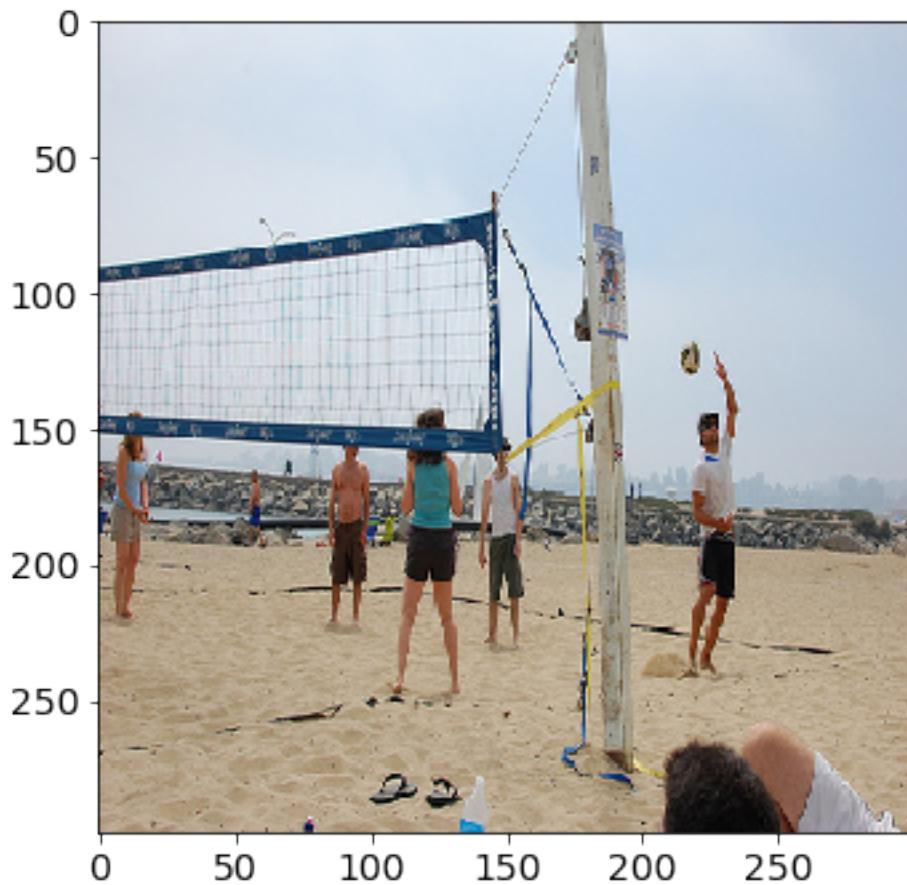


Notice a couple of things about this matrix: 1. The darker a pixel the closer two corresponding images are. - The matrix is simmetric with respect to the diagonal, which makes sense since the distance between image 1 and image 2 is the same as the distance between image 2 and image 1 - The diagonal is the darkest of all, which also makes sense, since an image will 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:

```
In [99]: 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:

```
In [100]: dist_from_sel = pd.Series(bn_dist[0])
```

Let's sort these and display the first few images:

```
In [101]: dist_from_sel.sort_values().head(9)
```

Out[101]:

---

	o
0	0.000000
431	5.348794
15	5.404700
32	5.787818
636	5.809691
648	5.847935
429	5.848355
48	5.952395
656	5.967599

---

Let's display these. We will display 9 images, in a grid of 3x3. Let's make this configurable by defining a few parameters:

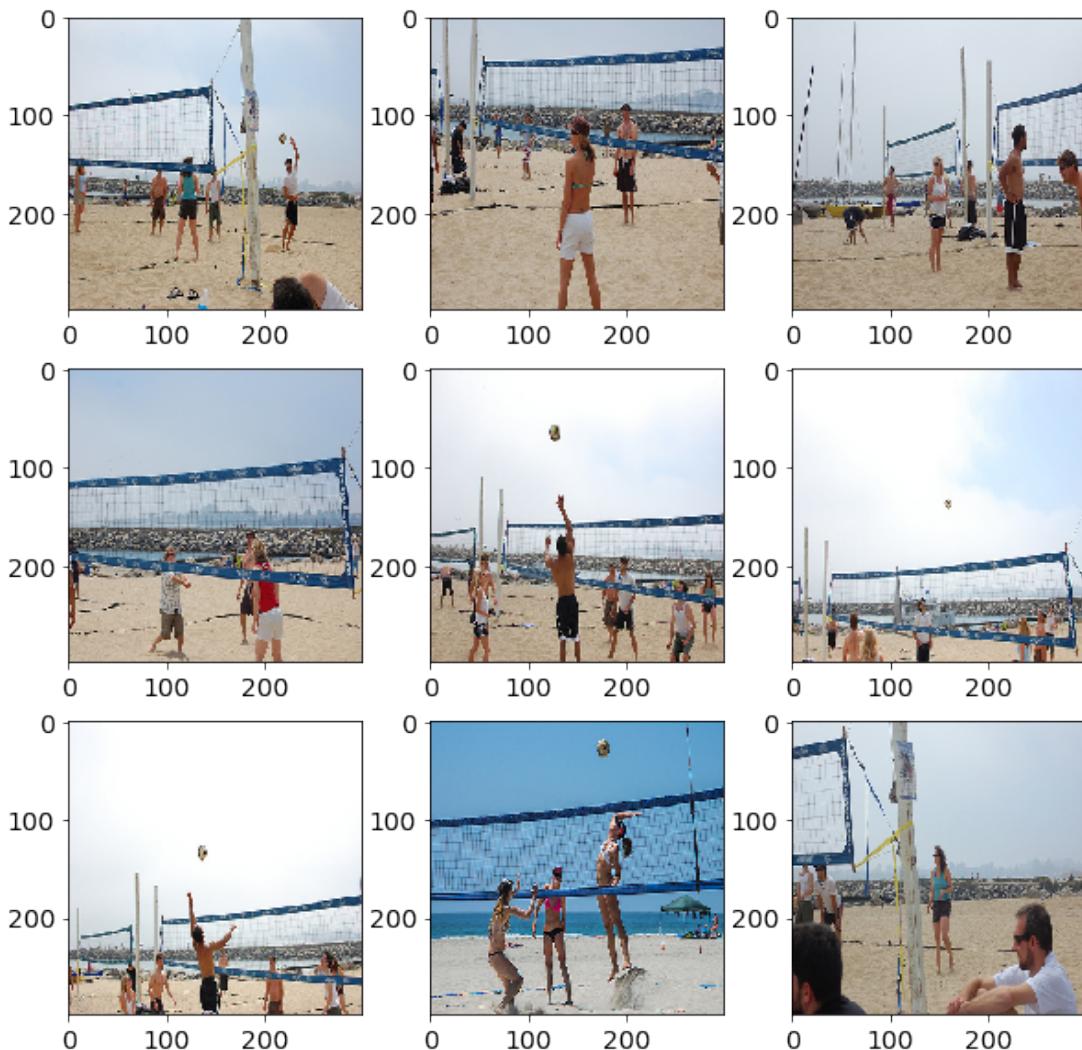
```
In [102]: n_rows = 3
          n_cols = 3
          n_images = n_rows * n_cols
```

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 `dist_from_sel` and then using the `.head` command that retrieves the first elements in the series:

```
In [103]: retrieved = dist_from_sel.sort_values().head(n_images)
```

Now let's loop over the index of the retrieved images and plot the images:

```
In [104]: 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
          plt.tight_layout()
```



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:

```
In [105]: def image_search(img_index, n_rows=3, n_columns=3):
    n_images = n_rows * n_columns

    # create Pandas Series with distances from image
    dist_from_sel = pd.Series(bn_dist[img_index])

    # sort Series and get top n_images
    retrieved = dist_from_sel.sort_values().head(n_images)

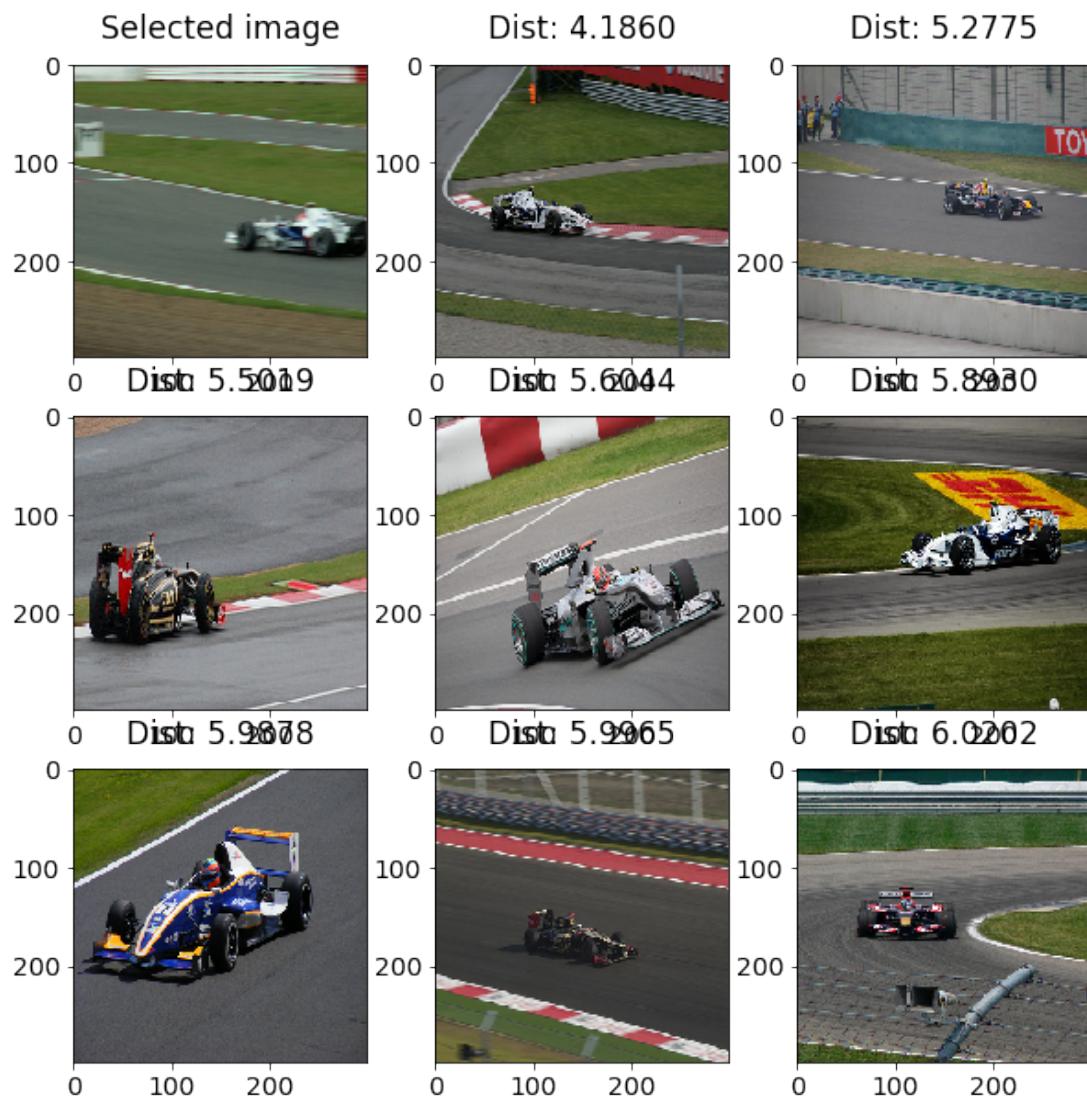
    # create figure, loop over closest images indices
    # and display them
```

```
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
plt.tight_layout()
```

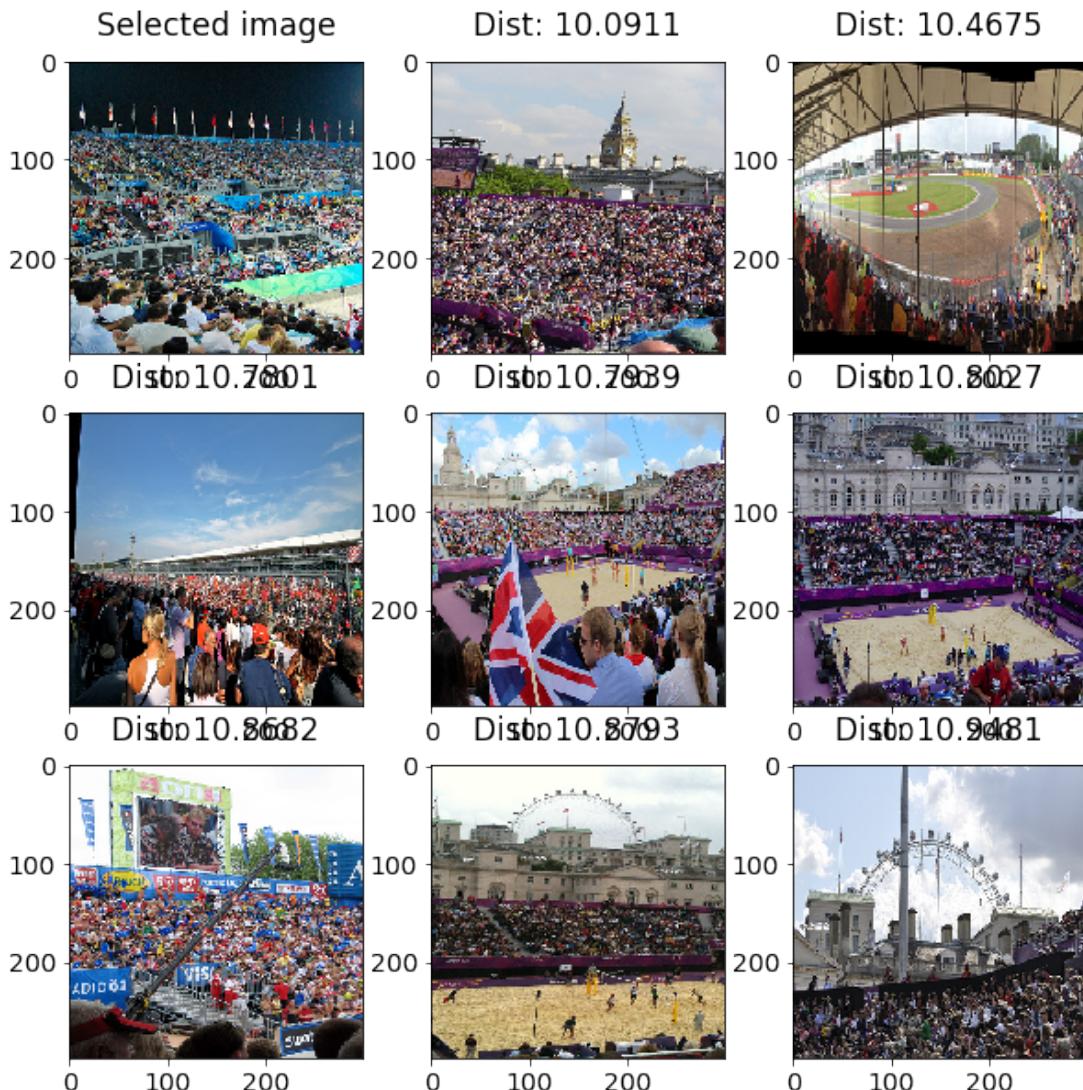
In [106]: image\_search(900)



```
In [107]: image_search(1600)
```

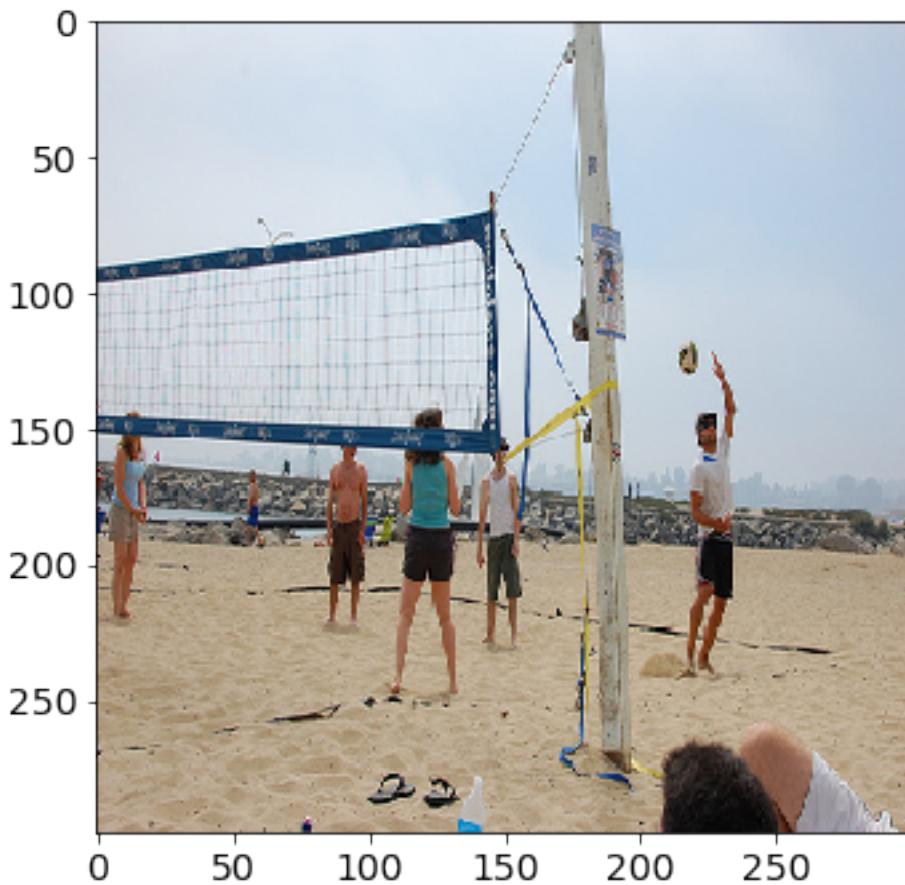


```
In [108]: image_search(100)
```



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:

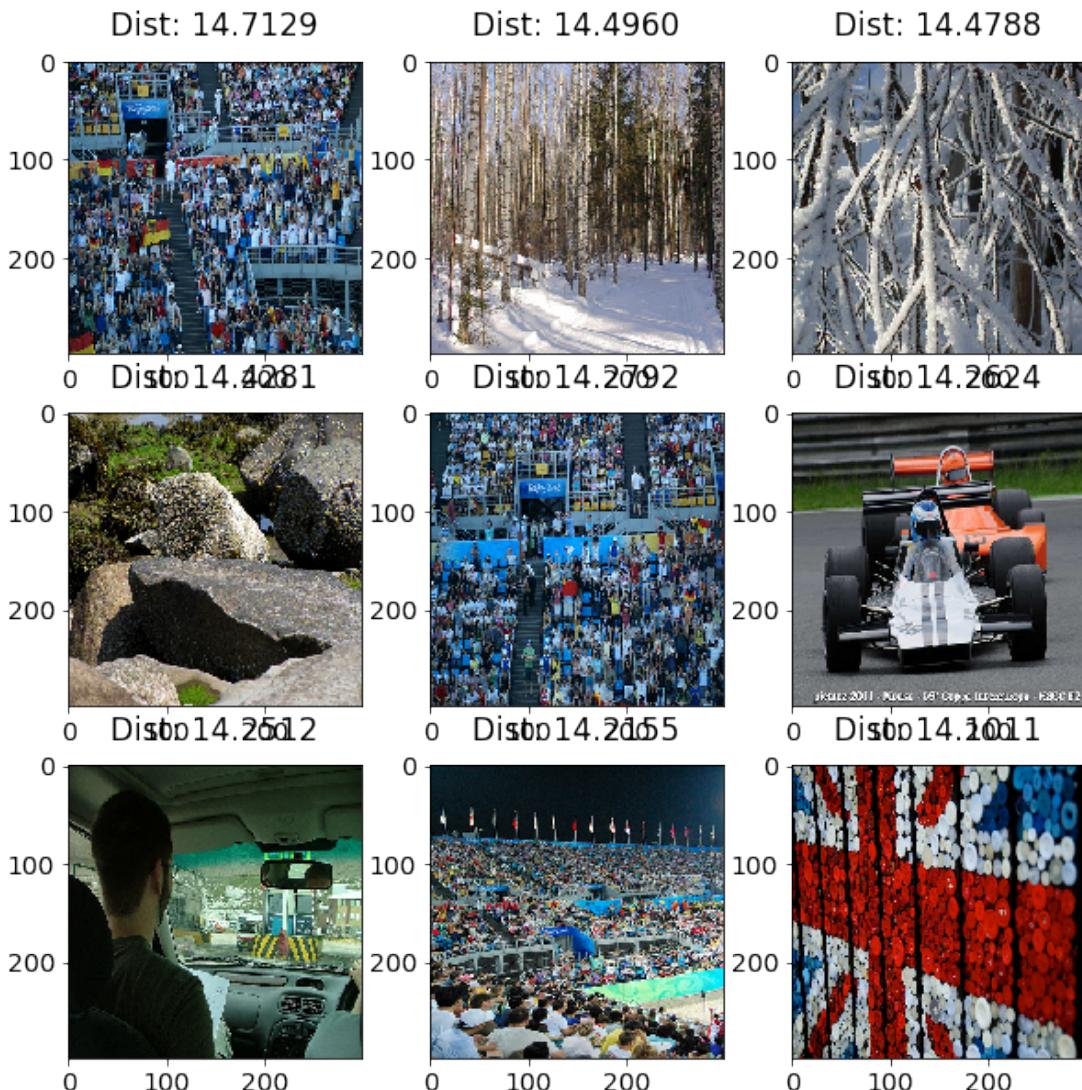
```
In [109]: imshow_scaled(images[0])
```



The most distant images are:

```
In [110]: retrieved = pd.Series(bn_dist[0]).sort_values(ascending=False).head(9)
```

```
In [111]: 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
plt.tight_layout()
```



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.



# 12

## Pretrained Embeddings for Text

In the last part of [Chapter 8](#) we introduced the concept of **Embeddings**. These are dense vectors that represent words and are often used as a starting point when approaching NLP problems like language translation or sentiment analysis. The word vectors we introduced in Chapter 8 were trained together with the rest of the model and therefore were specific to the particular problem we were trying to solve.

For example, when we trained our [Recurrent Model](#) for the movie review sentiment analysis task, the embedding layer was the first layer in a Sequential model, followed by a recurrent layer and a classification head for the sentiment prediction. The weights of the word vectors in the embedding layer were learned together with the weights of the recurrent layer and the classification layer. The single task of predicting the sentiment of a movie review would provide a value for the loss which would then propagate back through the network to adjust both the recurrent and the embedding weights.

The approach we have just described has 2 major drawbacks:

- 1) since the embedding layer has many of weights (e.g., for a vocabulary of 10k words, each embedded with 100 numbers, we have 1M weights), we need a lot of data for this model to generalize well and avoid overfitting
- 2) since the embeddings are trained on the sentiment analysis task, they will work well on that task but will not necessarily learn general properties of the semantic space. In other words, we will not be able to use those same embeddings for a completely different NLP task, like translation.

To overcome these two limitations, researchers have proposed a different approach to building more general embeddings. These approaches try to capture the meaning of a word in a language and build generic embeddings where words with similar meanings are represented by similar vectors. Although this may seem crazy at first, it works well in practice.

In this chapter, we will see a couple of different famous embeddings, and we will use them to do fun operations with text.

Let's get started.

## “Unsupervised”-“supervised learning”

How can we train a generic embedding layer that encapsulates the meaning of words? We'll have to resort to a trick that is often used when training very large Neural Networks. This is often referred to as “Unsupervised Learning” although, as we shall see, it is actually a special case of Supervised Learning where the labels are not generated by humans.

Let's think back of the [sequence generation](#) example we introduced in Chapter 8. There we built a network that learned to predict the most likely letter after a sequence of 3 letters, using a corpus of English baby names. The same approach can be used to build a model that is trained to predict the most likely word after a sequence of words. For example, if trained using a corpus of songs from John Lennon, the model should be able to learn that “heaven” should be the most likely word after the words “imagine there's no”. This task is called **language modeling** because what the model learn is the structure of a language.

The output of this model is a Softmax over the vocabulary of the language, while the input is a sequence of words that will be encoded as vectors by the input embedding layer of the language model. Since the model is essentially solving a forecasting task (predict the most likely words after a sequence of words), we are still in the domain of Supervised Learning. The labels, however, are contained in the corpus of text itself.

Consider for example this excerpt from the song [Imagine](#) by John Lennon:

```
In [1]: text = """
    imagine there's no heaven
    it's easy if you try
    no hell below us
    above us only sky
    imagine all the people living for today

    imagine there's no country
    it isn't hard to do
    nothing to kill or die for
    and no religion too
    imagine all the people living life in peace,
    you

    you may say i'm a dreamer
    but i'm not the only one
    i hope some day you'll join us
    and the world will be as one

    ...
"""
print(text)
```

From this text we can build the following pairs of inputs and labels:

Sequence	Label
imagine there's no	heaven
there's no heaven	it's
no heaven it's	easy
heaven it's easy	if
it's easy if	you
easy if you	try
...	...

Both the inputs and the labels are obtained from the same corpus by simply sliding a window of fixed length and asking the model to predict the word coming immediately after the window.

This generic forecasting approach is amazingly powerful! We are no longer limited by our ability to label data. We can use any text, literally we could use the whole of Wikipedia, and train a very generic language model that attempts to predict the next word in a sequence. The embeddings of this model must be more generic than the ones trained on the sentiment problem!

Starting with this intuition, that you can obtain labels from the text itself, researchers have invented several approaches to train generic embeddings. We will mention here a few of the most famous and show you where to find them and how to use them.

Let's start with a very common set of embeddings called **GloVe**, which stands for Global Vectors for Word Representation.

## GloVe embeddings

```
In [2]: with open('common.py') as fin:
    exec(fin.read())
```

```
In [3]: with open('matplotlibconf.py') as fin:
    exec(fin.read())
```

In the `data/embeddings` folder we provide a download script that downloads and extracts GloVe embeddings from: <https://nlp.stanford.edu/projects/glove/>. Here is its content:

```
In [4]: cat ../data/embeddings/glove_download.sh
```

```
# Script to download and extract Glove
# word embeddings. More information at:
# https://nlp.stanford.edu/projects/glove/
```

```
# Uncomment the file you'd like to download
EMBEDDINGS=glove.6B
# EMBEDDINGS=glove.42B.300d
# EMBEDDINGS=glove.840B.300d

# download and extract
wget http://nlp.stanford.edu/data/$EMBEDDINGS.zip
unzip $EMBEDDINGS.zip
rm $EMBEDDINGS.zip
```

Go ahead and run the script to retrieve the `glove.6B` embeddings. Let's take a look at them. First let's define a path variable:

```
In [5]: glove_path = '../data/embeddings/glove.6B.50d.txt'
```

And let's look at the first line in the file:

```
In [6]: with open(glove_path, 'r', encoding='utf-8') as fin:
    line = fin.readline()
```

```
In [7]: line
```

```
Out[7]: 'the 0.418 0.24968 -0.41242 0.1217 0.34527 -0.044457 -0.49688 -0.17862 -0.00066023 -0.65
```

As you can see the line contains the word `the` as first element, followed by 50 space-separated floating point numbers, which form the word vector. Let's define a parse function that parses the line and returns the word and the vector as a numpy array.

We should take care of removing the trailing `\n` character at the end, then split the line at spaces, which will return a list. Finally we'll take the first element in the list as `word` and the remaining values as the vector. Here's the parse function:

```
In [8]: def parse_line(line):
    values = line.strip().split()
    word = values[0]
    vector = np.asarray(values[1:], dtype='float32')
    return word, vector
```

Now that we have defined a parse function, let's use it to load the word embeddings. We will use a Python dictionary for the embeddings and one for the word index. Let's create two empty dictionaries:

```
In [9]: embeddings = {}
word_index = {}
```

Let's also create an empty list for the inverted index that will map numbers to words:

```
In [10]: word_inverted_index = []
```

Now we can loop over the lines in the file, parse each line and store it in the embeddings and word index dictionary. We will enumerate the lines as we proceed with the loop so that we can also retrieve their numeric index.

Let's do it:

```
In [11]: with open(glove_path, 'r', encoding='utf-8') as fin:
    for idx, line in enumerate(fin):
        word, vector = parse_line(line) # parse a line

        embeddings[word] = vector # add word vector
        word_index[word] = idx # add idx
        word_inverted_index.append(word) # append word
```

Let's check a few entries in the indexes we built. For example, using `word_index`, we can retrieve the line number at which the word `good` appears:

```
In [12]: word_index['good']
```

```
Out[12]: 219
```

Using the `word_inverted_index` we can do the reverse i.e. given a line number, find the corresponding word:

```
In [13]: word_inverted_index[219]
```

```
Out[13]: 'good'
```

The `embeddings` dictionary contains the actual word vectors, so for example, the word vector corresponding to the word `good` is the following:

```
In [14]: embeddings['good']
```

```
Out[14]: array([-3.5586e-01,  5.2130e-01, -6.1070e-01, -3.0131e-01,  9.4862e-01,
   -3.1539e-01, -5.9831e-01,  1.2188e-01, -3.1943e-02,  5.5695e-01,
   -1.0621e-01,  6.3399e-01, -4.7340e-01, -7.5895e-02,  3.8247e-01,
   8.1569e-02,  8.2214e-01,  2.2220e-01, -8.3764e-03, -7.6620e-01,
   -5.6253e-01,  6.1759e-01,  2.0292e-01, -4.8598e-02,  8.7815e-01,
   -1.6549e+00, -7.7418e-01,  1.5435e-01,  9.4823e-01, -3.9520e-01,
   3.7302e+00,  8.2855e-01, -1.4104e-01,  1.6395e-02,  2.1115e-01,
   -3.6085e-02, -1.5587e-01,  8.6583e-01,  2.6309e-01, -7.1015e-01,
   -3.6770e-02,  1.8282e-03, -1.7704e-01,  2.7032e-01,  1.1026e-01,
   1.4133e-01, -5.7322e-02,  2.7207e-01,  3.1305e-01,  9.2771e-01],
  dtype=float32)
```

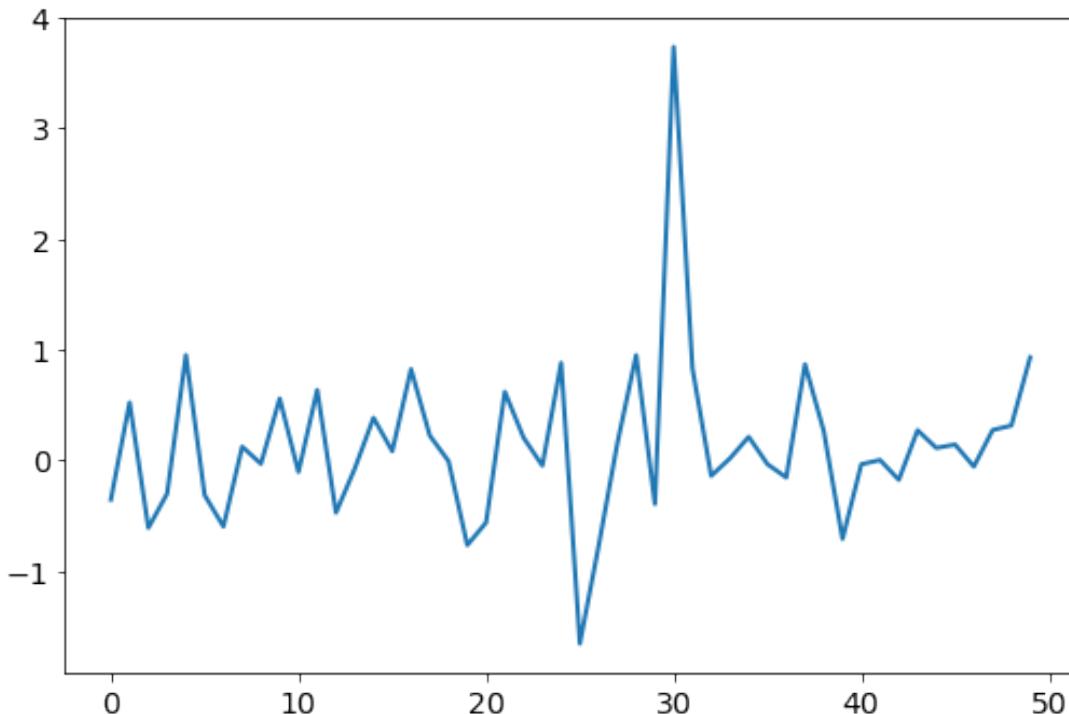
How many components does this vector have? Let's check its length:

```
In [15]: embedding_size = len(embeddings['good'])
embedding_size
```

```
Out[15]: 50
```

We can also plot it:

```
In [16]: plt.plot(embeddings['good']);
```



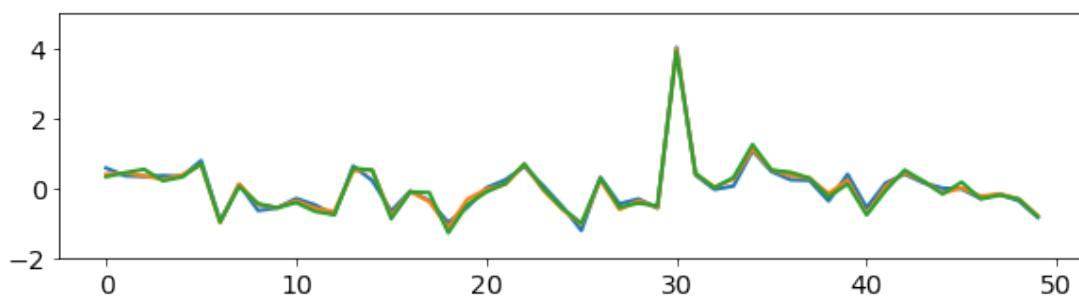
It doesn't tell us much, but for example we can compare the word vectors of a few words and see how they look like. Let's plot a few numbers, like *two*, *three* and *four* and a few animals like *cat*, *dog* and *rabbit*. As you will see numbers will look very similar to one another, and animals will be distinctly different from the numbers:

```
In [17]: plt.subplot(211)
plt.plot(embeddings['two'])
plt.plot(embeddings['three'])
plt.plot(embeddings['four'])
plt.title("A few numbers")
plt.ylim(-2, 5)

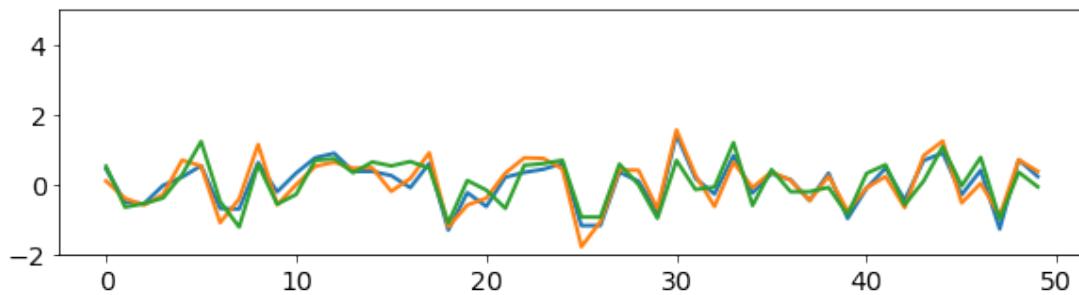
plt.subplot(212)
plt.plot(embeddings['cat'])
plt.plot(embeddings['dog'])
plt.plot(embeddings['rabbit'])
plt.title("A few animals")
plt.ylim(-2, 5)

plt.tight_layout()
```

A few numbers



A few animals



This is reminiscent of Bottleneck features we've encountered in Chapter 11. Each word has been encoded with a vector with 50 numbers and words that carry similar semantic value will be encoded with similar vectors. As we shall see, GloVe vectors are built by looking at co-occurrence of words, so the above plots tell us that numbers like two, three and four are often found nearby similar words, which makes sense. I can say: "I ran *two* miles" or "I ran *three* miles" and both sentences make sense, while I cannot say "I ran *cat* miles". *Two* and *three* can be found in similar contexts and therefore are encoded as similar vectors.

Let's see how many words are contained in our GloVe embeddings by checking the `len` of the `embeddings` variable:

```
In [18]: vocabulary_size = len(embeddings)
vocabulary_size
```

```
Out[18]: 400000
```

There are 400000 words in the embeddings, which will cover most of our needs.

TIP: If you are curious to know more about how GloVe embeddings are build, we encourage you to read the [original paper](#) or take a look at the [original source code](#).

## Loading pre-trained embeddings in Keras

As we have seen in [Chapter 8](#), Keras has an `Embedding` layer that can be trained to build custom embeddings. Here we will learn how to initialize it using pre-trained embeddings like the GloVe embeddings we have just loaded. First let's load the `Sequential` model and the `Embedding` layer from Keras:

```
In [19]: from keras.models import Sequential
from keras.layers import Embedding
```

Using TensorFlow backend.

Next we are going to arrange our pre-trained embeddings as a giant matrix of shape (`vocabulary_size`, `embedding_size`). We will do this in 2 steps. First let's create a zero matrix with the correct shape:

```
In [20]: embedding_weights = np.zeros((vocabulary_size,
                                     embedding_size))
```

Then let's iterate over the items in our `word_index` dictionary and let's assign each vector in the `embeddings` dictionary to a line in the matrix. For example, we know from above that the word `good` has index 219. This means we will assign its vector to the row in the matrix corresponding to index 219 (i.e. the 220-th row). Let's do it:

```
In [21]: for word, index in word_index.items():
    embedding_weights[index, :] = embeddings[word]
```

Now that we have our pre-trained weights arranged in a matrix, we can create an `Embedding` layer in Keras, passing the weights as initialization. We will specify the `input_dim` to be equal to the `vocabulary_size` and the `output_dim` to be equal to the `embedding_size`. Then we'll use the parameter `weights` to pass a list of weights, in this case just the embedding weights. Finally we'll set both the `mask_zero` and the `trainable` flags to `False`.

```
In [22]: emb_layer = Embedding(input_dim=vocabulary_size,
                             output_dim=embedding_size,
                             weights=[embedding_weights],
                             mask_zero=False,
                             trainable=False)
```

Let's stop for a second and make sure we understand the last 2 flags. Here's the documentation of `mask_zero`:

```
mask_zero: Whether or not the input value 0 is a special "padding"
           value that should be masked out.
           This is useful when using [recurrent layers] (recurrent.md)
           which may take variable length input.
           If this is `True` then all subsequent layers
           in the model need to support masking or an exception will be raised.
           If mask_zero is set to True, as a consequence, index 0 cannot be
           used in the vocabulary (input_dim should equal size of
           vocabulary + 1).
```

In our case, we have used the index 0 in the vocabulary for the word `the`, as we can check in the `word_inverted_index`:

```
In [23]: word_inverted_index[0]
```

```
Out[23]: 'the'
```

so we have to set `mask_zero` to `False`. Had we started enumerating the word vectors from 1, we could have reserved the 0 value for padding, which, as the doc says, is useful when using recurrent layers.

The `trainable=False` flag simply tells Keras that this layer is not trainable, i.e. its weights cannot be changed during training. We used this earlier when using pre-trained models for images in [Chapter11](#).

Notice that simply passing the matrix to the `Embedding` constructor is not enough. We need to put this layer in a model in order for Keras to actually create a Tensorflow graph with it. Let's do it:

```
In [24]: model = Sequential()
          model.add(emb_layer)
```

Now that we have created a model we can check that the embedding layer does use the pre-trained weights. Let's check the embeddings for the word `cat`. Here are the original values we loaded from the file:

```
In [25]: embeddings['cat']
```

```
Out[25]: array([ 0.45281 , -0.50108 , -0.53714 , -0.015697,  0.22191 ,  0.54602 ,
       -0.67301 , -0.6891 ,  0.63493 , -0.19726 ,  0.33685 ,  0.7735 ,
       0.90094 ,  0.38488 ,  0.38367 ,  0.2657 , -0.08057 ,  0.61089 ,
      -1.2894 , -0.22313 , -0.61578 ,  0.21697 ,  0.35614 ,  0.44499 ,
       0.60885 , -1.1633 , -1.1579 ,  0.36118 ,  0.10466 , -0.78325 ,
       1.4352 ,  0.18629 , -0.26112 ,  0.83275 , -0.23123 ,  0.32481 ,
       0.14485 , -0.44552 ,  0.33497 , -0.95946 , -0.097479,  0.48138 ,
      -0.43352 ,  0.69455 ,  0.91043 , -0.28173 ,  0.41637 , -1.2609 ,
       0.71278 ,  0.23782 ], dtype=float32)
```

Now let's retrieve the index of the word `cat` and let's pass it to the `model.predict` method. First we retrieve the index of the word `cat` using the `word_index`:

```
In [26]: cat_index = word_index['cat']
```

The index is:

```
In [27]: cat_index
```

```
Out[27]: 5450
```

Now that we have the index, we run `model.predict` on a double nested list containing the single index of the word `cat`:

TIP: we need to use a list here because the `predict` method expects as input an integer matrix of size (batch, input\_length), as explained in the [documentation](#) and here we have a batch of 1 point with a sequence of 1 word.

```
In [28]: model.predict([[cat_index]])
```

```
Out[28]: array([[[ 0.45281 , -0.50108 , -0.53714 , -0.015697,  0.22191 ,
   0.54602 , -0.67301 , -0.6891 ,  0.63493 , -0.19726 ,
   0.33685 ,  0.7735 ,  0.90094 ,  0.38488 ,  0.38367 ,
   0.2657 , -0.08057 ,  0.61089 , -1.2894 , -0.22313 ,
  -0.61578 ,  0.21697 ,  0.35614 ,  0.44499 ,  0.60885 ,
  -1.1633 , -1.1579 ,  0.36118 ,  0.10466 , -0.78325 ,
   1.4352 ,  0.18629 , -0.26112 ,  0.83275 , -0.23123 ,
   0.32481 ,  0.14485 , -0.44552 ,  0.33497 , -0.95946 ,
  -0.097479,  0.48138 , -0.43352 ,  0.69455 ,  0.91043 ,
  -0.28173 ,  0.41637 , -1.2609 ,  0.71278 ,  0.23782 ]]],  
      dtype=float32)
```

As you can see the method returns exactly the same values, so we have successfully initialized a Keras model with a pre-trained embedding.

## Gensim

Gensim is a topic modelling library in Python that contains a lot of functions related to extracting meaning and manipulating text. Let's import it and have some fun with word embeddings:

```
In [29]: import gensim
```

In order to load Glove embeddings using Gensim we need to convert them into the appropriate format. Luckily for us Gensim has a function for that. We just need to import the `glove2word2vec` script:

```
In [30]: from gensim.scripts.glove2word2vec import glove2word2vec
```

and then run it. We first set input and output paths:

```
In [31]: glove_path = '../data/embeddings/glove.6B.50d.txt'  
glove_w2v_path = '../data/embeddings/glove.6B.50d.txt.vec'
```

```
In [32]: glove2word2vec(glove_path, glove_w2v_path)
```

```
Out[32]: (400000, 50)
```

Next we use the `gensim.models.KeyedVectors.load_word2vec_format` to load the word vectors from the file:

```
In [33]: from gensim.models import KeyedVectors
```

```
In [34]: glove_model = KeyedVectors.load_word2vec_format(
    glove_w2v_path, binary=False)
```

Now that we have loaded the vectors into a Gensim model, we have access to a lot of functionality. For example, we can quickly find what are the most similar words to a given word.

Here's how we look for the 5 closest words to the word `good`:

```
In [35]: glove_model.most_similar(positive=['good'], topn=5)
```

```
Out[35]: [('better', 0.9284390807151794),
('really', 0.9220625162124634),
('always', 0.9165270328521729),
('sure', 0.9033513069152832),
('something', 0.9014205932617188)]
```

The `.most_similar` method allows for both a list of positive and negative words. Feel free to play with the list of words to get a feel for how they affect the output. The closest words to `good` are words that can appear in the same context as `good`, so it's quite obvious that we should get similar adjectives like `better` or adverbs like `really` and `always`.

If we try with the number `two` we should get other numbers:

```
In [36]: glove_model.most_similar(positive=['two'], topn=5)
```

```
Out[36]: [('three', 0.9885902404785156),
('four', 0.9817472696304321),
('five', 0.9644663333892822),
('six', 0.964131236076355),
('seven', 0.9512959718704224)]
```

## Word Analogies

Since word vectors are vectors, we can do any vector operation with them, including addition, subtraction and dot products. For example we can perform operations between words like:

```
result = king - man + woman
```

where the vector `result` is a perfectly valid vector in the embedding space. Using the `.most_similar` method, we can look for the 3 vectors closest to `result`. Can you guess which vector will be the closest?

If you guessed `queen`, which is the feminine counterpart of `king`, you guessed right. Let's see it in action:

```
In [37]: glove_model.most_similar(positive=['king', 'woman'],
                                 negative=['man'], topn=3)
```

```
Out[37]: [('queen', 0.8523603677749634),
           ('throne', 0.7664333581924438),
           ('prince', 0.7592144012451172)]
```

i.e. we have found that

```
queen ~ king - man + woman
```

Another way to look at this is to say that the vector `queen - king` is similar to the vector `woman - man`. This is often represented with this famous picture:

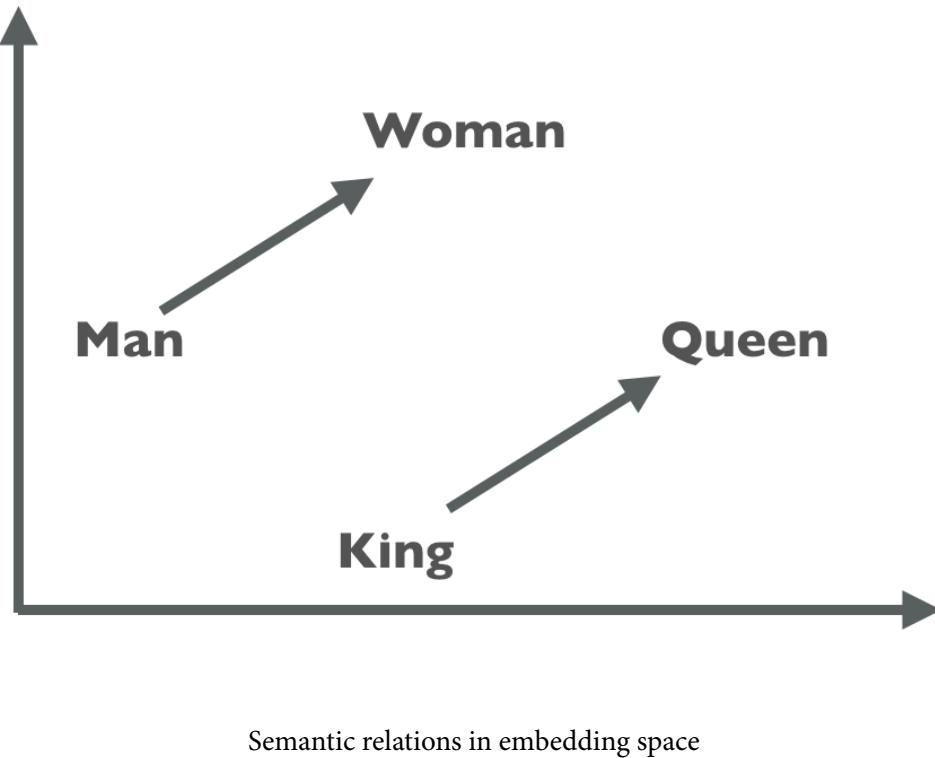
In this figure we imagine an embedding space that has only 2 axes (instead of 50 or 300) and represent the words as points in the embedded space. The arrows represent the vector distances between the words.

Since this chart can be useful to understand how the model is representing the semantic space, it's legitimate to ask if we can visualize all of glove words in a similar chart using a dimensionality reduction technique. The answer to this question is yes, and we can actually leverage tensorboard for this. We'll see how in the next Section.

## Visualization

Tensorboard also contains a projector that allows us to explore word embeddings visually. Let's save our word embeddings in a tensorflow model and let's visualize them in Tensorboard. First we need to create an output folder. We'll use the `/tmp/ztdl_models/embeddings/` folder for output. Let's first create it. We will need the `os` module:

```
In [38]: import os
```



Then let's define a path variable that we'll use later too:

```
In [39]: model_dir = '/tmp/ztdl_models/embeddings/'
```

Let's also load the `rmtree` function from `shutil` so that we can delete the directory if it already exists:

```
In [40]: from shutil import rmtree
```

```
In [41]: rmtree(model_dir, ignore_errors=True)
```

Finally let's create the folder:

```
In [42]: os.makedirs(model_dir)
```

For the purposes of this visualization we will limit our Embedding layer to the top most frequent 4000 words in the glove set. Let's set a variable called `n_viz` to 4000:

```
In [43]: n_viz = 4000
```

Now we create a new embedding layer, with only 4000 x 50 weights. Notice that we still pass the `mask_zero=False` parameter since our first vector, corresponding to the index 0, is the word `the`:

```
In [44]: emb_layer_viz = Embedding(n_viz,
                                  embedding_size,
                                  weights=[embedding_weights[:n_viz]],
                                  mask_zero=False,
                                  trainable=False)
```

Let's stick this layer into a Sequential model so that the Tensorflow graph gets populated:

```
In [45]: model = Sequential([emb_layer_viz])
```

Now we retrieve the tensorflow variable pointing to the weights:

```
In [46]: word_embeddings = emb_layer_viz.weights[0]
```

Let's see what this variable is:

```
In [47]: word_embeddings
```

```
Out[47]: <tf.Variable 'embedding_2/embeddings:0' shape=(4000, 50) dtype=float32_ref>
```

As you can see it is a `tf.Variable`. As explained in the documentation, maintains state in the graph across calls to `run()`. Variables are used in Tensorflow precisely to contain the values of the weights of a network.

We now would like to use tensorboard to visualize this data. The [most recent documentation](#) on how to do this is a bit cryptic, so we refer to an [earlier version of the documentation](#) which is more helpful.

We need to accomplish three things, which are independent: - save the embedding tensor - save a file with the words (metadata) - save a configuration file that binds the metadata to the embedding tensor

Let's start by saving the model. The code that follows is specific to Tensorflow. First we load the Keras backend and tensorflow itself:

```
In [48]: import keras.backend as K
        import tensorflow as tf
```

Next we retrieve the tensorflow session from Keras:

```
In [49]: sess = K.get_session()
```

And we create an instance of `tf.train.Saver` to save the model:

```
In [50]: saver = tf.train.Saver([word_embeddings])
```

Now we can use the `saver` to save the model:

```
In [51]: saver.save(sess, os.path.join(model_dir, 'model.ckpt'), 1)
```

```
Out[51]: '/tmp/ztdl_models/embeddings/model.ckpt-1'
```

This operation creates a few files in the `model_dir` folder, as you can see with the `os.listdir` command:

```
In [52]: os.listdir(model_dir)
```

```
Out[52]: ['checkpoint',
          'model.ckpt-1.data-00000-of-00001',
          'model.ckpt-1.index',
          'model.ckpt-1.meta']
```

These files contain the weights, but have no information about which word corresponds to each vector. What we need is a `metadata.tsv` file with the list of words. We can easily create it by looping over the indexes from 0 to `n_viz` adding one word per line to the file:

```
In [53]: fname = os.path.join(model_dir, 'metadata.tsv')

with open(fname, 'w', encoding="utf-8") as fout:
    for index in range(0, n_viz):
        word = word_inverted_index[index]
        fout.write(word + '\n')
```

You can check the content of this file and see that it contains one word per line like:

```
the
,
.
of
to
and
...
```

This file can be loaded in tensorboard using

```
In [54]: config = """embeddings {{
    tensor_name: "{tensor}"
    metadata_path: "{metadata}"
}}""".format(tensor=word_embeddings.name,
            metadata='metadata.tsv')
```

```
In [55]: print(config)
```

```
embeddings {
    tensor_name: "embedding_2/embeddings:0"
    metadata_path: "metadata.tsv"
}
```

```
In [56]: fname = os.path.join(model_dir, 'projector_config.pbtxt')
```

```
with open(fname, 'w', encoding="utf-8") as fout:
    fout.write(config)
```

**TIP:** Note that you could also use the following code to generate and save the same config file in a programmatic way. Since the config file is so simple, we opted for the explicit solution above.

```
from tensorflow.contrib.tensorboard.plugins import projector

config = projector.ProjectorConfig()
emb_config = config.embeddings.add()
emb_config.tensor_name = word_embeddings.name
emb_config.metadata_path = 'metadata.tsv'

summary_writer = tf.summary.FileWriter(model_dir, sess.graph)
projector.visualize_embeddings(summary_writer, config)
```

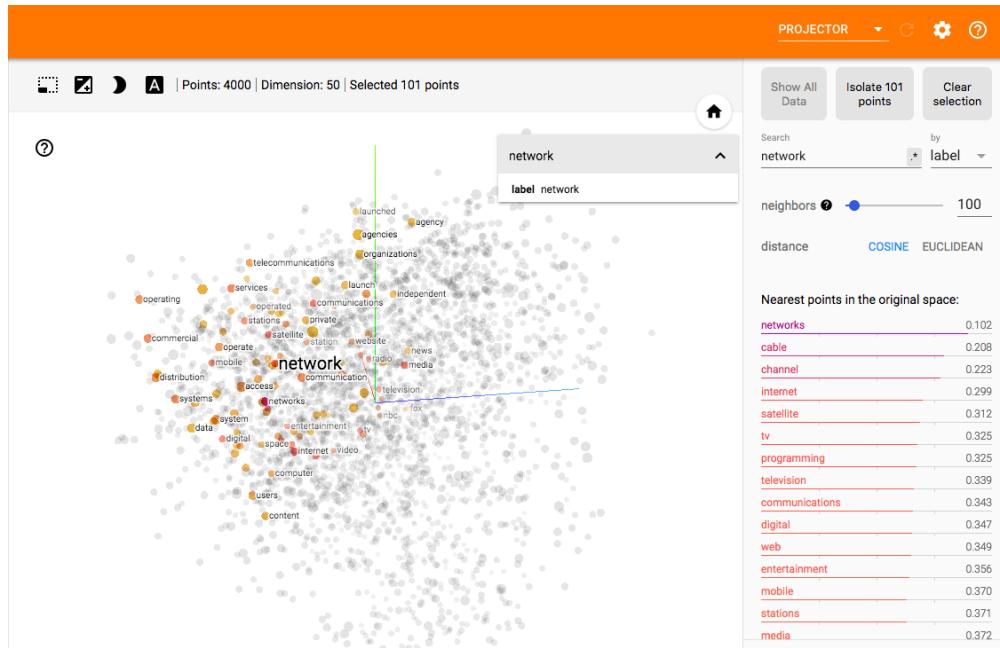
We can now start [Tensorboard](#):

```
tensorboard --logdir=/tmp/ztdl_models/embeddings/
```

and point our browser to:

<http://localhost:6006/#projector>

We should see the word embedding projector spinning. Using the Search tab on the right, let's look for a specific word, for example the word **network** and see what are the closest words. You should see something like this:



Tensorflow projector

## Other pre-trained embeddings

We have learned how to use and visualize pre-trained embeddings using GloVe. GloVe is a commonly used set of pre-trained embeddings but it's not the only one. We will briefly introduce here two other very popular sets of pre-trained embeddings: *Word2Vec* and *FastText*.

### Word2Vec

**Word2Vec** is a set of word vectors introduced by Google in 2013. These vectors also try to encapsulate the meaning of a word by looking at its context, i.e. the words that precede it and follow it.

You can find a detailed tutorial on how Word2Vec vectors are built in the [Tensorflow Tutorials page](#). The two main useful ideas are the *Skip-gram Model* and the *noise-contrastive estimation (NCE) loss*. Let's take a look at these in a bit more detail.

## Skip-grams

Skip-grams are simply pairs of words that appear in context. Let's consider the first 2 lines of the song imagine:

```
imagine there's no heaven it's easy if you try
```

and let's focus on the word *heaven*. If we choose a context of -2, +2 words, we see that the following words appear in the context of heaven: - *there's* - *no* - *it's* - *easy*

We could therefore try to build a model that takes a word in input and tries to predict the probability that another word in the dictionary appears in its context by using input/output pairs like:

INPUT	OUTPUT
heaven	there's
heaven	no
heaven	it's
heaven	easy

We could use a Softmax over the whole dictionary and eventually learn these probabilities. However this would require a huge amount of data, since the dictionary size is huge.

Hence, Word2Vec is trained using a trick called *Negative Sampling*.

## Negative Sampling and the noise-contrastive estimation (NCE) loss

Imagine solving a slightly different problem where instead of having a word as input and a word as output, we have a pair as input and a binary label as output. We can use the pairs above as positive examples, since they are actual pairs found in the training text, and we can build fake pairs that have the same first word and a random second word. Our data will look like this:

INPUT	OUTPUT
(heaven, there's)	1
(heaven, no)	1
(heaven, it's)	1
(heaven, easy)	1
(heaven, cat)	0
(heaven, brain)	0
(heaven, swimming)	0
(heaven, chair)	0
...	...

We have constructed negative pairs by randomly choosing words from the dictionary. This model learns to predict the probability that given a first word, the second word is in its context or not, which is the same problem as before. However, this model is much faster to train, because we are only choosing a small set of negative examples at each batch instead of having the whole dictionary.

These two tricks allow to train Word2Vec quite easily. In our case we will not even train it but rely on a pre-trained version of these embeddings that's been trained using data from Google News.

TIP: if you want to learn more about Word2Vec we encourage you to read the [Wikipedia page](#) and to go through the [tutorial](#) mentioned earlier.

## FastText

[FastText](#) is a library for efficient learning of word representations and sentence classification developed by Facebook Research. It is open-source, free and lightweight and it allows users to learn text representations and text classifiers. [Here](#) is the Github repository and [here](#) you can read the blog post announcing its publication.

FastText has two very interesting aspects.

- 1) fastText word vectors are built from vectors of substrings of characters contained in it. This allows to build vectors even for misspelled words or concatenation of words.
- 2) fastText has been designed to work on a variety of languages by taking advantage of the languages morphological structure. This means pre-trained vectors are available for many other languages besides english.

You can download the pre-trained English vectors [here](#). You can download the pre-trained vectors for many other languages [here](#)

In the exercises we will use these vectors and compare their results with GloVe and Word2Vec.

## Exercises

### Exercise 1

Compare the representations of Word2Vec, Glove and FastText. In the `data/embeddings` folder we provided you with two additional scripts to download FastText and Word2Vec. Go ahead and download each of them into the `data/embeddings`. Then load each of the 3 embeddings in a separate Gensim model and complete the following steps:

1. define a list of words containing the following words: 'good', 'bad', 'fast', 'tensor', 'teacher', 'student'.

- create a function called `get_top_5(words, model)` that retrieves the top 5 most similar words to the list of words and compare what the 3 different embeddings give you
- apply the same function to each word in the list separately and compare the lists of the 3 embeddings.
- explore the following word analogies:

man:king=woman:? ==> expected queen  
france:paris=germany:? ==> expected berlin  
teacher:teach=student:? ==> expected learn  
cat:kitten=dog:? ==> expected puppy  
english:friday=italiano:? ==> expected venerdì

Can word analogies be used for translation?

Note that loading the vector may take several minutes depending on your computer.

## Exercise 2

The [Reuters Newswire topic classification dataset](#) is a dataset of 11,228 newswires from Reuters, labeled over 46 topics. This dataset is provided in the `keras.datasets` module and it's easy to use.

Let's compare the performance of a model using pre-trained embeddings with a model using random embeddings on the topic classification task.

- Load the data from `keras.datasets.reuters`
- Retrieve the word index and create the `reverse_word_idx` as it was done for IMDB in [Chapter 8](#).
- Augment the reverse word index with `pad_char`, `start_char` and `oov_char` at indices 0, 1, 2 respectively.
- Check the maximum length of a newswire and use the `pad_sequences` function to pad everything to that 100 words.
- Create and train two models, one using pre-trained embeddings and the other using a randomly initialized embedding
- Compare their performance on this dataset using a recurrent model. In particular check which of the two models shows the worst overfitting.



# 13

## Serving Deep Learning Models

In this chapter we will learn how to serve a trained model. This topic is very important one! The goal of training a Machine Learning model is to use it to generate predictions. Deployment of Machine Learning models is a vast topic, we could write a whole book just about this topic. In this chapter we will present two ways of deploying a model and outline a set of things we need to consider for deployment. In the end, how will decide to deploy a model depends on the requirements that the application will have to satisfy and this will change case by case.

In this chapter we want to achieve a few goals. We want to explain at a high level how to think about the deployment process, highlighting the issues involved, and outlining the possible choices. This chapter will help us understand the decision you'll need to make when deploying a model as well as equip you with a list of resources that you can tap into, according to your needs.

Then we will show you two ways of deploying a model: a simple [Flask application](#) using a Python server and a more general deployment using [Tensorflow Serving](#). These are not the only two ways to deploy a model and we'll make sure to point you to additional resources, companies and products that simplify the management of the model deployment cycle.

So, let's start from the model development/deployment cycle.

### The model development cycle

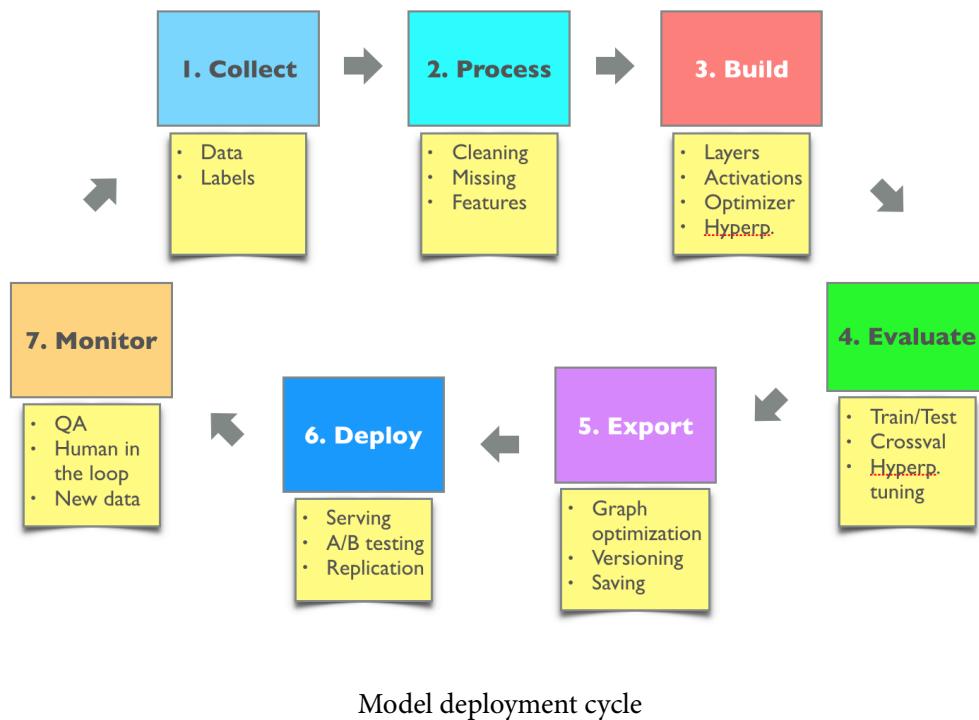
The concepts explained in this part are general to Machine Learning, not just Deep Learning, and they are independent of the framework used. At a high level, the model development cycle includes these 7 steps:

1. Data Collection
2. Data Processing

3. Model Development
4. Model Evaluation
5. Model Exporting
6. Model Deployment
7. Model Monitoring

These steps are part of a continuous deployment cycle: we never finish to improve our models and learning from new data. After deploying our first model, we will deploy a second version, a third, and so on. The performance of each new version will be compared to the performance of the previous one. Traffic will gradually be shifted towards the new model, like it happens with any other release in a continuous integration setup.

The 7 steps are not mutually exclusive, they happen in parallel. In other words, while you are working on developing (3) and evaluating (4) the current version of a model, the previous version of the model is already being monitored (7) and additional data and labels are being collected (1) and processed (2).



Let's now look at each step in greater detail.

## Data Collection

Throughout this book we used datasets based on files. These were either tabular files (Csv, Excel) or folders containing images or documents. In the real world there is usually a process involving data collection, where data is stored in a database or in a distributed file system for later use.

Examples of this process are:

- a database with the actions of your users in your website or app
- a data-lake with millions of documents that we would like to classify
- a cloud object store with images that we would like to recognize

Depending on the type of data, on its frequency and on its size, you will design different collection and storage systems.

Let's consider a few examples.

As a first case let's consider a bank that would like to train a model to decide which people can receive a loan. The information used as input for the model are things like:

- user information
- account activity
- past loans
- history of credit etc.

This information will typically be stored in several tables in a database. We will be able to create a dataset to train our model by simply joining data from a few tables of the database. Furthermore, we can probably work with a sample of the whole data as a starting point, especially if our first model is not going to be personalized to each user. This "snapshot" of the world, a dataset we extracted at some point in time, is going to be valid for at least a few days, if not for a few weeks or even months. That is to say, the general lending behavior of a population will surely evolve over time, but it will do so quite slowly, not from one day to the next. Another way to say this is to say that the statistical distribution of our users is stationary, or quasi-stationary, i.e. independent from time.

These facts allow us to train a model on a file, maybe a large file, but a fixed snapshot, like we did throughout this book and use that model in production. Once we have trained and evaluated our model, we will deploy it to our branches and the managers will have an "AI helper" to decide when to issue a loan or not.

Let's consider a very different case now. Let's say we want to build a system that based on the actions of a user in our web application will decide which advertisement to show. In this case the input will be a series of events in the app. Both the app and the ads inventory will change much more frequently in time, due to new feature releases, new clients etc. In this case we will re-train our model much more frequently, possibly every day, using the most recent data.

Besides frequency of re-training, other things to consider are:

- the kind of data, is it: files, documents, images, text, numbers in a table?
- the amount of data, how many new data points do we collect per day? 100? 1000? 1 million? 1 billion?  
The data collection and storage process will change dramatically based on that.

Modern Machine Learning products usually work as a continuous pipeline, where a model is continuously learning from new data and every now and then a snapshot of the most current model is saved and used in production for inference.

## Labels

As we know very well by now, in order to train a model with supervision, we need labels. Here too, there can be many different scenarios.

In some cases, we may not have those labels at all. For example, let's say we are training an algorithm to recognize offensive pictures in our user-generated content. We will need to collect a sample of pictures and have human supervisors manually label the offensive pictures with labels such as "violence", "nudity", etc. This labeling process will be slow and costly, but it will be necessary before we can proceed with any training.

Additionally, if we randomly sample our images, there will likely be very few offensive images, which would make labeling very slow because our human supervisors would receive mostly normal pictures. This is why most websites implement a button for users to report offensive content. This will effectively triage the pictures bringing the offensive ones to surface so that human supervisors can review them and generate labels accordingly.

On the opposite end, if we are training a model to advertise products, i.e. to predict the likelihood that a user will click on a certain product, the labels, i.e. the past clicks, are automatically recorded by our system.

In general, the win-win strategy for label generation is when the product manager can design a product in such a way that labels are automatically generated by the users or the process. Examples of this win-win approach are:

- tagging your friends on Facebook => labels for face recognition algorithm
- recording purchase actions on Amazon => labels for recommendation of other products
- "flag this post" button on Craigslist => labels for fraud/spam detection algorithm
- captchas that ask you to recognize street signs => labels for image recognition algorithm

What is the process for label generation in your case?

## Data Processing

This process is usually referred to as ETL in enterprise setting. It involves going from the raw data in your data store to features ready for consumption by the Machine Learning model.

At this stage you will focus on operations like data cleaning, data imputation, feature extraction and feature engineering. Once again, this will depend on your specific situation but it is important to keep in mind that when NULL values are present, i.e. when some data is missing, we need to stop and ask ourselves why such data is missing. Discussion of the different cases of missing data is beyond the scope of this book, but we invite you to read this [Wikipedia article on missing data](#) to be cognizant of the issues involved when dealing with missing data.

Other data processing steps may involve generating features, augmenting the data, one-hot encoding, using pre-trained models for feature extraction etc.

## Model Development

The model development is deeply interconnected with the data processing step. Here is where we focus the attention on deciding which model we will apply.

- Will you attempt with a classical Machine Learning model first?
- Will you go straight to Neural Networks and Deep Learning?
- What will your model architecture look like?
- Which hyper-parameters will you start with?
- Are you going to train the model with gradient descent?
- If so, which optimizer will you use?

Again, the choices will depend on the particular situation you are in. It is important that you keep in mind a general approach to this phase: keep your feedback loop as rapid as possible. It's not a mystery that a rapid feedback loop is a great strategy in software development (e.g. Agile development). In Machine Learning this is just as true, so if you are considering two options to improve your model and one takes 1 hour to test and the other 1 week to test, you should absolutely choose the former over the latter.

Let's look at one example. Let's say we have some indication that our model will improve if we had more data but we are also not sure that we have chosen the right architecture for the model. In some cases, getting more data could be as simple as running a new SQL query to extract a few more million data points from our database, in some other cases it could be much more complicated, involving manual label generation with a team of supervisors, which would likely require days if not weeks of delay.

On the other hand, if re-training the model takes minutes or even just a few hours we could spin up a new copy of the model with a different architecture and train it quickly. If training the model takes 1 week that's clearly not an easy option.

We will have to take all these factors into account when developing the model and choosing where to start first.

## Model Evaluation

Once we have decided what model architecture we are going to use, we need to train the model on the data. The majority of this book has been dedicated to this process, so you should be pretty familiar with terms like train/test splitting, cross-validation, and hyper-parameter tuning. It is important that at this stage you know what baseline you are measuring against and what metric you are going to use. If this is your first model attempt, you are probably comparing the performance with a dummy model (i.e. one that always predicts the average label or the majority class). On the other hand, if you have previously deployed other models, you will compare the model performance with that of the previous model.

You will have to consider what overall goal you are trying to achieve. In the case of a binary classification problem, you will consider metrics like precision and recall to evaluate if your model has lots of false positives or false negatives. The choice you make will depend on your business goal and your data. For example, if you are deploying a model that predicts patient sickness you will try to avoid false negatives, because you wouldn't want to leave any screened patient with a false impression that they are healthy when

they are are not. On the other hand, if you are developing a system for flagging spam, you will focus more on avoiding false positives, which would route legitimate emails to the spam folder, creating a bad user experience.

In summary you will need to decide:

- the metric you are trying to optimize
- the baseline for that metric
- what is considered a significant improvement over the baseline
- the kind of errors you would like to minimize

These considerations will guide you in during the training process.

If you plan to do hyper-parameter training, it is also very important that you split your total dataset into three parts:

- training
- validation
- test

The **training** data will be used to train the model. The **validation** data will serve as “test” data for hyper-parameter tuning. I.e. for each new combination of hyper-parameters you will train the model on the training portion and validate the model on the validation portion. This split is like having two nested training loops. The inner training loop will choose the weights and biases of your network, the outer training loop will choose the hyper-parameters like learning rate, batch size, number of layers etc.

The test set will never be seen by any of these models until the end. Once you have chosen the best hyper-parameters and you have trained the best model on that data, only then, you will test your trained model on the **test** set to get a sense of how well your model is going to perform with out-of-sample data, i.e. an indication of how well your model is going to do when deployed.

## Model Exporting

Once the model has been trained and evaluated, it is time to get it ready for serving. What happens at this stage will depend on many requirements including desired latency, footprint, the device you are planning to use for serving and many more.

At one end of the spectrum, this steps is as simple as saving the trained model to disk as is. The trained model is composed of 2 parts, the model architecture and the trained weights. This method is perfect when we use Keras to build our model and plan to serve it as part of a Python/Flask application. It is not optimized at all, but if all we care is to build a proof of concept and if we don’t need to support high traffic then this can be fast to execute. This is the first method we will explore in this chapter.

On the other end of the spectrum are large scale deployments. If we are planning to use our model in a high availability production environment, we need to make sure it is optimized for serving predictions within the constraints required by our application.

For example, if we plan to use a model to make a real-time decision on which ad to serve or which product to recommend to a user visiting our website, we will have very stringent latency requirements, usually few tens of milliseconds at most and this will influence the choices we make when designing the model as well as when saving it.

The topic of model optimization is vast and it requires tools that go beyond the scope of this book. We will therefore limit ourselves to pointing out what kind of optimizations are possible and where to look for information about them.

Model optimization techniques may involve:

- stripping away all operations that are not needed for inference. The Tensorflow graph underlying our Keras model contains all the operations required for training as well, including the gradients calculations and the optimizer. None of these is relevant at inference time and we should strip them away from the graph. Tensorflow has a [Graph Transform Tool](#) that includes a lot of options to check out. Common cases covered by the tool are:
  - Optimizing for Deployment
  - Fixing Missing Kernel Errors on Mobile
  - Shrinking File Size
  - Eight-bit Calculations
- low level compilation of tensorflow operation using the [Accelerated Linear Algebra compiler \(XLA\)](#). This compiler optimizes the operation for the specific platform that we are going to use to deploy and it can help in the following areas:
  - Improve execution speed
  - Improve memory usage
  - Reduce reliance on custom Ops
  - Reduce mobile footprint
  - Improve portability

In addition to model optimization, inference performance can be improved by choosing the hardware platform that is most adapt to our model. Currently Tensorflow supports CPU, GPU and TPU training and inference. In the coming years we'll see a flourishing of hardware platforms dedicated to Deep Learning model training and serving, which will bring additional options to the table.

In this chapter we'll see how to save a Keras model in Tensorflow format, so that all the above tools can be applied.

## Model Deployment

Model deployment refers to how we are going to make our model available to the rest of the world. In several cases, this is a Python/Flask application that simply loads the model to memory and then runs `model.predict` when requested. This is the first method we will explore in this chapter. It's a great way to deploy a proof of concept in situations where we do not need a high throughput.

The natural extension of this method is to containerize the Flask app with Docker so that we can replicate the model multiple times and adjust our model to the load requested by our application. While this works, it is not the recommended solution when scaling out operations. Tensorflow offers its own server which is the preferred way to deploy models at scale.

That's why in this chapter we will also go through a minimal deployment with Tensorflow Serving. Tensorflow Serving is a powerful package developed with large deployments in mind. We will introduce it and guide you to more resources if you need to scale out operations with your models.

More generally, a new model will be deployed in parallel to an existing model and its performance will be validated with live traffic in a classic A/B test scenario where traffic is only partially routed to the new model and its performance is monitored against for some time before completely adopting it and phasing out the old one.

This strategy is why deployment must also include monitoring of the model performance.

## Model Monitoring

Last but not least, when we deploy a model we want to monitor its performance. It is important to sample the predictions of the model and send a few of them to human supervision in order to verify their quality. In other words, label collection never ends. We need to keep measuring the performance of our model against a known set of labels.

In some cases this process is automated, for example the case where our model predicts future values of a time series e.g. the price of a stock for trading purposes. In this case, as soon as we get the next value of the time series we can immediately compare it with the prediction from our model and monitor its quality in real-time.

In other cases, where labels are generated by human supervisors, we need to keep sending data to a QA team that will generate new labels for them. We can then compare the predictions with the labels and decide how to improve the model on the cases where it failed.

This process never ends, we can always come up with better models. However, we should not be discouraged by this. As British mathematician [George E. P. Box](#) said: “[All models are wrong; some models are useful](#)”. You can reap enormous benefits from a model that is not perfect.

Let's deploy our first model. We will build an API that can predict the location of a user based on the strength of WiFi signals detected.

## Deploy a model to predict indoor location

As an example of deployment we will develop an app that can determine the indoor location of a user based on the WiFi signal strength observed on smartphone. Data comes from [the UCI Machine Learning repository](#) and it is made available to you in [data/wifi\\_location.csv](#). We will quickly load and train a model and then focus on exporting this model to build an API that can predict the location of a user based on the WiFi signal strength.

## Data exploration

Let's start by loading the usual packages:

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:  
    exec(fin.read())
```

And let's load the data with Pandas:

```
In [3]: df = pd.read_csv('../data/wifi_location.csv')
```

Let's quickly inspect the data to get a sense for what we have:

```
In [4]: df.head()
```

Out[4] :

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	location
0	-64	-56	-61	-66	-71	-82	-81	0
1	-68	-57	-61	-65	-71	-85	-85	0
2	-63	-60	-60	-67	-76	-85	-84	0
3	-61	-60	-68	-62	-77	-90	-80	0
4	-63	-65	-60	-63	-77	-81	-87	0

It looks like we have 7 features, presumably the strengths of the wifi signals coming from 7 different access points. There's also a column called `location` that will be our label. Let's see how many locations there are in the dataset:

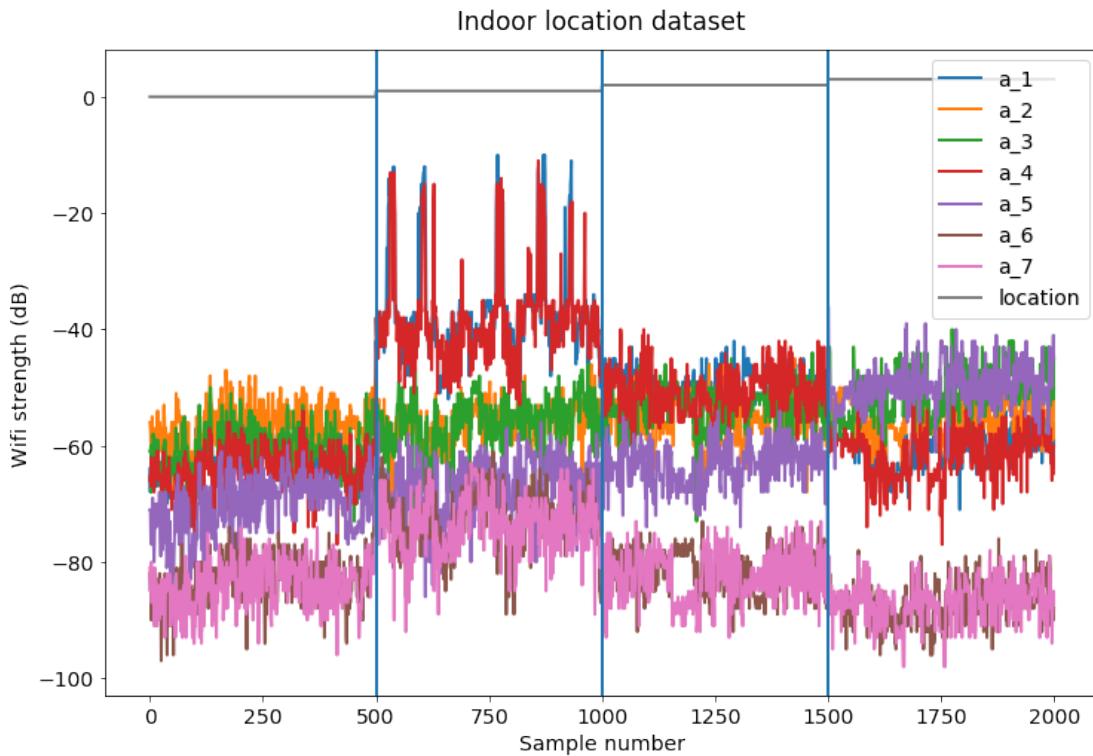
```
In [5]: df['location'].value_counts()
```

Out[5] :

	location
3	500
2	500
1	500
0	500

Great! The dataset is balanced and it has 500 examples of each location! Since we have only 2000 points total, we can plot the features and take a look at them:

```
In [6]: df.plot(figsize=(12, 8))
    plt.axvline(500)
    plt.axvline(1000)
    plt.axvline(1500)
    plt.title('Indoor location dataset')
    plt.xlabel('Sample number')
    plt.ylabel('Wifi strength (dB)');
```



From the plot we can clearly see that the wifi signal strengths are different in the 4 locations and therefore we can hope to be able to predict the location of a person based on these features. To further remark this point, let's do a pairplot using Seaborn and let's color the data by location:

```
In [7]: import seaborn as sns
```

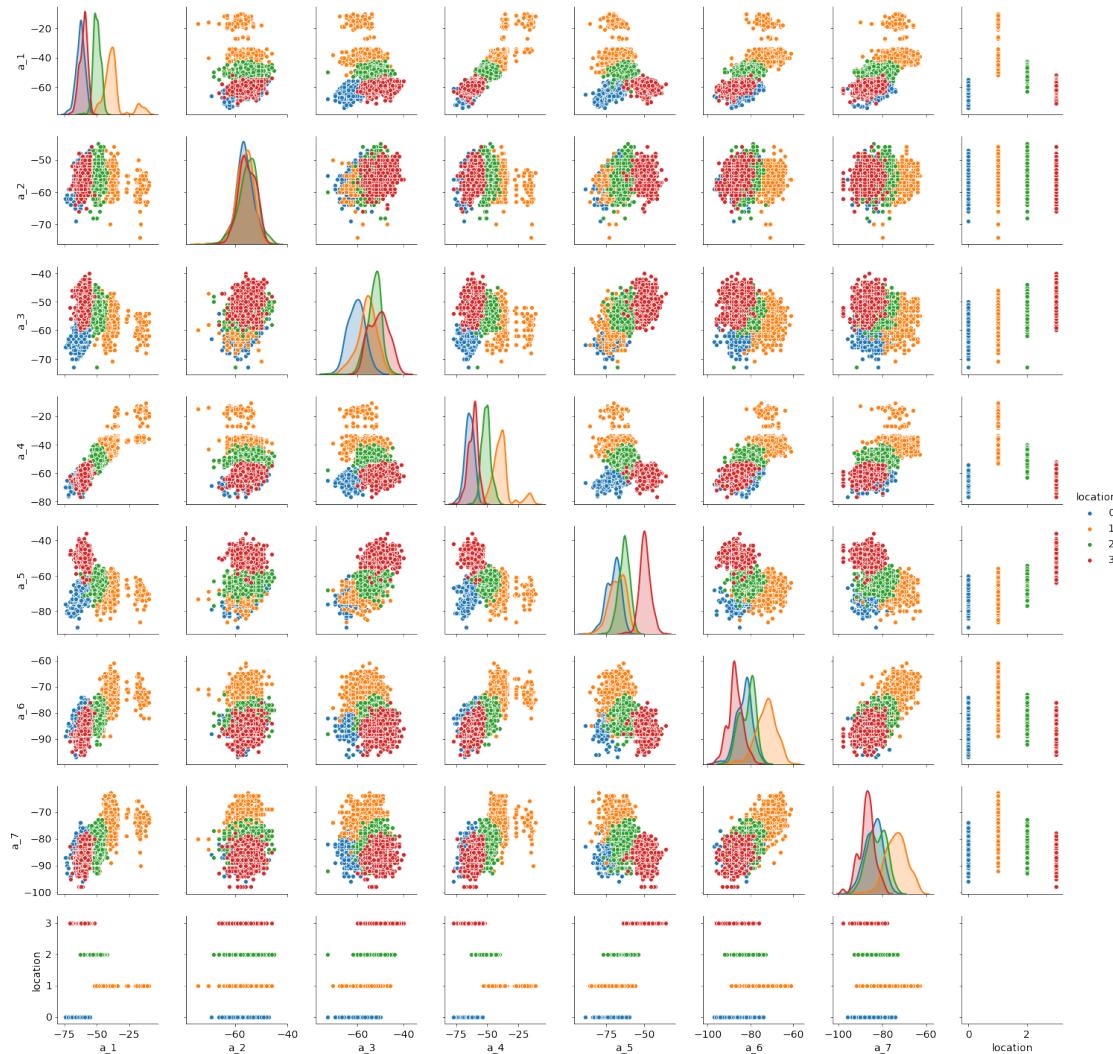
```
In [8]: sns.pairplot(df, hue='location');
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kde.py:488: RuntimeWarning: invalid value
```

```

encountered in true_divide
    binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kdetools.py:34: RuntimeWarning: invalid value
encountered in double_scalars
    FAC1 = 2*(np.pi*bw/RANGE)**2

```



## Model definition and training

It is very clear that the 4 locations are quite well defined and therefore we can hope to train a good model. Let's do that! First let's define our usual X and y arrays of features and labels:

```
In [9]: X = df.drop('location', axis=1).values
y = df['location'].values
```

Then we'll split our data into training and test, using a 25% test split.

```
In [10]: from sklearn.model_selection import train_test_split
```

```
In [11]: X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.25,
                    random_state=0)
```

Now let's build a fully connected model using the [Functional API in Keras](#). Let's import the Model class as well as a few layers:

```
In [12]: from keras.models import Model
        from keras.layers import Input, Dense, BatchNormalization
```

Using TensorFlow backend.

In particular notice that we'll use the BatchNormalization layer right after the input, since our features take negative large numbers, which may slow the convergence of our model. Let's build a fully connected model with the following architecture:

- Input
- Batch Normalization
- Fully connected inner layer with 50 nodes and a ReLU activation
- Fully connected inner layer with 30 nodes and a ReLU activation
- Fully connected inner layer with 10 nodes and a ReLU activation
- Output layer with 4 nodes and a Softmax activation

The functional API makes it very easy to build this model:

```
In [13]: inputs = Input(shape=X_train.shape[1:])
x = BatchNormalization()(inputs)
x = Dense(50, activation='relu')(x)
x = Dense(30, activation='relu')(x)
x = Dense(10, activation='relu')(x)
predictions = Dense(4, activation='softmax')(x)

model = Model(inputs=inputs, outputs=predictions)
```

Let's display a model summary and make sure that we have built exactly what we wanted:

In [14]: `model.summary()`

```
-----  
Layer (type)          Output Shape         Param #  
=====-----  
input_1 (InputLayer)    (None, 7)           0  
-----  
batch_normalization_1 (Batch (None, 7)       28  
-----  
dense_1 (Dense)        (None, 50)          400  
-----  
dense_2 (Dense)        (None, 30)          1530  
-----  
dense_3 (Dense)        (None, 10)          310  
-----  
dense_4 (Dense)        (None, 4)           44  
=====-----  
Total params: 2,312  
Trainable params: 2,298  
Non-trainable params: 14  
-----
```

Great! Now we can compile the model. Notice that since our labels are not one-hot encoded we should use the `sparse_categorical_crossentropy` instead of the usual `categorical_crossentropy`:

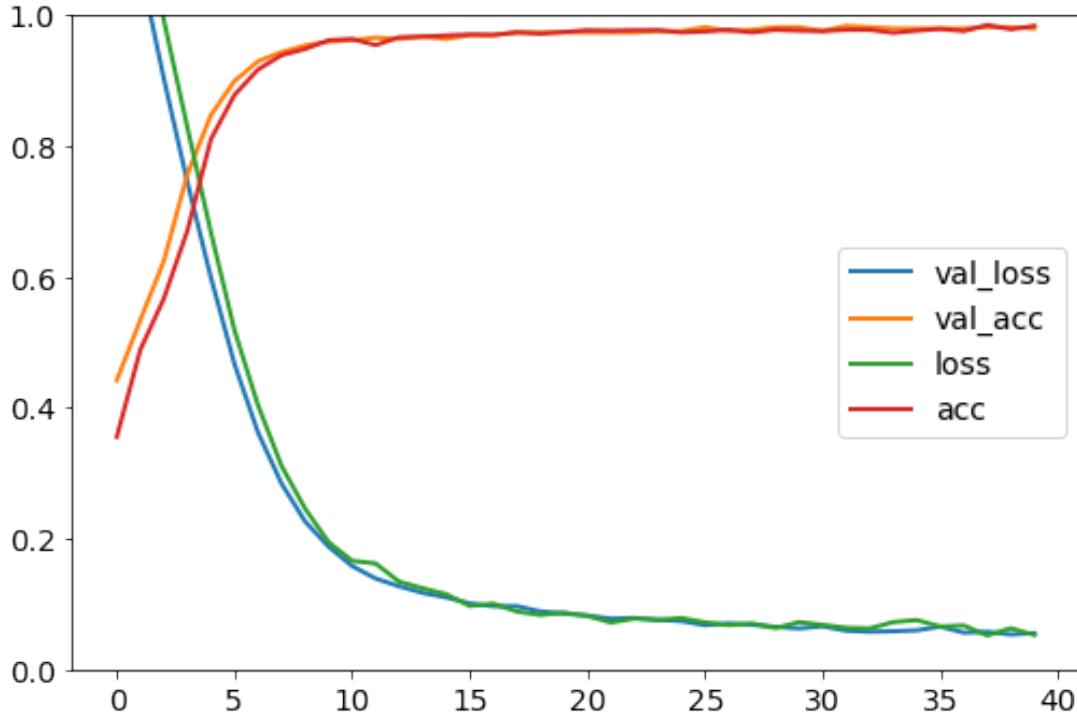
```
In [15]: model.compile('adam',  
                      'sparse_categorical_crossentropy',  
                      metrics=['accuracy'])
```

We are now ready to train the model. Let's train it for 40 epochs, using the test data to validate the performance:

```
In [16]: h = model.fit(X_train, y_train,  
                      batch_size=128,  
                      epochs=40,  
                      verbose=0,  
                      validation_data=(X_test, y_test))
```

As we have done several times in the book, we can display the history of training leveraging Pandas plotting capabilities:

```
In [17]: pd.DataFrame(h.history).plot()  
plt.ylim(0, 1);
```



The training graph looks very good. The model has converged to almost perfect accuracy and there is no sign of overfitting. This is great! We are ready to export the model for deployment.

### Export the model with Keras

Keras offers several ways to export a model. The simplest way is to save the model architecture and the weights as separate compressed file. You can read more about the various ways of saving a model [here](#). Let's start by importing the `os`, `json` and `shutil` packages:

```
In [18]: import os # Miscellaneous operating system interfaces
           import json # JSON encoder and decoder
           import shutil # High-level file operations
```

Next we define the output path to save our model. This path will be composed of three parts:

- a base path, in this case it's going to be `/tmp/ztdl_models/wifi/`
- a subpath, referring to the type of deployment system we'd like to use, here: `/flask`
- a version number, starting from 1. We use this in case we'd like to deploy a new version of the model later on.

```
In [19]: base_path = '/tmp/ztdl_models/wifi'
```

```
sub_path = 'flask'  
version = 1
```

Let's combine these in a single path using `join`:

```
In [20]: from os.path import join
```

```
In [21]: export_path = join(base_path, sub_path, str(version))  
export_path
```

```
Out[21]: '/tmp/ztdl_models/wifi/flask/1'
```

Next we create the export path. We delete it first and then re-create it as an empty path:

```
In [22]: shutil.rmtree(export_path, ignore_errors=True) # delete path, if exists  
os.makedirs(export_path) # create path
```

Now we are ready to save the model. Let's have a look at the json description of the model:

```
In [23]: json.loads(model.to_json())
```

```
Out[23]: {'class_name': 'Model',  
          'config': {'name': 'model_1',  
                     'layers': [{'name': 'input_1',  
                                 'class_name': 'InputLayer',  
                                 'config': {'batch_input_shape': [None, 7],  
                                           'dtype': 'float32',  
                                           'sparse': False,  
                                           'name': 'input_1'},  
                                 'inbound_nodes': []},  
                     {'name': 'batch_normalization_1',  
                      'class_name': 'BatchNormalization',  
                      'config': {'name': 'batch_normalization_1',  
                                 'trainable': True,  
                                 'axis': -1,  
                                 'momentum': 0.99,  
                                 'epsilon': 0.001,  
                                 'center': True,  
                                 'scale': True,  
                                 'beta_initializer': {'class_name': 'Zeros', 'config': {}}},  
                     'inbound_nodes': []},  
          'inbound_layers': []}
```

```
'gamma_initializer': {'class_name': 'Ones', 'config': {}},
'moving_mean_initializer': {'class_name': 'Zeros', 'config': {}},
'moving_variance_initializer': {'class_name': 'Ones', 'config': {}},
'beta_regularizer': None,
'gamma_regularizer': None,
'beta_constraint': None,
'gamma_constraint': None},
'inbound_nodes': [[[['input_1', 0, 0, {}]]]],

{'name': 'dense_1',
 'class_name': 'Dense',
 'config': {'name': 'dense_1',
 'trainable': True,
 'units': 50,
 'activation': 'relu',
 'use_bias': True,
 'kernel_initializer': {'class_name': 'VarianceScaling',
 'config': {'scale': 1.0,
 'mode': 'fan_avg',
 'distribution': 'uniform',
 'seed': None}},
 'bias_initializer': {'class_name': 'Zeros', 'config': {}},
 'kernel_regularizer': None,
 'bias_regularizer': None,
 'activity_regularizer': None,
 'kernel_constraint': None,
 'bias_constraint': None},
 'inbound_nodes': [[[['batch_normalization_1', 0, 0, {}]]]],

{'name': 'dense_2',
 'class_name': 'Dense',
 'config': {'name': 'dense_2',
 'trainable': True,
 'units': 30,
 'activation': 'relu',
 'use_bias': True,
 'kernel_initializer': {'class_name': 'VarianceScaling',
 'config': {'scale': 1.0,
 'mode': 'fan_avg',
 'distribution': 'uniform',
 'seed': None}},
 'bias_initializer': {'class_name': 'Zeros', 'config': {}},
 'kernel_regularizer': None,
 'bias_regularizer': None,
 'activity_regularizer': None,
 'kernel_constraint': None,
 'bias_constraint': None},
 'inbound_nodes': [[['dense_1', 0, 0, {}]]]},

{'name': 'dense_3',
 'class_name': 'Dense',
```

```
'config': {'name': 'dense_3',
    'trainable': True,
    'units': 10,
    'activation': 'relu',
    'use_bias': True,
    'kernel_initializer': {'class_name': 'VarianceScaling',
        'config': {'scale': 1.0,
            'mode': 'fan_avg',
            'distribution': 'uniform',
            'seed': None}},
    'bias_initializer': {'class_name': 'Zeros', 'config': {}},
    'kernel_regularizer': None,
    'bias_regularizer': None,
    'activity_regularizer': None,
    'kernel_constraint': None,
    'bias_constraint': None},
    'inbound_nodes': [[[['dense_2', 0, 0, {}]]]],
{'name': 'dense_4',
    'class_name': 'Dense',
    'config': {'name': 'dense_4',
        'trainable': True,
        'units': 4,
        'activation': 'softmax',
        'use_bias': True,
        'kernel_initializer': {'class_name': 'VarianceScaling',
            'config': {'scale': 1.0,
                'mode': 'fan_avg',
                'distribution': 'uniform',
                'seed': None}},
        'bias_initializer': {'class_name': 'Zeros', 'config': {}},
        'kernel_regularizer': None,
        'bias_regularizer': None,
        'activity_regularizer': None,
        'kernel_constraint': None,
        'bias_constraint': None},
        'inbound_nodes': [[[['dense_3', 0, 0, {}]]]]],
    'input_layers': [[[['input_1', 0, 0]]]],
    'output_layers': [[[['dense_4', 0, 0]]]],
    'keras_version': '2.2.2',
    'backend': 'tensorflow'}
```

Nice! The whole model is specified in a few lines! To save it we'll open a `model.json` file and then write to it the json version of the file:

```
In [24]: with open(join(export_path, 'model.json'), 'w') as fout:
    fout.write(model.to_json())
```

Next we save the weights. We do this with the `.save_weights` method of the model:

```
In [25]: model.save_weights(join(export_path, 'weights.h5'))
```

Let's check the content of the `export_path` using the `os.listdir` command:

```
In [26]: os.listdir(export_path, )
```

```
Out[26]: ['weights.h5', 'model.json']
```

As you can see there are 2 files, the json description of the model and the weights. Great! Let's see how one would re-load these into a new model. First we need to import the `model_from_json` function:

```
In [27]: from keras.models import model_from_json
```

Next we create a model by reading the json file:

```
In [28]: with open(join(export_path, 'model.json')) as fin:
    loaded_model = model_from_json(fin.read())
```

The loaded model has random weights, as we can verify by generating predictions on the test set and then comparing them with the labels. Notice that since the model was defined using the functional API, there is no `.predict_classes` method. Let's use the `.predict` method to obtain the probabilities for each class:

```
In [29]: probas = loaded_model.predict(X_test)
probas
```

```
Out[29]: array([[2.79596770e-05, 1.55901864e-20, 7.04144835e-01, 2.95827240e-01],
[1.34780967e-05, 2.74981338e-17, 3.98236483e-01, 6.01750016e-01],
[2.18204368e-05, 1.87449942e-18, 5.78068078e-01, 4.21910048e-01],
...,
[9.38129290e-07, 3.54882486e-18, 2.49536168e-02, 9.75045502e-01],
[1.16164018e-04, 1.68911121e-19, 8.20722282e-01, 1.79161638e-01],
[3.03749084e-06, 4.84038953e-19, 1.01790816e-01, 8.98206174e-01]],  
dtype=float32)
```

To retrieve the predicted classes we need to use the `argmax` function from Numpy:

```
In [30]: preds = np.argmax(probas, axis=1)  
preds
```

Finally we can check the accuracy of these prediction using the `accuracy_score` from Scikit-Learn:

```
In [31]: from sklearn.metrics import accuracy_score
```

```
In [32]: accuracy_score(y_test, preds)
```

Out [32] : 0.274

As expected, this model is not trained. Let's load the weights now:

```
In [33]: loaded_model.load_weights(join(export_path, 'weights.h5'))
```

And let's repeat the steps above:

```
In [34]: probas = loaded_model.predict(X_test) # class probabilities
        preds = np.argmax(probas, axis=1) # class prediction
        accuracy_score(y_test, preds) # accuracy score
```

Out[34]: 0.978

Great! The model is now using the trained weights, so we can use it for inference in deployment.

Notice that this model is not trainable. If you tried to run the command:

```
loaded_model.fit(X_train, y_train)
```

You would get a `RuntimeError` like this one:

```
RuntimeError: You must compile a model before training/testing. Use `model.compile(optimizer, lo
```

As the message explains, in order to train the model we need to compile it first, i.e. add to the graph all the operations concerning gradient calculation, loss calculation and optimizer. We don't need any of this for deployment, so let's not compile the model.

## A simple deployment with Flask

**WARNING:** The simple script we run here is not meant for production use. Please make sure to read how to deploy a Flask app to production in the [Flask documentation](#).

As the documentation says, [Flask](#) is a microframework for Python based on Werkzeug, Jinja 2, and good intentions. And before you ask: It's BSD licensed!

It's a very common choice for simple websites, APIs, and in general web development. We'll use it here to load our model in a very simple application that will be launched from a script.

We will here go through the commands that compose the script and tell you how to run it from the shell. Let's get started.

**TIP:** if you have installed the most recent version of our `ztdlbook` environment file, you should already have flask installed. Otherwise go back to [Chapter 1](#) and check the instructions on how to create or update the environment.

The script will first import the Flask and request classes. Flask is the main app, while request will be used to collect the data received by the app.

```
In [35]: from flask import Flask  
        from flask import request
```

We also import tensorflow which we'll need when loading the model:

```
In [36]: import tensorflow as tf
```

Then we define a few global variables, like the path of the model, the model and the tensorflow graph. These last two are initialized as None, they will be assigned later.

```
In [37]: export_path = '/tmp/ztdl_models/wifi/flask/1/'  
        loaded_model = None  
        graph = None
```

Next we create the flask app, which is also a global variable:

```
In [38]: app = Flask(__name__)
```

The next step is to define a load\_model function that loads the model from the export\_path like we did before:

```
In [39]: def load_model():  
    """  
        Load model and tensorflow graph  
        into global variables.  
    """  
  
    # global variables  
    global loaded_model  
    global graph  
  
    # load model architecture from json  
    with open(join(export_path, 'model.json')) as fin:  
        loaded_model = model_from_json(fin.read())  
  
    # load weights  
    loaded_model.load_weights(join(export_path, 'weights.h5'))
```

```
# get the tensorflow graph
graph = tf.get_default_graph()
print("Model loaded.")
```

The second function we define is a preprocess function that can be used to perform any normalization, feature engineering or other preprocessing. In the current scenario we use this function to convert the data from json to a Numpy array.

```
In [40]: def preprocess(data):
    """
    Generic function for normalization
    and feature engineering.
    Convert data from json to numpy array.
    """
    res = json.loads(data)
    return np.array(res['data'])
```

Next we define a function called predict, which performs the following operations:

- take the data from `request.data`
- preprocess the data with the `preprocess` function
- use the loaded model to predict probabilities
- extract the predicted classes from probabilities using `np.argmax`
- return a json version of the predictions

Notice that we will “decorate” this function with the decorator:

```
@app.route('/', methods=["POST"])
```

This method tells Flask that this function should be called when a POST request is received at the / route. For more information on how this is done in flask please make sure to check the [extensive documentation](#).

```
In [41]: @app.route('/', methods=["POST"])
def predict():
    """
    Generate predictions with the model
    when receiving data as a POST request
    """
    if request.method == "POST":
        # get data from the request
```

```

data = request.data

# preprocess the data
processed = preprocess(data)

# run predictions using the global tf graph
with graph.as_default():
    probas = loaded_model.predict(processed)

# obtain predicted classes from probabilities
preds = np.argmax(probas, axis=1)

# print in backend
print("Received data:", data)
print("Predicted labels:", preds)

return jsonify(preds.tolist())

```

Finally we complete the script with an if statement that runs the app in debug mode:

```

if __name__ == "__main__":
    print("* Loading model and starting Flask server...")
    load_model()
    app.run(host='0.0.0.0', debug=True)

```

Please note that this is not the preferred mode to run a flask app. Please refer to the [documentation](#) for more information.

## Full script

Let's take a look at the whole script using the `cat` shell command.

TIP: if this doesn't work on your system, simply open the script in your favorite text editor:

```
In [42]: !cat 13_flask_serve_model.py
#!/pygmentize -O style=monokai -g 13_flask_serve_model.py
```

```

import os
import json
import numpy as np

```

```
from keras.models import model_from_json

from flask import Flask
from flask import request, jsonify
import tensorflow as tf

loaded_model = None
graph = None

app = Flask(__name__)

def load_model(export_path):
    """
    Load model and tensorflow graph
    into global variables.
    """

    # global variables
    global loaded_model
    global graph

    # load model architecture from json
    with open(os.path.join(export_path, 'model.json')) as fin:
        loaded_model = model_from_json(fin.read())

    # load weights
    loaded_model.load_weights(os.path.join(export_path, 'weights.h5'))

    # get the tensorflow graph
    graph = tf.get_default_graph()
    print("Model loaded.")

def preprocess(data):
    """
    Generic function for normalization
    and feature engineering.
    Convert data from json to numpy array.
    """
    res = json.loads(data)
    return np.array(res['data'])

@app.route('/', methods=["POST"])
def predict():
    """
    Generate predictions with the model
    when receiving data as a POST request
    """
    if request.method == "POST":
        # get data from the request
        data = request.data

        # preprocess the data
        processed = preprocess(data)

        # run predictions using the global tf graph
        with graph.as_default():

            # build a session
            sess = tf.Session(graph=graph)

            # run the graph
            predictions = sess.run(loaded_model.predict(processed))

            # close the session
            sess.close()

            # return the predictions
            return jsonify(predictions)
```

```

    probas = loaded_model.predict(processed)

    # obtain predicted classes from predicted probabilities
    preds = np.argmax(probas, axis=1)

    # print in backend
    print("Received data:", data)
    print("Predicted labels:", preds)

    return jsonify(preds.tolist())

if __name__ == "__main__":
    from sys import argv
    print("* Loading model and starting Flask server...")
    if len(argv) > 1:
        export_path = argv[1]
    else:
        export_path = '/tmp/ztdl_models/wifi/flask/1/'
    load_model(export_path)
    app.run(host='0.0.0.0', debug=True)

```

## Run the script

We can run this script from the course folder as:

```
python 13_flask_serve_model.py
```

Make sure to check [Flask Documentation](#) if you encounter any issues with the above steps.

You should see the following output:

```

Using TensorFlow backend.
* Loading model and starting Flask server...
2018-06-18 11:34:31.339142: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports
Model loaded.
* Serving Flask app "13_flask_serve_model" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
Using TensorFlow backend.
* Loading model and starting Flask server...
2018-06-18 11:37:57.101110: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports
Model loaded.
* Debugger is active!
* Debugger PIN: xxx-xxx-xxx

```

## Get Predictions from the API

Now that the server is running, let's send some data to it and get predictions. We can test the application with a simple CURL request like:

```
curl -d '{"data": [[-62, -58, -59, -59, -67, -80, -77],  
                     [-49, -53, -50, -48, -67, -78, -88],  
                     [-52, -57, -49, -50, -66, -80, -80]]}' \  
-H "Content-Type: application/json" \  
-X POST http://localhost:5000
```

Which should return:

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

What did we just do? We have sent the wifi signal detected by 3 mobile phones and obtained their location. The first one is in zone 0 and the other two are in zone 2. Great!

We can also ping our API using Python from the notebook by importing the `requests` module:

```
In [43]: import requests
```

We set the `api_url` variable:

```
In [44]: api_url = "http://localhost:5000/"
```

Get a few points from the test dataset:

```
In [45]: data = X_test[:5].tolist()
```

```
In [46]: data
```

```
Out[46]: [[-62, -58, -59, -59, -67, -80, -77],  
           [-49, -53, -50, -48, -67, -78, -88],  
           [-52, -57, -49, -50, -66, -80, -80],  
           [-40, -55, -52, -43, -60, -76, -72],  
           [-64, -59, -51, -67, -43, -88, -92]]
```

Create payload and headers dictionaries:

```
In [47]: payload = {'data': data}
            headers = {'content-type': 'application/json'}
```

Finally send a post request to the `api_url` with our data in json format. We collect the request response into a `response` variable:

```
In [48]: response = requests.post(api_url,
                                  data=json.dumps(payload),
                                  headers=headers)
```

Let's check the response:

```
In [49]: response
```

```
Out[49]: <Response [200]>
```

If you see: `<Response [200]>` it means the request worked. Let's check the response we obtained:

```
In [50]: response.json()
```

```
Out[50]: [0, 2, 2, 1, 3]
```

We can compare that with our labels:

```
In [51]: y_test[:5]
```

```
Out[51]: array([0, 2, 2, 1, 3])
```

The deployed model is working pretty well! Very nice! There are many options to host your deployed model, including: - hosting the Flask app on [AWS](#), [GCloud](#), [Azure](#) - deploying it on [Heroku](#) - deploying it on [Floydhub](#)

Now go ahead and amaze your friends!

This chapter continues introducing a different way to export and deploy a model, which leverages [Tensorflow Serving](#). This is the preferred way for larger production deployments.

## Deployment with Tensorflow Serving

As the [documentation](#) says, TensorFlow Serving is a flexible, high-performance serving system for Machine Learning models, designed for production environments. TensorFlow Serving makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs. TensorFlow Serving provides out-of-the-box integration with TensorFlow models, but can be easily extended to serve other types of models and data.

Tensorflow Serving can accommodate both small and large deployments, and it is built for production. It is not as simple as Flask, and here we will barely scratch the surface of what it's possible with it. If you are serious about using it, we strongly recommend you take a look at the [Architecture overview](#) where many concepts like Servables, Managers and Sources are explained.

In this part of the book, we will just show you how to export a model for serving and how to ping a Tensorflow serving server. We will leave the full installation of Tensorflow serving for the end of the chapter. Installation is strongly dependent on the system you are using and is [well documented](#).

### Saving a model for Tensorflow Serving

Let's get started by exporting the model for Tensorflow Serving. Let's start by defining an export path:

```
In [52]: base_path = '/tmp/ztdl_models/wifi'
         sub_path = 'tf-serving'
         version = 1
```

Notice that we can bump up the version number if we save a new model later on. Like before, we can combine these:

```
In [53]: export_path = join(base_path, sub_path, str(version))
         export_path
```

```
Out[53]: '/tmp/ztdl_models/wifi/tfserving/1'
```

Let's clear the `export_path` in case it already exists:

```
In [54]: shutil.rmtree(export_path, ignore_errors=True)
```

Next, let's load the `SaveModelBuilder` class from `tensorflow.python.saved_model.builder`:

```
In [55]: from tensorflow.python.saved_model.builder import SavedModelBuilder
```

The `SaveModelBuilder` class provides functionality to build a `SavedModel` instance [protocol buffer](#). Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.

We'll use this instance to save our model. Before we get that far, let's create the instance:

```
In [56]: builder = SavedModelBuilder(export_path)
```

The builder allows multiple meta graphs (i.e. TF models) to be saved as part of a single language-neutral `SavedModel`, while sharing variables and assets. To build a `SavedModel`, the first meta graph must be saved with variables. Subsequent meta graphs will simply be saved with their graph definitions.

Each meta graph added to the `SavedModel` must be annotated with *tags*. The tags provide a means to identify the specific meta graph to load and restore, along with the shared set of variables and assets. Together with tags we will need to pass the session and a *signature map*. This is a dictionary that contains the methods that can be called when we call the model from the front-end. We'll need to create at least one signature to be able to use the model from within Tensorflow serving.

Let's start by creating a signature using the `predict_signature_def` from `tensorflow.python.saved_model.signature_def_utils`:

```
In [57]: from tensorflow.python.saved_model.signature_def_utils \
          import predict_signature_def
```

Now let's define a signature that references the `model.input` tensors as input and the `model.output` as output. By doing this we are tying the graph of our Keras model to the tensorflow serving builder.

```
In [58]: signature = predict_signature_def(
            inputs={"inputs": model.input},
            outputs={"outputs": model.output})
```

Next let's retrieve the [Tensorflow Session](#), which is the class that represents the connection between our Python client program and the C++ runtime. Keras makes the session available through the `backend` module. Let's import it:

```
In [59]: import keras.backend as K
```

And then let's retrieve the session using the `.get_session` method:

```
In [60]: sess = K.get_session()
```

We are now ready to use the builder to save our model. Let's do that:

```
In [61]: builder.add_meta_graph_and_variables(
    sess=sess,
    tags=[tf.saved_model.tag_constants.SERVING],
    signature_def_map={'predict': signature})
```

```
INFO:tensorflow:No assets to save.
INFO:tensorflow:No assets to write.
```

As we stated earlier, the first meta graph must be tagged, so we tagged our graph with the SERVING tag. Finally let's also write the SavedModel protocol buffer to disk using the `builder.save()` method:

```
In [62]: builder.save()
```

```
INFO:tensorflow:SavedModel written to:
/tmp/ztdl_models/wifi/tfserving/1/saved_model.pb
```

```
Out[62]: b'/tmp/ztdl_models/wifi/tfserving/1/saved_model.pb'
```

Let's take a look at the content of the export path:

```
In [63]: os.listdir(export_path)
```

```
Out[63]: ['variables', 'saved_model.pb']
```

As you can see there's a protocol buffer file `saved_model.pb` which contains the serialized model and a folder called `variables`. This folder contains the weights of the trained model that we have saved with the builder:

```
In [64]: os.listdir(join(export_path, 'variables'))
```

```
Out[64]: ['variables.index', 'variables.data-00000-of-00001']
```

## Inference with Tensorflow Serving using Docker and the Rest API

By far the easiest way to get tensorflow serving up and running is to use the pre-built [Docker image](#) as explained in the [documentation](#).

TIP: If you are new to Docker this may be a bit unfamiliar and complicate. Feel free to either skip this section or read more about Docker and how it works in the wonderful [documentation](#).

Assuming you have docker installed and running on your machine, let's pull the `tensorflow/serving` docker container:

```
docker pull tensorflow/serving
```

Next let's run the docker container with the following command:

```
docker run \
-v /tmp/ztdl_models/wifi/tfserving/:/models/wifi \
-e MODEL_NAME=wifi \
-e MODEL_PATH=/models/wifi \
-p 8500:8500 \
-p 8501:8501 \
-t tensorflow/serving
```

Let's go through the options selected in detail:

- `-v`: This bind mounts a volume, basically it tells docker to map the internal path `/models/wifi` to the `/tmp/ztdl_models/wifi/tfserving/` in our host computer.
- `-e`: Sets environment variables, in this case we set the `MODEL_NAME` and `MODEL_PATH` variables
- `-p`: Publishes a container's port to the host. In this case we are publishing port 8500 (default gRPC) and 8501 (default REST).
- `-t`: Allocate a pseudo-TTY
- `tensorflow/serving` is the name of the container we are running.

Since Tensorflow 1.8, Tensorflow serving comes with both a gRPC and REST endpoints by default, so we can test our running server by simply using curl. The correct command for this is:

```
curl -d '{"signature_name": "predict",
"instances": [[-62.0, -58.0, -59.0, -59.0, -67.0, -80.0, -77.0],
```

```

[-49.0, -53.0, -50.0, -48.0, -67.0, -78.0, -88.0],
[-52.0, -57.0, -49.0, -50.0, -66.0, -80.0, -80.0]]}' \
-H "Content-Type: application/json" \
-X POST http://localhost:8501/v1/models/wifi:predict

```

Go ahead and run that in a shell, you should receive an output that looks similar to the following:

```
{
  "predictions": [[0.997524, 1.19462e-05, 0.00171472, 0.000749083],
                  [3.40611e-06, 0.00262853, 0.997005, 0.000363284],
                  [2.52653e-05, 0.00507444, 0.993813, 0.00108718]
    ]
}
```

Wonderful! You have just ran your first model using Tensorflow Serving and Docker.

To stop the server, first press CTRL+C to exit from the tty session. Then run:

```
docker container ls
```

to list all the containers currently running. The output should look similar to this:

CONTAINER ID	IMAGE	COMMAND	...
fdd7c0958cdf	tensorflow/serving	"/bin/sh -c 'tensorf...'"	...

Find the id of the tensorflow/serving container and then run:

```
docker stop fdd7c0958cdf
```

To stop it from running. You can always restart it later if you need it.

## The gRPC API

Tensorflow serving can also receive data serialized as protocol buffers, so we will need to do a little bit more work in order to use our server for predictions.

First of all let's create a prediction service. We'll need to import the `insecure_channel` from `grpc`:

```
In [65]: from grpc import insecure_channel
```

Next let's create an insecure channel to localhost (or to your server) on port 8500, which is the port we chose for tensorflow serving:

```
In [66]: channel = insecure_channel('localhost:8500')
```

```
In [67]: channel
```

```
Out[67]: <grpc._channel.Channel at 0x7f53606f8a20>
```

Through this channel we'll be able to perform RPCs. Next we are going to create an instance of `PredictionServiceStub` from `tensorflow_serving.apis.prediction_service_pb2_grpc`. Notice that most of the documentation you can find online is outdated and uses the legacy beta API. We are using the most recent version of the gRPC API:

```
In [68]: from tensorflow_serving.apis.prediction_service_pb2_grpc import PredictionServiceStub
```

A `PredictionService` provides access to machine-learned models loaded by `model_servers`. Let's create a stub:

```
In [69]: stub = PredictionServiceStub(channel)
```

We are almost ready to send data to our server. The last thing we need to do is convert the data to protocol buffers. Let's use the same data we have used for the Flask example:

```
In [70]: data
```

```
Out[70]: [[-62, -58, -59, -59, -67, -80, -77],  
          [-49, -53, -50, -48, -67, -78, -88],  
          [-52, -57, -49, -50, -66, -80, -80],  
          [-40, -55, -52, -43, -60, -76, -72],  
          [-64, -59, -51, -67, -43, -88, -92]]
```

Let's convert this data to Protocol Buffers. First we import the `make_tensor_proto` function from Tensorflow:

```
In [71]: from tensorflow.contrib.util import make_tensor_proto
```

Then we use it to serialize our data. Notice that we will need wrap our data in a numpy array since it was passed as a list to the Flask application:

```
In [72]: data_np = np.array(data)
```

Let's make the protobufs:

What do protobufs look like? Let's print out `data_pb`:

In [74]: data\_pb

As you can see it's a binary file, with a text header. In the header we can read the data type and the tensor shape, while the values have been converted to binary values. Now that we have prepared the data, we are ready to create an instance of `PredictRequest`, which is a class in `tensorflow.serving.apis.predict_pb2`:

```
In [75]: from tensorflow_serving.apis.predict_pb2 import PredictRequest
```

```
In [76]: request = PredictRequest()
```

When we started our tensorflow serving server, we specified wifi as the model name, so let's use wifi as the model name for the request:

```
In [77]: request.model.spec.name = 'wifi'
```

Let's also indicate the signature name, which is predict:

```
In [78]: request.model_spec.signature_name = 'predict'
```

Finally let's pass our serialized data to the request input:

```
In [79]: request.inputs['inputs'].CopyFrom(data_pb)
```

In [80]: request

```
Out[80]: model_spec {
    name: "wifi"
    signature_name: "predict"
}
inputs {
    key: "inputs"
    value {
        dtype: DT_FLOAT
        tensor_shape {
            dim {
                size: 5
            }
            dim {
                size: 7
            }
        }
        tensor_content: "\000\000x\302\000\000h\302\000\0001\302\000\0001\302\000\000\206\3
    }
}
```

Great! Now let's pass the request to the `Stub.future` method which will invoke the underlying RPC asynchronously. This method returns an object that is both a Call for the RPC and a Future. In the event of RPC completion, the return Call-Future's result value will be the response message of the RPC. Should the event terminate with non-OK status, the returned Call-Future's exception value will be an `RpcError`.

```
In [81]: result_future = stub.Predict.future(request, 5.0)
```

Let's get the result of this future:

```
In [82]: result = result_future.result()
         result
```

```
Out[82]: outputs {
    key: "outputs"
    value {
        dtype: DT_FLOAT
        tensor_shape {
            dim {
                size: 5
            }
            dim {
                size: 4
            }
        }
        float_val: 0.9939389228820801
        float_val: 0.00017752652638591826
        float_val: 0.000844559574034065
        float_val: 0.005038977600634098
        float_val: 2.9748796805506572e-05
        float_val: 0.006226719357073307
        float_val: 0.9936287999153137
        float_val: 0.00011463184637250379
        float_val: 0.0005520731210708618
        float_val: 0.00699754199013114
        float_val: 0.9918016791343689
        float_val: 0.0006487205973826349
        float_val: 2.8535052933875704e-06
        float_val: 0.9599703550338745
        float_val: 0.03942705690860748
        float_val: 0.0005996639374643564
        float_val: 3.383963189662609e-07
        float_val: 1.808920160328853e-06
        float_val: 1.0559380037022947e-08
        float_val: 0.9999978542327881
    }
}
model_spec {
    name: "wifi"
    version {
        value: 1
    }
    signature_name: "predict"
}
```

Wonderful! Our Tensorflow server returned the predicted probabilities. We can convert them back to a familiar numpy array using the `make_ndarray` from `tensorflow.contrib.util`:

```
In [83]: from tensorflow.contrib.util import make_ndarray
```

```
In [84]: scores = make_ndarray(result.outputs['outputs'])
```

Here we are, back with our familiar array:

```
In [85]: scores
```

```
Out[85]: array([[9.9393892e-01, 1.7752653e-04, 8.4455957e-04, 5.0389776e-03],
 [2.9748797e-05, 6.2267194e-03, 9.9362880e-01, 1.1463185e-04],
 [5.5207312e-04, 6.9975420e-03, 9.9180168e-01, 6.4872060e-04],
 [2.8535053e-06, 9.5997036e-01, 3.9427057e-02, 5.9966394e-04],
 [3.3839632e-07, 1.8089202e-06, 1.0559380e-08, 9.9999785e-01]],  
 dtype=float32)
```

We can retrieve the classes by using argmax, like we previously did:

```
In [86]: prediction = np.argmax(scores, axis=1)  
prediction
```

```
Out[86]: array([0, 2, 2, 1, 3])
```

and we can compare this with the local model we still have in memory:

```
In [87]: model.predict(np.array(data)).argmax(axis=1)
```

```
Out[87]: array([0, 2, 2, 1, 3])
```

Wonderful! We have successfully retrieved predictions from a Tensorflow serving server. This barely scratches the surface of what's possible with Tensorflow Serving. If you are serious about bringing your models to production we strongly encourage you to read the [Documentation](#) as well as to complete the [Basic Tutorial](#) and the [Advanced Tutorial](#).

Next we briefly explain how install Tensorflow Serving on your system.

## Tensorflow Serving installation

If you don't want to use docker, you can still run Tensorflow Model Server natively on your machine.

The installation of Tensorflow Serving depends on the system you are using. The [installation guide](#). There are 2 methods, one involves installing Bazel and compiling Tensorflow Serving, the other leverages the

Python packages. We will choose this method, which is simpler, and show you how to complete it on Ubuntu Linux 16.04 with system Python.

First of all let's install all the required dependencies:

```
sudo apt-get update && sudo apt-get install -y \
    build-essential \
    curl \
    libcurl3-dev \
    git \
    libfreetype6-dev \
    libpng12-dev \
    libzmq3-dev \
    pkg-config \
    python-dev \
    python-numpy \
    python-pip \
    software-properties-common \
    swig \
    zip \
    zlib1g-dev
```

Next we install the Model Server. This requires adding the TensorFlow Serving distribution URI as a package source (one time setup):

```
echo "deb [arch=amd64]" \
    "http://storage.googleapis.com/tensorflow-serving-apt" \
    "stable tensorflow-model-server" \
    "tensorflow-model-server-universal" \
| sudo tee /etc/apt/sources.list.d/tensorflow-serving.list

curl https://storage.googleapis.com/tensorflow-serving-apt/ \
    tensorflow-serving.release.pub.gpg | sudo apt-key add -
```

and then installing the tensorflow-model-server package:

```
sudo apt-get update && sudo apt-get install tensorflow-model-server
```

Now you are ready to serve your models, simply start the Model Server by running:

```
tensorflow_model_server --port=8500 \
    --model_name=wifi \
    --model_base_path=/tmp/ztdl_models/wifi/tfserving/
```

This command specifies the port to use for serving, the `model_name` we are giving to this model and the base path where the version folders are saved. If this command runs correctly you should see log messages from Tensorflow serving indicating that it has found the model:

and you should see the following log messages:

```
Successfully loaded servable version {name: wifi version: 1}
Running ModelServer at 0.0.0.0:8500 ...
```

At this point you'll be ready to generate predictions! Yeah!

If the machine where you installed the model server is not your local machine, you'll need to pack your local model and upload it to your remote machine, for example using `tar & scp`:

```
cd /tmp/ztdl_models/wifi
tar -czvf tf-serving.tgz tf-serving
scp tf-serving.tgz <your-ubuntu-user>@<your-remote-ip>:~/
```

Then connect to the remote machine and unpack the model:

```
mkdir -p /tmp/ztdl_models/wifi/
tar -xvzf tf-serving.tgz -C /tmp/ztdl_models/wifi/
```

Finally start the server on the remote machine and make sure that port 8500 is accessible (either by setting firewall rules or by tunnelling through SSH).

If you want to run the model server on your local machine and/or optimize the performance of the model server or you'll need to install Bazel and compile tensorflow serving. We encourage you to take a look at the Documentation for [Bazel](#) and for [Tensorflow Serving Setup](#).

Assuming you have completed the installation of Tensorflow Serving on your system (see [the end of this Chapter for information](#)), you just need to run the command:

This concludes our Chapter on Deployment!

## Exercises

### Exercise 1

Let's deploy an image recognition API using Tensorflow Serving. The main difference from the API we have deployed in this chapter is that we will have to deal with how to pass an image to the model through

tensorflow serving. Since this chapter focuses on deployment, we will take a shortcut and deploy a pre-trained model that uses Imagenet. In particular we will deploy the Xception model. If you are unsure about how to use pre-trained model, please go back to [Chapter 11](#) for a refresher.

Here are the steps you will need to complete:

- load the model in keras
- export the model for tensorflow serving:
  - set the learning phase to zero
  - choose the right inputs and outputs
  - choose the right prediction signature
- run the model server
- write a short script that:
  - loads an image
  - pre-processes it with the appropriate function
  - serializes the image to protobuf
  - sends the image to the server
  - receives a prediction
  - decodes the prediction with keras decode\_prediction function

## Exercise 2

The above method of serving a pre-trained model has an issue: we are doing pre-processing and prediction decoding on the client side. This is actually not a best practice, because it requires the client to be aware of what kind of pre-processing and decoding functions the model needs.

We would like a server that takes the image as it is and returns a string with the name of the object in the image.

The easy way to do this is to use the Flask app implementation we have shown in this chapter and move pre-processing and decoding on the server side.

Go ahead and build a Flask version of the API that takes an image url as a json string, applies pre-processing, runs and decodes the prediction and returns a string with the response.

You will not use tensorflow serving for this exercise.

Once your script is ready, save it as `13_flask_serve_xception.py`, run it as:

and test the prediction with the following command:

```
curl -d 'http://bit.ly/2wb7uqN' \
-H "Content-Type: application/json" \
-X POST http://localhost:5000
```

If you've done things correctly, this should return:

```
"king_penguin"
```

**Disclaimer: this script is not meant for production purposes. Retrieving a file from a URL is not secure and you should avoid building an API that retrieves a file from a URL provided from the client. Here we used the url retrieval trick in order to make the curl command shorter.**



# 14

## Appendix

```
In [1]: with open('common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('matplotlibconf.py') as fin:  
    exec(fin.read())
```

Throughout the book we use several mathematical concepts drawn from [linear algebra](#) and [calculus](#). In this appendix we review them in little more detail. This is meant to be for the curious reader and it's not necessary in order to complete the book.

### Matrix multiplication

We have introduced matrix in chapter 1. As you know an  $N \times M$  matrix is an array of numbers organized in  $N$  rows and  $M$  columns.

Matrices are multiplied with the same rule of the dot product. Two matrices  $A$  and  $B$  can be multiplied if the number of columns of the first is equal to the number of rows of the second. If  $A$  is  $2 \times 3$  and  $B$  is  $3 \times 2$ , they can be multiplied and the resulting matrix will have shape  $2 \times 2$  if we do  $A \cdot B$  and  $3 \times 3$  if we do  $B \cdot A$ .

However, if  $A$  is  $2 \times 4$  and  $B$  is  $3 \times 5$ , we **cannot** multiply the two matrices.

The figure below shows how the elements of this matrix are calculated:

Let's create 2 matrices in numpy using 2D-array method and check this formula:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{bmatrix}$$

$$A \cdot B = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{bmatrix} =$$

$$\begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} & a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} \\ a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} & a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} \end{bmatrix}$$

```
In [3]: A = np.array([[0, 1, 2],
                   [2, 3, 0]])

B = np.array([[0, 1],
              [2, 3],
              [4, 5]])

C = np.array([[0, 1],
              [2, 3],
              [4, 5],
              [0, 1],
              [2, 3],
              [4, 5]])

print("A is a {} matrix".format(A.shape))
print("B is a {} matrix".format(B.shape))
print("C is a {} matrix".format(C.shape))
```

A is a (2, 3) matrix  
B is a (3, 2) matrix  
C is a (6, 2) matrix

The matrix product in Numpy is a function called `dot`. We can access it as a method of an array:

```
In [4]: A.dot(B)
```

```
Out[4]: array([[10, 13],  
               [ 6, 11]])
```

or as a function in Numpy:

```
In [5]: np.dot(A, B)
```

```
Out[5]: array([[10, 13],  
               [ 6, 11]])
```

Notice that if we invert the order we do get a  $3 \times 3$  matrix instead:

```
In [6]: B.dot(A)
```

```
Out[6]: array([[ 2,  3,  0],  
               [ 6, 11,  4],  
               [10, 19,  8]])
```

Or, using the `np.dot()` version, we get the same as these two methods are functionally equivalent:

```
In [7]: np.dot(B, A)
```

```
Out[7]: array([[ 2,  3,  0],  
               [ 6, 11,  4],  
               [10, 19,  8]])
```

We can also perform the matrix multiplication `C.dot(A)`, however, matrix multiplications are only possible along axes with the same length. So, for example, we cannot perform the multiplication `A.dot(C)`.

```
In [8]: C.dot(A)
```

```
Out[8]: array([[ 2,  3,  0],
   [ 6, 11,  4],
   [10, 19,  8],
   [ 2,  3,  0],
   [ 6, 11,  4],
   [10, 19,  8]])
```

For example, uncomment the next line to get the following error:

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-11-9c5b5a616184> in <module>()
      1 # uncomment the next line to get an error
----> 2 A.dot(C)

ValueError: shapes (2,3) and (6,2) not aligned: 3 (dim 1) != 6 (dim 0)
```

In [9]: # A.dot(C)

TIP: remember this `ValueError` for mismatching shapes. It's very common when building Neural Networks.

## Chain rule

### Univariate functions

The [chain rule](#) is a rule to calculate the derivative of nested functions.

For example, let's say we have a function:

$$h(x) = \log(2 + \cos(x)) \quad (14.1)$$

How do we calculate the derivative of this function with respect to  $x$ ? This function is a composition of the two functions

$$f(y) = \log(y) \quad g(x) = 2 + \cos(x) \quad (14.2)$$

so we can write  $h$  as a nested function:

$$h(x) = f(g(x)) \quad (14.3)$$

We can calculate the derivative of  $h$  with respect to  $x$  by applying the *chain rule*:

- first we calculate the derivative of  $g$  with respect to  $x$
- then we calculate the derivative of  $f$  with respect to  $g$
- finally we multiply the two

$$\frac{dh(x)}{dx} = \frac{d}{dx}f(g(x)) = \frac{df}{dg} \cdot \frac{dg}{dx} \quad (14.4)$$

Using the table above we find:

$$\frac{d}{dx}(2 + \cos(x)) = -\sin(x) \quad \frac{d}{dg}\log(g) = \frac{1}{g} \quad (14.5)$$

So finally, the derivative of our nested function  $h$  of  $x$  is the product of the two derivatives:

$$\frac{d}{dx}\log(2 + \cos(x)) = \frac{-\sin(x)}{2 + \cos(x)} \quad (14.6)$$

Notice that we substituted  $g$  with  $2 + \cos(x)$  in the denominator.

### Code example

Let's define all the above functions and verify that the derivative of  $h$  calculated with the chain rule is equivalent to the derivative calculated with the `np.diff` function.

```
In [10]: def f(x):
    return np.log(x)

def g(x):
    return 2 + np.cos(x)

def h(x):
    return f(g(x))

def df(x):
    return 1/x
```

```
def dg(x):
    return -np.sin(x)

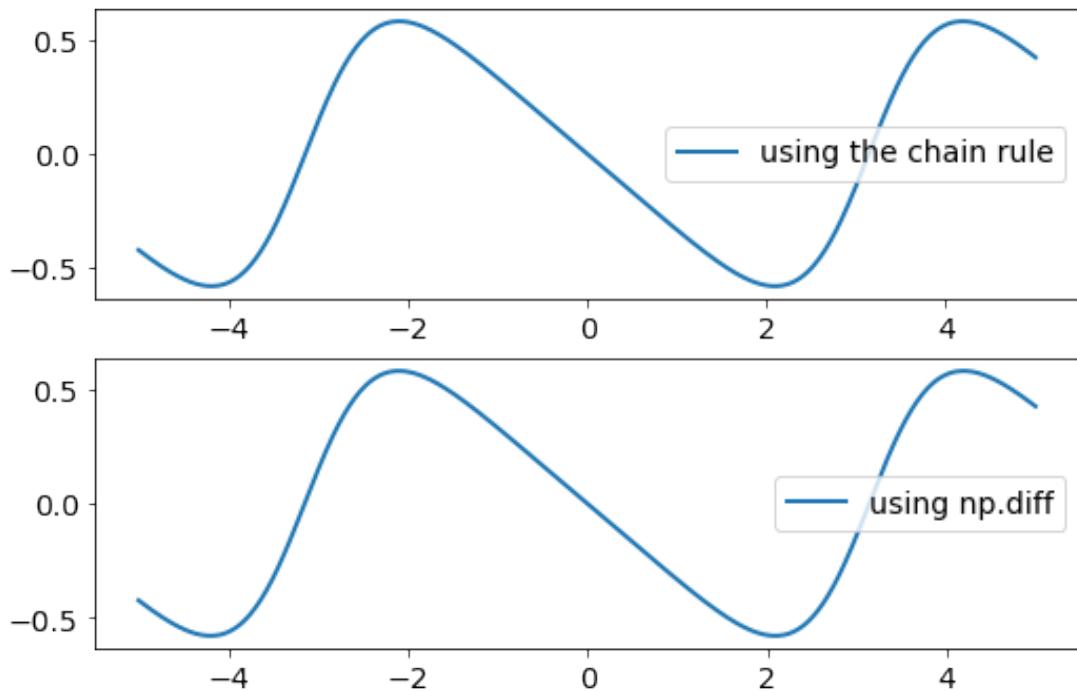
def dh(x):
    return df(g(x)) * dg(x)

In [11]: x = np.linspace(-5, 5, 500)

plt.subplot(211)
plt.plot(x, dh(x))
plt.legend(['using the chain rule'])

plt.subplot(212)
plt.plot(x[:-1], np.diff(h(x))/np.diff(x))
plt.legend(['using np.diff'])
```

Out[11]: <matplotlib.legend.Legend at 0x7fd80a60dfd0>



The two results are the same (minus numerical errors), as expected!

## Multivariate functions

The chain rule can be easily extended to the case where  $f$  has multiple functions as arguments:

$$h(x) = f(g(x), k(x)) \quad (14.7)$$

We simply distribute the chain rule to all the arguments that depend on  $x$ :

$$\frac{dh(x)}{dx} = \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial k} \cdot \frac{dk}{dx} \quad (14.8)$$

Notice that here we are using the [partial derivative](#) symbol  $\partial$  which simply means we are taking the derivative with respect to one of the variables while keeping all the others fixed.

## Exponentially Weighted Moving Average (EWMA)

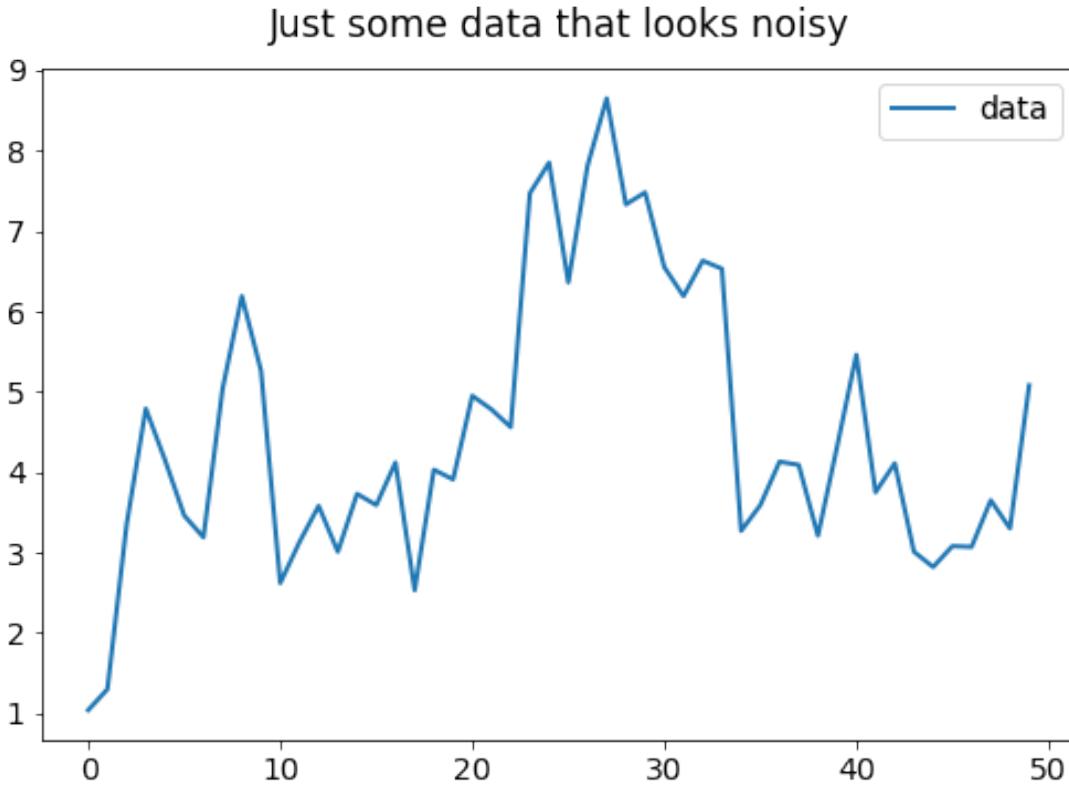
The [EWMA is the most important algorithm of your life](#). We often use this joke in classes to get the attention of our students. Although this may or may not be true in your particular case, it is true that this algorithm crops up everywhere, from financial time series to signal processing to Neural Networks.

Different domains name it in different ways but it's actually always the same thing and it's worth knowing how it works in detail.

Let us have a look at how it works. Let's say we have a sequence of ordered datapoints. These could be the values of a stock, temperature measurements, anything that is measured in a sequence.

```
In [12]: points = pd.DataFrame([1.04, 1.30, 3.35, 4.79, 4.15,
                               3.46, 3.19, 5.04, 6.19, 5.26,
                               2.62, 3.13, 3.58, 3.01, 3.73,
                               3.59, 4.12, 2.53, 4.03, 3.91,
                               4.95, 4.78, 4.56, 7.47, 7.85,
                               6.36, 7.81, 8.65, 7.33, 7.48,
                               6.55, 6.19, 6.63, 6.53, 3.27,
                               3.59, 4.13, 4.09, 3.21, 4.32,
                               5.46, 3.75, 4.11, 3.01, 2.82,
                               3.08, 3.07, 3.65, 3.30, 5.08],
                               columns=['data'])

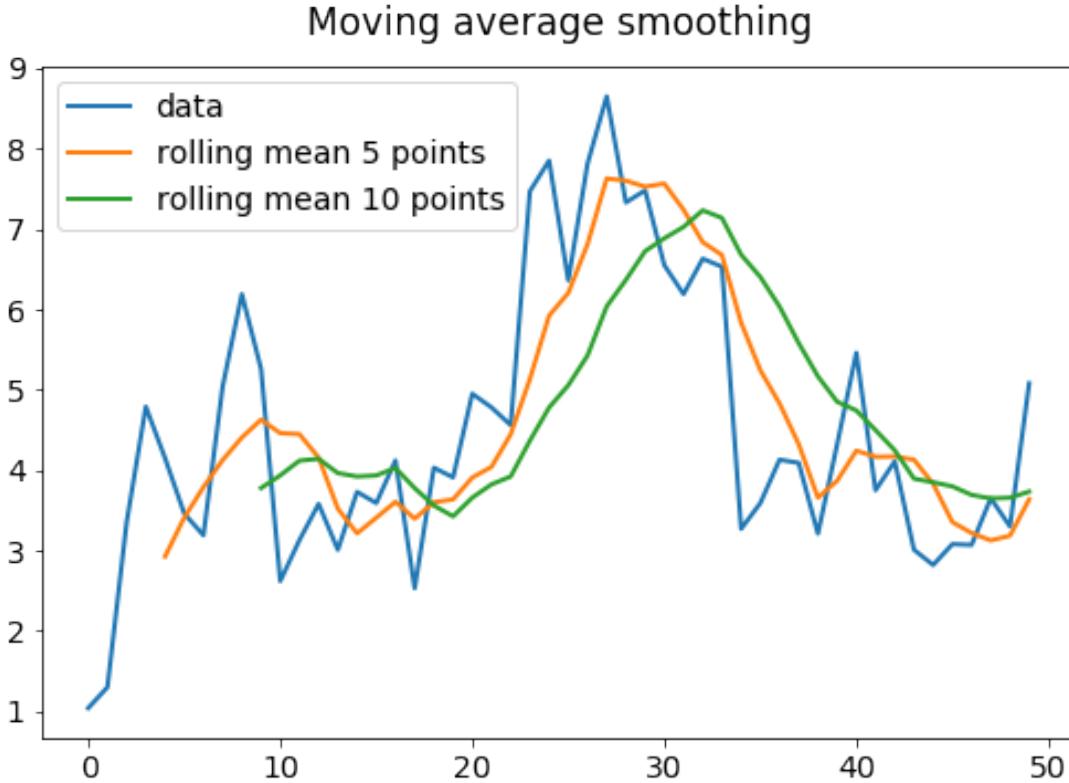
points.plot(title='Just some data that looks noisy')
plt.show()
```



If this data is noisy, we may want to reduce the noise in order to obtain a more accurate estimation of the underlying actual values. One easy way to remove noise from a time series is to perform a **rolling average** or **moving average**: you wait to accumulate a certain number of observations and use their average as the estimation of the current value. This method works, but it requires to hold the past values in a memory buffer and constantly update such buffer when a new data point of the sequence arrives. So if we want to average over a long window, we have to keep the whole window in memory, and also we cannot calculate the first average until we have observed at least as many points as the window contains (unless we pad with zeros).

Rolling averages are available in Pandas through the `.rolling()` method. Let's plot a few examples:

```
In [13]: points['rolling mean 5 points'] = \
    points['data'].rolling(5).mean()
points['rolling mean 10 points'] = \
    points['data'].rolling(10).mean()
points.plot(title='Moving average smoothing')
plt.show()
```



EWMA differs from the moving average because it only requires knowledge of the previous value of the data and of the current value of the EWMA itself.

Let's indicate the values of our sequence as  $x_0, x_1, x_2, \dots, x_n$ . We can calculate the value of the corresponding EWMA recursively as:

$$y_0 = x_0 \quad (14.9)$$

$$y_n = (1 - \alpha) y_{n-1} + \alpha x_n \quad (14.10)$$

The two extreme cases of this formula are  $\alpha = 0$ , in which case the value of  $y_n$  will remain fixed to  $x_0$  forever and  $\alpha = 1$ , in which case  $y_n$  will be exactly tracking the value of  $x_n$ .

If  $\alpha$  is between 0 and 1, the EWMA will smooth the signal reducing its fluctuations. Let's walk through an example with  $\alpha = 0.9$  to clarify how it works.

When the first point  $x_0$  comes in, the EWMA is set to be equal to the raw data, so  $y_0 = x_0$ .

Then, the second raw value  $x_1$  comes in, we take 90% of it and add it to 10% of the previous value of the moving average  $y_0$ :

$$y_1 = 0.1 y_0 + 0.9 x_1 \quad (14.11)$$

Since,  $y_0 = x_0$ , the previous formula is equivalent to:

$$y_1 = 0.1 x_0 + 0.9 x_1 \quad (14.12)$$

So, the value of the EWMA will be almost equal to the initial value, with 90% contribution from the new value  $x_1$ .

Then, the third point  $x_2$  comes in. Again, we take 90% of its value and add it to 10% of the current EWMA value  $y_1$ .

$$y_2 = 0.1 y_1 + 0.9 x_2 \quad (14.13)$$

$$= 0.1(0.1 x_0 + 0.9 x_1) + 0.9 x_2 \quad (14.14)$$

$$= 0.01 x_0 + 0.09 x_1 + 0.9 x_2 \quad (14.15)$$

$$(14.16)$$

This third point will still be mostly influenced by the initial point, but it will also contain contributions from the most recent two points.

Let's look at a couple more steps. Here's  $y_3$ :

$$y_3 = 0.1 y_2 + 0.9 x_3 \quad (14.17)$$

$$= 0.1(0.01 x_0 + 0.09 x_1 + 0.9 x_2) + 0.9 x_3 \quad (14.18)$$

$$= 0.001 x_0 + 0.009 x_1 + 0.09 x_2 + 0.9 x_3 \quad (14.19)$$

$$(14.20)$$

and here's  $y_4$ :

$$y_4 = 0.1 y_3 + 0.9 x_4 \quad (14.21)$$

$$= 0.1(0.001 x_0 + 0.009 x_1 + 0.09 x_2 + 0.9 x_3) + 0.9 x_4 \quad (14.22)$$

$$= 0.0001 x_0 + 0.0009 x_1 + 0.009 x_2 + 0.09 x_3 + 0.9 x_4 \quad (14.23)$$

As you can see the value of  $y_4$  is influenced by all the previous values of  $x$  in an exponentially decreasing fashion.

We can continue playing this game at each new point, and all we need to keep in memory is the previous value of the EWMA  $y_{n-1}$  until we have mixed it with the current raw value of the signal  $x_n$ .

This formula is great for two reasons:

1. We only keep the last values of the EWMA in memory, no need for a buffer.
2. We can calculate it from the beginning of the sequence instead of waiting to accumulate some values.

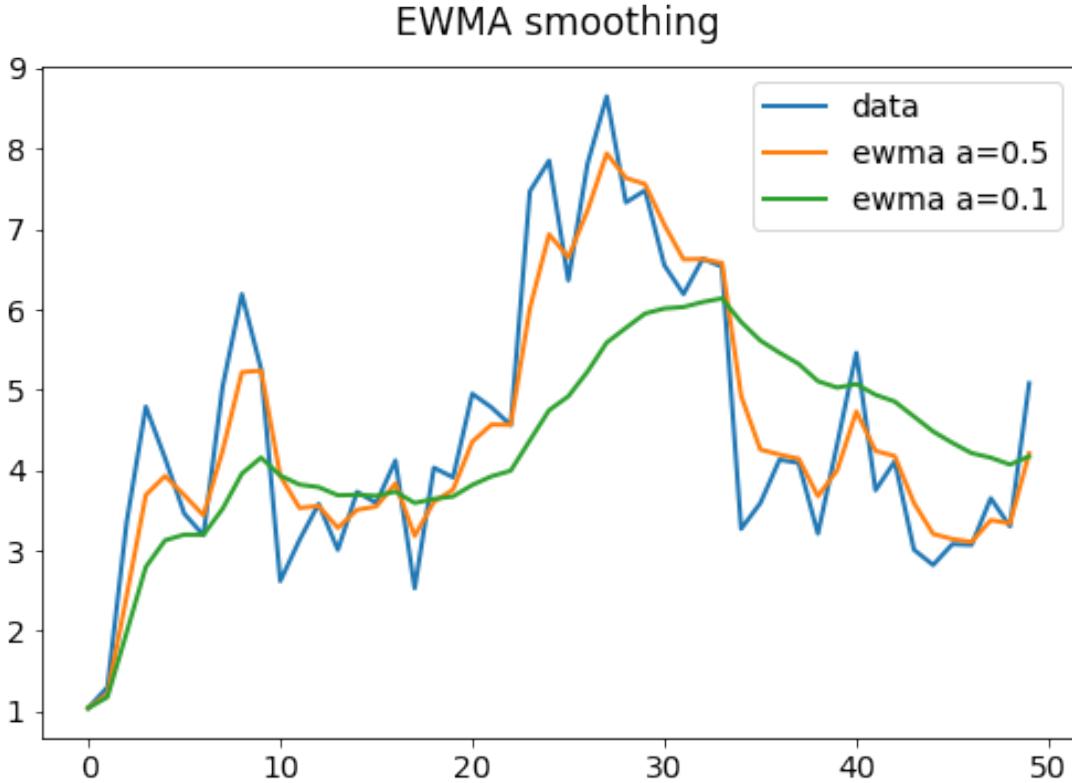
This formula is very popular and goes under different names in different domains. Statisticians would call it an **autoregressive integrated moving average** model with no constant term or (ARIMA)  $(0, 1, 1)$ . Signal processing people would call it a first order **Infinite Impulse Response** (IIR) filter, but it's the same thing.

The idea is simple. Each new value of the smoothed sequence is the sum of two terms: its own previous value and the current new value of the sequence. The ratio of the mixing is controlled by the parameter  $\alpha$ : very large values will skew the mix towards the raw data, with very little smoothing, very small  $\alpha$  (pronounced *alpha*) will skew the mix towards the previous smoothed value, therefore with very strong smoothing.

EWMAs are also available in pandas, let's plot a few:

```
In [14]: points['ewma a=0.5'] = points['data'].ewm(alpha=0.5).mean()
points['ewma a=0.1'] = points['data'].ewm(alpha=0.1).mean()

cols_ = ['data', 'ewma a=0.5', 'ewma a=0.1']
points[cols_].plot(title='EWMA smoothing')
plt.show()
```



You can notice a couple of things when comparing this plot with the previous one:

1. the smoothed curves start immediately, we don't have to wait in order to calculate the EWMA
  - a smaller value for alpha gives a stronger smoothing

This algorithm is simple and beautiful, and you will encounter it in many places, beyond optimizers for neural nets.

## Tensors

Let's create a couple of test tensors. We will create a tensor A of order 4 and a tensor B of order 2:

```
In [15]: A = np.random.randint(10, size=(2, 3, 4, 5))
         B = np.random.randint(10, size=(2, 3))
```

```
In [16]: A
```

```
Out[16]: array([[[[4, 4, 8, 4, 9],  
                  [1, 8, 6, 2, 9],  
                  [6, 6, 3, 1, 7],  
                  [4, 7, 0, 5, 1]],  
  
                 [[2, 7, 3, 8, 2],  
                  [7, 4, 7, 7, 2],  
                  [0, 4, 1, 5, 6],  
                  [3, 3, 7, 9, 4]],  
  
                 [[0, 0, 8, 7, 7],  
                  [7, 2, 2, 5, 6],  
                  [1, 1, 6, 5, 5],  
                  [3, 2, 2, 3, 9]]],  
  
                [[[1, 1, 4, 4, 2],  
                  [6, 6, 1, 0, 0],  
                  [2, 0, 1, 8, 7],  
                  [4, 1, 7, 2, 5]],  
  
                 [[6, 4, 7, 6, 9],  
                  [2, 4, 2, 7, 2],  
                  [5, 3, 2, 6, 4],  
                  [5, 2, 8, 3, 9]],  
  
                 [[3, 6, 9, 5, 0],  
                  [4, 8, 7, 8, 4],  
                  [9, 7, 9, 7, 6],  
                  [2, 9, 1, 4, 9]]]])
```

```
In [17]: B
```

```
Out[17]: array([[1, 0, 3],  
                  [5, 8, 3]])
```

A single number in A is located by four coordinates, so for example:

```
In [18]: A[0, 1, 0, 3]
```

```
Out[18]: 8
```

Tensors can be multiplied by a scalar, and their shape remains the same:

In [19]: A2 = 2 \* A  
A2

Out[19]: array([[[[ 8, 8, 16, 8, 18],  
[ 2, 16, 12, 4, 18],  
[12, 12, 6, 2, 14],  
[ 8, 14, 0, 10, 2]],  
  
[[ 4, 14, 6, 16, 4],  
[14, 8, 14, 14, 4],  
[ 0, 8, 2, 10, 12],  
[ 6, 6, 14, 18, 8]],  
  
[[ 0, 0, 16, 14, 14],  
[14, 4, 4, 10, 12],  
[ 2, 2, 12, 10, 10],  
[ 6, 4, 4, 6, 18]]],  
  
[[[ 2, 2, 8, 8, 4],  
[12, 12, 2, 0, 0],  
[ 4, 0, 2, 16, 14],  
[ 8, 2, 14, 4, 10]],  
  
[[12, 8, 14, 12, 18],  
[ 4, 8, 4, 14, 4],  
[10, 6, 4, 12, 8],  
[10, 4, 16, 6, 18]],  
  
[[ 6, 12, 18, 10, 0],  
[ 8, 16, 14, 16, 8],  
[18, 14, 18, 14, 12],  
[ 4, 18, 2, 8, 18]]]))

In [20]: A.shape == A2.shape

Out[20]: True

We can also add tensors of the same shape element by element to obtain a third tensor with the same shape:

In [21]: A + A2

Out[21]: array([[[[12, 12, 24, 12, 27],  
[ 3, 24, 18, 6, 27],

```

[18, 18, 9, 3, 21],
[12, 21, 0, 15, 3]],

[[ 6, 21, 9, 24, 6],
[21, 12, 21, 21, 6],
[ 0, 12, 3, 15, 18],
[ 9, 9, 21, 27, 12]],

[[ 0, 0, 24, 21, 21],
[21, 6, 6, 15, 18],
[ 3, 3, 18, 15, 15],
[ 9, 6, 6, 9, 27]]],

[[[ 3, 3, 12, 12, 6],
[18, 18, 3, 0, 0],
[ 6, 0, 3, 24, 21],
[12, 3, 21, 6, 15]],

[[18, 12, 21, 18, 27],
[ 6, 12, 6, 21, 6],
[15, 9, 6, 18, 12],
[15, 6, 24, 9, 27]],

[[ 9, 18, 27, 15, 0],
[12, 24, 21, 24, 12],
[27, 21, 27, 21, 18],
[ 6, 27, 3, 12, 27]]])

```

## Tensor Dot Product

One of the most important operation between tensors is the *product*. If we think about the product between two scalars, we have no doubts how to perform it. If we think about two vectors  $a = \{a_i\}$  and  $b = \{b_i\}$ , we can perform different types of product (for example the dot product or the cross product).

Here we focus on the so called **Dot Product**. The dot product  $p$  between  $a$  and  $b$  is given by:

$$p = \sum_i a_i b_i$$

The operation consists in summing up the product between the components of the two vectors. As you may observe, the results of the dot product between two vectors is a scalar, which is an entity with a lower order if compared with the two factors. For this reason, this operation is also called **contraction**.

A similar operation can be performed also between two tensors of higher order, if the two tensors have an axis with the same length. In this case we can perform a dot product (or a contraction) along that axis. The

shape of the resulting tensor depends on the shapes of the original two tensors that got contracted.

Let's see a couple of examples. Here are the shapes of A and B. A has order 4, B has order 2:

In [22]: `A.shape`

Out[22]: (2, 3, 4, 5)

In [23]: `B.shape`

Out[23]: (2, 3)

Since both A and B have a first axis of length 2, we can perform a tensor dot product along the first axis using the `tensordot` function from numpy. In order to perform this product we have to specify not only the two arguments A and B, but also that we want to perform the operation along the first axis in each of the 2 tensors. This can be done through the argument `axes=([0], [0])`, as explained in the `np.tensordot` documentation.

In [24]: `T = np.tensordot(A, B, axes=([0], [0]))`

Let's check the shape of T:

In [25]: `T.shape`

Out[25]: (3, 4, 5, 3)

Interesting! Can you see what happened? T has four axes, i.e. it has order 4,  $T = \{t_{jklm}\}$ . We can calculate that by thinking how many free indices remained in A and B after the contraction on axis 0. The elements of A are indicated by four indices  $A = \{a_{ijkl}\}$ , the elements of B are indicated by two indices  $B = \{b_{mn}\}$ . Mathematically, the tensor product performs the operation:

$$T = \{t_{jklm}\} = \sum_i a_{ijkl} b_{in}$$

so, the elements of the resulting tensor T are located by 4 indices: 3 coming from the tensor A and 1 coming from the tensor B.

Let's do another example. What will be the shape of the tensor product of A and B if we contract along the first 2 axes? First of all we have to check that the first two axes have the same length. Then we have to change the argument into `axes=([0, 1], [0, 1])`.

```
In [26]: T = np.tensordot(A, B, axes=([0, 1], [0, 1]))
T
```

```
Out[26]: array([[ 66,  59, 135, 108, 112],
   [ 80, 100,  54,  97,  55],
   [ 86,  54,  69, 125, 107],
   [ 79,  61, 108,  60, 152]])
```

```
In [27]: T.shape
```

```
Out[27]: (4, 5)
```

Since both axis 0 and axis 1 have been contracted, the only remaining 2 indices come from axis 2 and 3 of tensor A. This yields a tensor of order 2.

Wonderful! We have learned to perform a few operations using tensors! While this may seem really abstract and removed from the practical applications of Deep Learning, actually it is not. We need to understand how to arrange our data using tensors if we want to leverage Neural Networks with their full potential.

Now that we know how to operate with tensors, it is time to dive into convolutions!

We will start from 1D convolutions and then extend the definition to 2D arrays.

## Convolutions

In chapter 6 we introduced Convolutional Neural Networks and we went a bit fast when talking about convolutions. Let's introduce [convolutions](#) here in a bit more detail.

### 1D Convolution & Correlation

Let's start with two arrays, a and v.

```
In [28]: a_ = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
a = np.array(a_, dtype='float32')
a
```

```
Out[28]: array([0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
dtype=float32)
```

```
In [29]: v = np.array([-1, 1], dtype='float32')
v
```

```
Out[29]: array([-1.,  1.], dtype=float32)
```

v is a short array of only two elements, while a is a longer array of several elements. The general question we are looking to answer is how similar the two arrays are. Since they are not the same length we cannot perform a dot product between the two. We can, however, define two operations involving a and v: the **correlation** and the **convolution**. These operations try to gauge the similarity between the two arrays, acknowledging the fact that they don't have the same length and performing a sort of "rolling dot product".

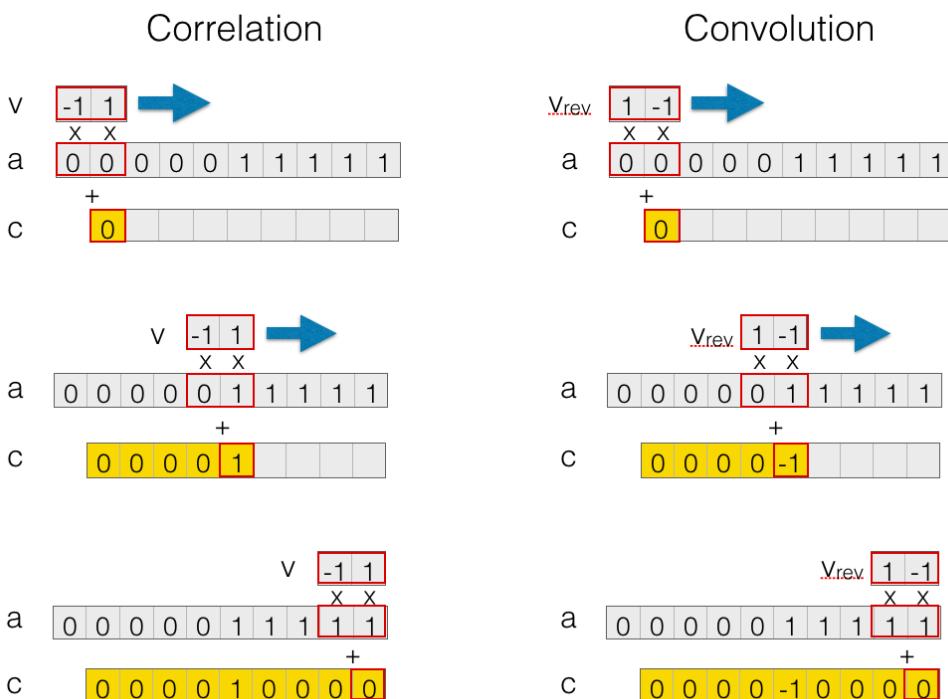
In both cases we start from the left-side of a and we take a short sub-array with the same length as v, in this case 2 numbers. In Machine Learning this sub-array is called **receptive field**.

Then we perform a tensor dot product of v with the receptive field a, i.e. we multiply the elements of v by the elements of the sub-array and we sum the products. We then store the result as the first element of our result array c.

Then we shift the window in a by one number and again perform a product between the new sub-array and v, also summing at the end. This second value gets stored in the result array as well.

We can continue shifting the window and performing dot products until we reach the end of the array a and no more shifting is possible.

The difference between convolution and correlation is that the array v is flipped before the multiplication.



For a more precise mathematical definition of the `correlation` and `convolution` of `a` with `v` we invite the user to consult the many detailed resources that can be found online.

Now, let's see how we can perform these operations with Numpy. The functions `np.correlate` and `np.convolve` offer these two functions to perform correlation and convolution of 1D arrays:

```
In [30]: cc = np.correlate(a, v, mode='valid')
cc
```

```
Out[30]: array([ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0., -1.,  0.,  0.,  0.,
 0.], dtype=float32)
```

```
In [31]: c = np.convolve(a, v, mode='valid')
c
```

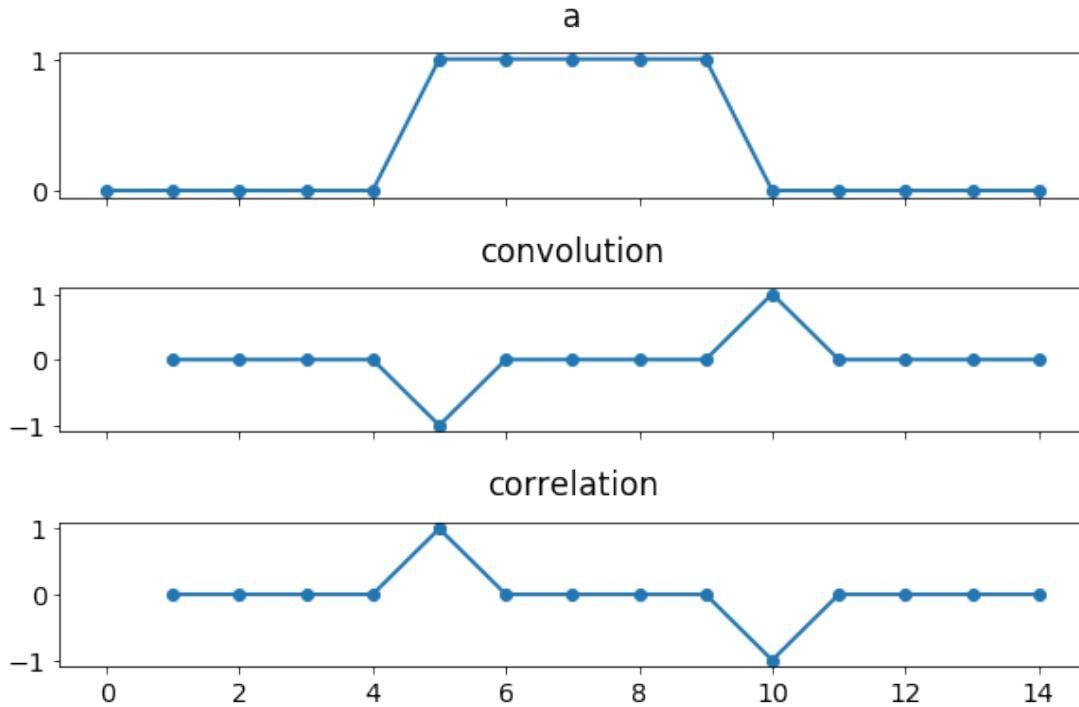
```
Out[31]: array([ 0.,  0.,  0.,  0., -1.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,
 0.], dtype=float32)
```

Let's plot `a` and `c` and see what they look like:

```
In [32]: fig, ax = plt.subplots(3, 1, sharex=True)
ax[0].plot(range(0, len(a)), a, 'o-')
ax[0].set_title('a')

ax[1].plot(range(1, len(a)), c, 'o-')
ax[1].set_title('convolution')

ax[2].plot(range(1, len(a)), cc, 'o-')
ax[2].set_title('correlation')
plt.tight_layout()
```



Looking at the plot we notice that both convolution and correlation have spikes when there's a jump in the array  $a$ . Our short filter  $v$  looks for jumps in  $a$  and the resulting convolution array represents how similar each window in  $a$  is with  $v$ .

TIP: Since in a convolutional Neural Network the filter  $v$  is learned during the training process, it makes no difference whether we flip the filter or not. In fact, if we perform the flip, the network will simply learn flipped weights. For this reason, convolutional layers in a Neural Network are actually calculating correlations. We still call them **convolutional layers**, but the operation performed is actually a **correlation**. In what follows we will keep talking about convolutions, but we'll keep in mind that flipping the array is not actually necessary in practice.

## 2D Convolution

We can easily extend the 1D convolution to 2D convolutions using 2D arrays instead of 1D.

Let's say we have a  $11 \times 11$  array  $A$  and a  $3 \times 3$  filter  $V$ .

$A$  contains a pattern in the shape of an "X". For the sake of simplicity, we will also rescale the values of the array so that the minimum value is  $-1$  and the maximum is  $+1$ , but same concepts apply for different range of

values.

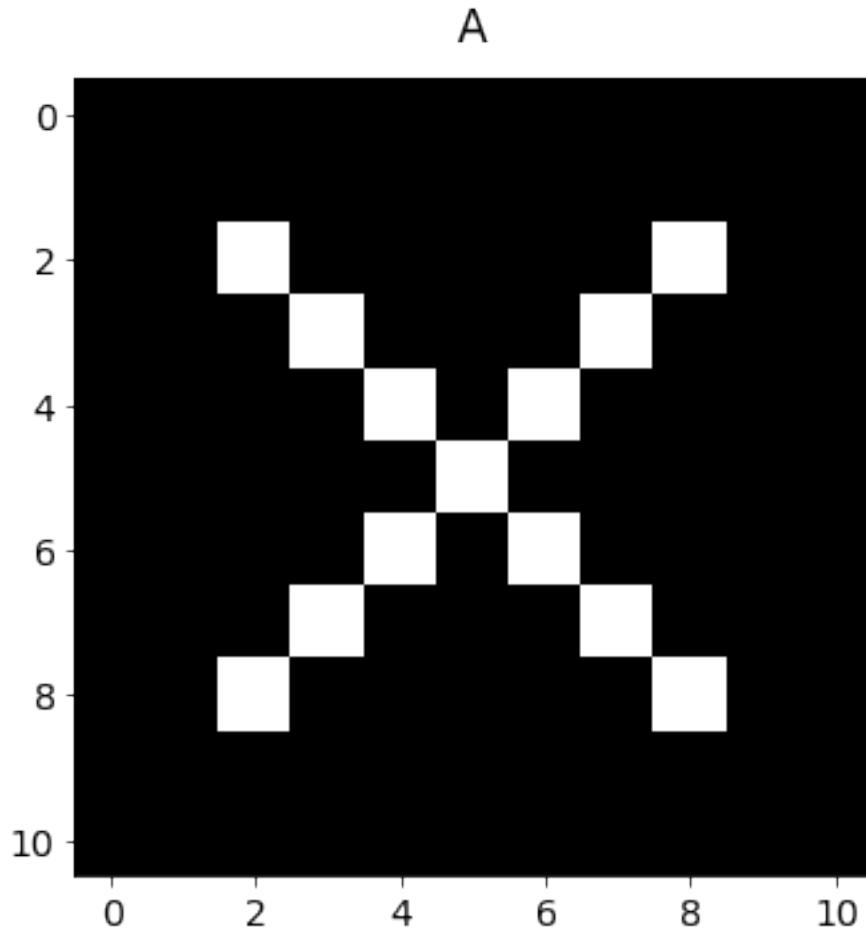
```
In [33]: A = np.zeros(shape=(11, 11))
A[2:-2, 2:-2] = np.diag(np.ones((7,))) + \
                  np.flip(np.diag(np.ones((7,))), 0)
A[5, 5] = 1
A = A * 2 - 1
A
```

```
Out[33]: array([[-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.],
                 [-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.],
                 [-1., -1., 1., -1., -1., -1., -1., 1., -1., -1.],
                 [-1., -1., -1., 1., -1., -1., -1., 1., -1., -1.],
                 [-1., -1., -1., -1., 1., -1., 1., -1., -1., -1.],
                 [-1., -1., -1., -1., -1., 1., -1., -1., -1., -1.],
                 [-1., -1., -1., -1., -1., -1., 1., -1., -1., -1.],
                 [-1., -1., -1., -1., -1., -1., -1., 1., -1., -1.],
                 [-1., -1., -1., -1., -1., -1., -1., -1., 1., -1.],
                 [-1., -1., -1., -1., -1., -1., -1., -1., -1., 1.]])
```

Let's display A with a gray colormap (because we're working with numbers between -1 and 1):

```
In [34]: plt.figure(figsize=(5.5, 5.5))
plt.imshow(A, cmap='gray')
plt.title('A')
```

```
Out[34]: Text(0.5, 1, 'A')
```



V is a 3x3 filter with a diagonal line:

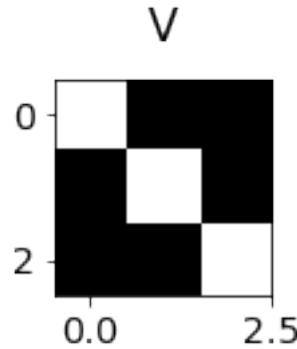
```
In [35]: V = np.diag([1, 1, 1])
V = V * 2 - 1
V
```

```
Out[35]: array([[ 1, -1, -1],
 [-1,  1, -1],
 [-1, -1,  1]])
```

Let's plot the v filter as well:

```
In [36]: plt.figure(figsize=(1.5, 1.5))
plt.imshow(V, cmap = 'gray')
plt.title('V')
```

```
Out[36]: Text(0.5,1,'V')
```



The 2D convolution can be calculated with the `scipy.signal.convolve2d` function as from `scipy`. Let's import the function from `scipy` first:

```
In [37]: from scipy.signal import convolve2d, correlate2d
```

Let's run the `convolve2d` function over A using the V tensor:

```
In [38]: C = convolve2d(A, V, mode='valid')
C
```

```
Out[38]: array([[ 5.,  1.,  1.,  3.,  3.,  3.,  5.,  1.,  1.],
   [ 1.,  7., -1.,  1.,  3.,  5., -1.,  3.,  1.],
   [ 1., -1.,  9., -1.,  3., -1.,  1., -1.,  5.],
   [ 3.,  1., -1.,  9., -3.,  1., -1.,  5.,  3.],
   [ 3.,  3.,  3., -3.,  5., -3.,  3.,  3.,  3.],
   [ 3.,  5., -1.,  1., -3.,  9., -1.,  1.,  3.],
   [ 5., -1.,  1., -1.,  3., -1.,  9., -1.,  1.],
   [ 1.,  3., -1.,  5.,  3.,  1., -1.,  7.,  1.],
   [ 1.,  1.,  5.,  3.,  3.,  1.,  1.,  1.,  5.]])
```

The convolved array, is obtained by taking the filter V, flipping it on both axis and then multiplying it with a 3x3 patch in the image. Then we shift the patch to the right and repeat. We start at the first patch on the top left of the image A[0:3, 0:3], multiply this patch with V\_rev element by element, then sum all the values.

We are effectively contracting the 2D tensor V with the patch over both axis:

```
In [39]: V_rev = np.flip(np.flip(V, 1), 0)
```

In [40]: `np.tensordot(A[0:3, 0:3], V_rev)`

Out[40]: `array(5.)`

This produces the first pixel in the output convolution. We then shift to the right by one pixel in A and repeat the contraction operation:

In [41]: `np.tensordot(A[1:4, 0:3], V)`

Out[41]: `array(1.)`

We can continue doing this and accumulate the result in a new 2D array.

Functionally, we can do the same exact thing that `scipy.convolve` does manually, although in practice we never need to do this thanks to `scipy`:

```
In [42]: win_h = V_rev.shape[0]
         win_w = V_rev.shape[1]

         out_h = A.shape[0] - win_h + 1
         out_w = A.shape[1] - win_w + 1

         res = np.zeros((out_h, out_w))

         for i in range(out_h):
             for j in range(out_w):
                 patch_ij = A[i:i+win_h, j:j+win_w]
                 try:
                     res[i, j] = np.tensordot(patch_ij, V_rev)
                 except Exception as ex:
                     print(i, j)
                     print(patch_ij)
                     print(V)
                     raise ex

         np.allclose(res, C)
```

Out[42]: `True`

TIP: the function `np.allclose` returns `True` if two arrays are element-wise equal within a tolerance. See the [documentation](#) for details.

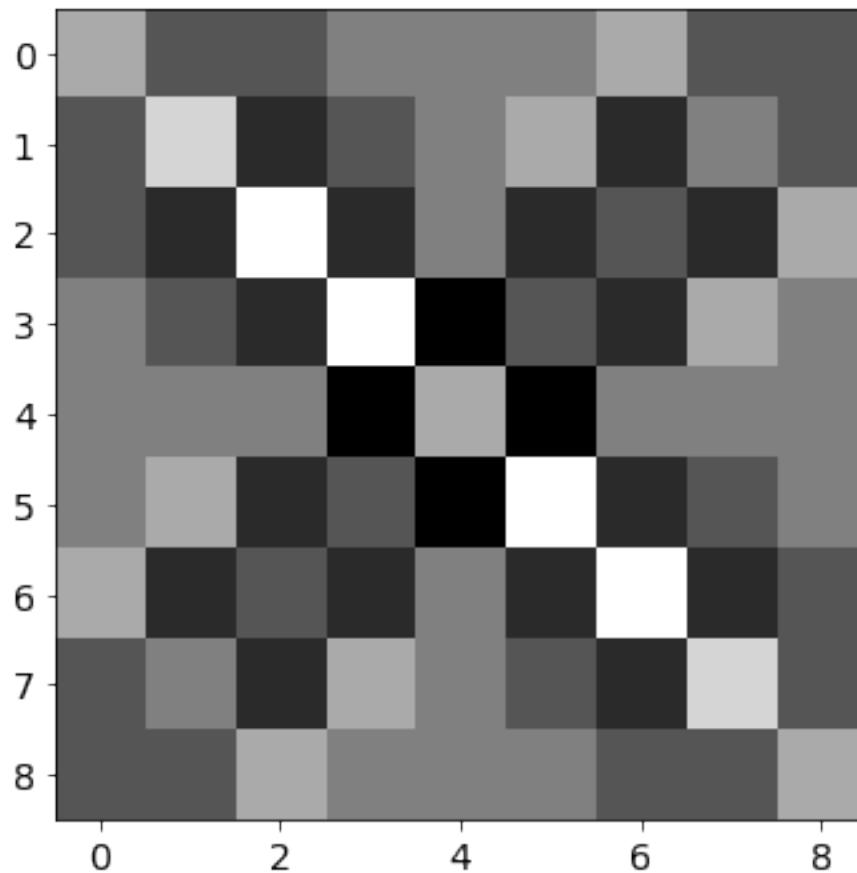
Notice that we can rescale the product by the number of elements in the filter  $V$ , which is 9, to obtain:

```
In [43]: C_resc = C / 9  
C_resc.round(2)
```

```
Out[43]: array([[ 0.56,  0.11,  0.11,  0.33,  0.33,  0.33,  0.56,  0.11,  0.11],  
                 [ 0.11,  0.78, -0.11,  0.11,  0.33,  0.56, -0.11,  0.33,  0.11],  
                 [ 0.11, -0.11,  1. , -0.11,  0.33, -0.11,  0.11, -0.11,  0.56],  
                 [ 0.33,  0.11, -0.11,  1. , -0.33,  0.11, -0.11,  0.56,  0.33],  
                 [ 0.33,  0.33,  0.33, -0.33,  0.56, -0.33,  0.33,  0.33,  0.33],  
                 [ 0.33,  0.56, -0.11,  0.11, -0.33,  1. , -0.11,  0.11,  0.33],  
                 [ 0.56, -0.11,  0.11, -0.11,  0.33, -0.11,  1. , -0.11,  0.11],  
                 [ 0.11,  0.33, -0.11,  0.56,  0.33,  0.11, -0.11,  0.78,  0.11],  
                 [ 0.11,  0.11,  0.56,  0.33,  0.33,  0.11,  0.11,  0.11,  0.56]])
```

```
In [44]: plt.imshow(C_resc, cmap='gray')
```

```
Out[44]: <matplotlib.image.AxesImage at 0x7fd808764780>
```



Four pixels in the resulting convolution are exactly equal to 1, corresponding to a perfect match of the filter with the image at those locations. The other pixels have smaller values with varying degrees, indicating partial match only.

## Image filters with convolutions

Convolutions can be used to perform filters on images, for example to blur it or detect the edges. Let's have a look at one example. We load an example image from `keras.datasets.mnist`:

```
In [45]: from keras.datasets import mnist
```

```
Using TensorFlow backend.
```

```
In [46]: (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

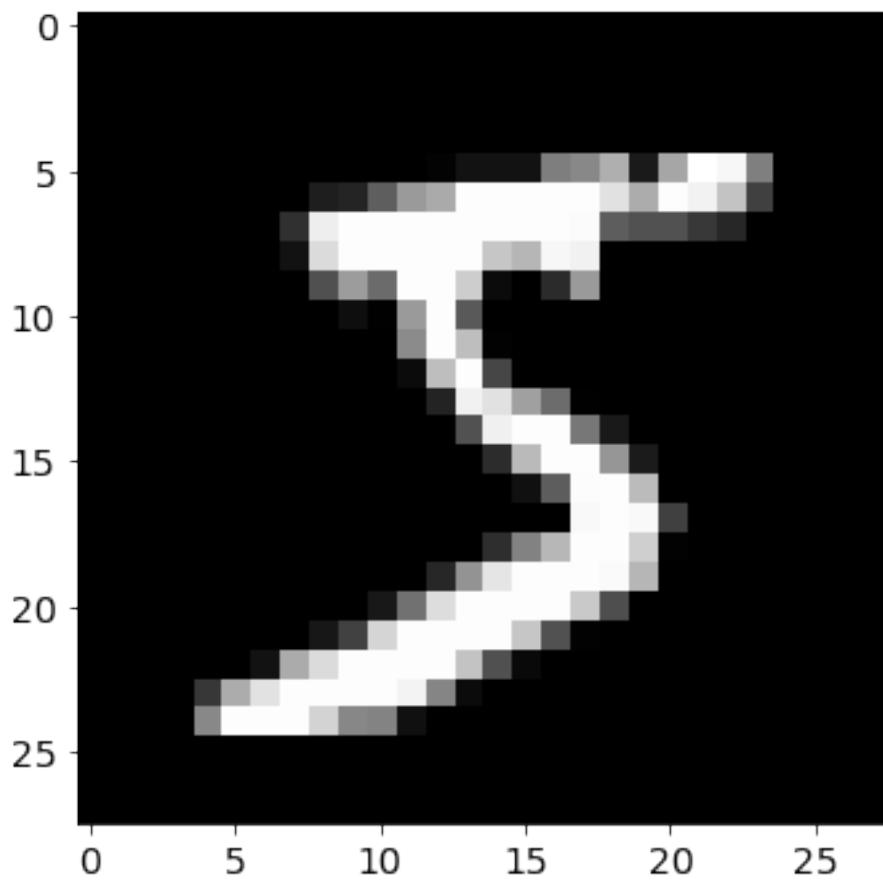
```
In [47]: img = x_train[0]
```

```
In [48]: img.shape
```

```
Out[48]: (28, 28)
```

```
In [49]: plt.imshow(img, cmap='gray')
```

```
Out[49]: <matplotlib.image.AxesImage at 0x7fd80c9d0be0>
```



Let's filter this image with 3x3 kernels that recognize lines:

```
In [50]: f1 = np.array([[ 1,  0,  0],
                      [ 0,  1,  0],
                      [ 0,  0,  1]])

f2 = np.array([[ 0,  0,  1],
              [ 0,  1,  0],
              [ 1,  0,  0]])

f3 = np.array([[[-1, -1, -1],
               [ 0,  0,  0],
               [ 1,  1,  1]]])

f4 = np.array([[[-1,  0,  1],
               [-1,  0,  1],
               [-1,  0,  1]])
```

Let's see what these kernels look like visually with the method `imshow()`:

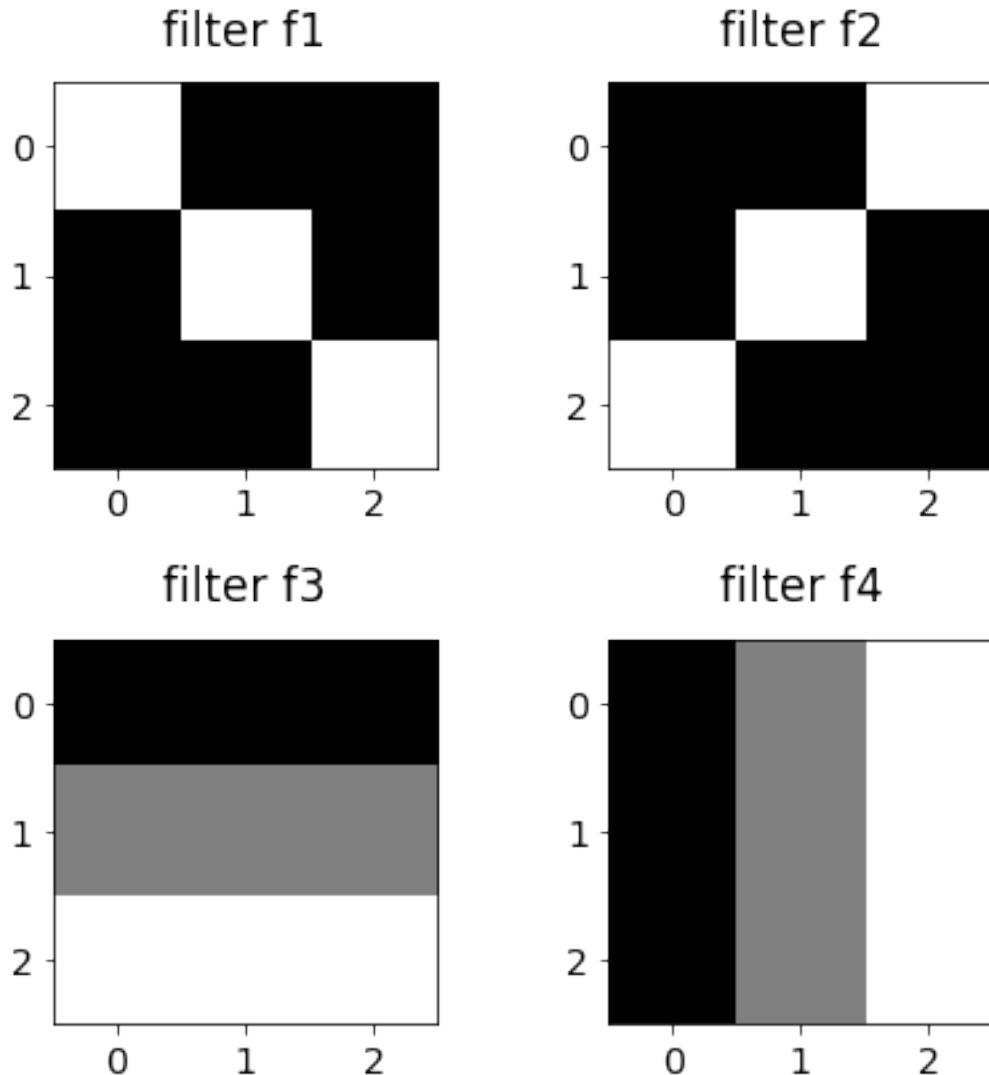
```
In [51]: plt.figure(figsize=(6, 6))
plt.subplot(221)
ax = plt.imshow(f1, cmap='gray')
plt.title('filter f1')

plt.subplot(222)
plt.imshow(f2, cmap='gray')
plt.title('filter f2')

plt.subplot(223)
plt.imshow(f3, cmap='gray')
plt.title('filter f3')

plt.subplot(224)
plt.imshow(f4, cmap='gray')
plt.title('filter f4')

plt.tight_layout()
plt.show()
```



Now let's run the 2D convolution on the image and see what these convolutions produce:

```
In [52]: plt.figure(figsize=(6, 6))

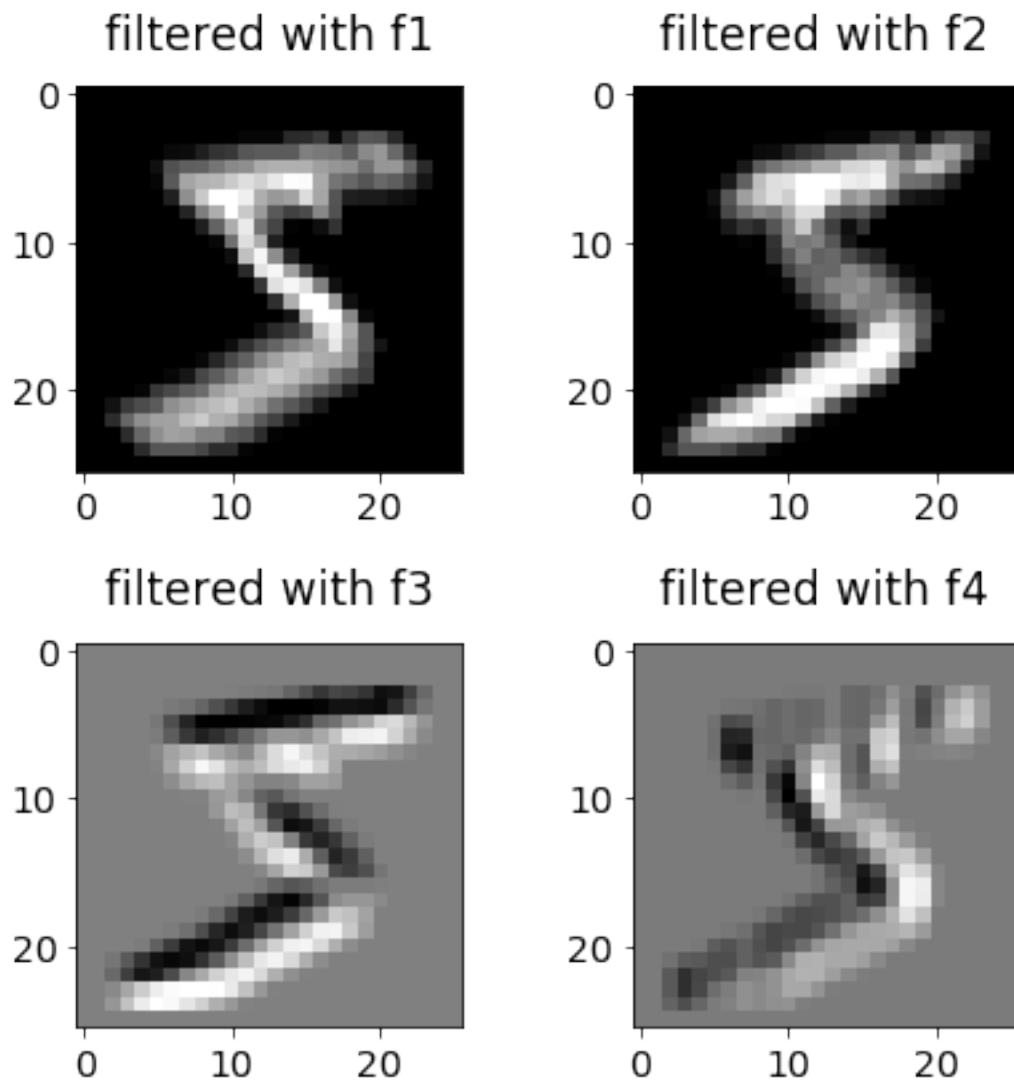
plt.subplot(221)
res = convolve2d(img, f1, mode='valid')
plt.imshow(res, cmap='gray')
plt.title('filtered with f1')

plt.subplot(222)
res = convolve2d(img, f2, mode='valid')
plt.imshow(res, cmap='gray')
plt.title('filtered with f2')
```

```
plt.subplot(223)
res = convolve2d(img, f3, mode='valid')
plt.imshow(res, cmap='gray')
plt.title('filtered with f3')

plt.subplot(224)
res = convolve2d(img, f4, mode='valid')
plt.imshow(res, cmap='gray')
plt.title('filtered with f4')

plt.tight_layout()
plt.show()
```



Great! We have seen how convolutions can be used to filter images. Each pixel in the filtered image is the result of a tensor contraction of the filter with a patch in the original image. In this respect, the convolution is the operation that allows us to leverage the fact that information is related to spatial patterns of nearby pixels.

TIP: If you've ever used an image program like Adobe Photoshop, these convolutions are how the image filters are created for images.

## Backpropagation for Recurrent Networks

The recurrent relations in the general case can be written as:

$$z_t = w h_{t-1} + u x_t \quad (14.24)$$

$$h_t = \phi(z_t) \quad (14.25)$$

$$r_t = v h_t \quad (14.26)$$

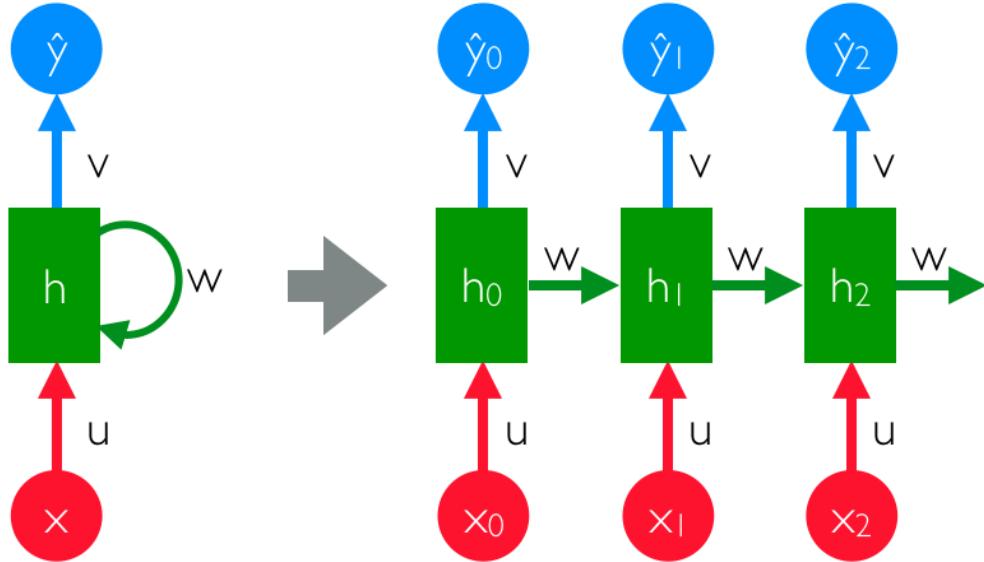
$$\hat{y}_t = \phi(r_t) \quad (14.27)$$

$$(14.28)$$

where we substituted the tanh activation function to a generic activation  $\phi$  and allowed for different weights on the recurrent relation and the output relation.

The graph of this more general looks like this:

The backpropagation relations can be written as:



recurrent\_2.png

$$\bar{\hat{y}} = \frac{\partial J}{\partial \hat{y}} \quad (14.29)$$

$$\bar{r}_t = \bar{\hat{y}} \phi'(r_t) \quad (14.30)$$

$$\bar{h}_t = \bar{r}_t v + \bar{z}_{t+1} w \quad (14.31)$$

$$\bar{z}_t = \bar{h}_t \phi'(z_t) \quad (14.32)$$

$$\bar{u} = \sum_{t=0}^T \bar{z}_t x_t \quad (14.33)$$

$$\bar{v} = \sum_{t=0}^T \bar{r}_t h_t \quad (14.34)$$

$$\bar{w} = \sum_{t=0}^T \bar{z}_{t+1} h_t \quad (14.35)$$

$$(14.36)$$

As you can see, these relations are very similar to the fully connected backpropagation relations we saw in [Chapter 5](#), with a big difference: the updates to the weights require a summation on the contributions from all time.

# 15

## Getting Started Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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`

```
In [3]: a = np.zeros((10, 10))
```

```
In [4]: a[::2, ::2] = 1  
      a[1::2, 1::2] = 1
```

```
In [5]: a
```

```
Out[5]: array([[1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],  
               [0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],  
               [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],  
               [0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],  
               [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],  
               [0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],  
               [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],  
               [0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],  
               [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],  
               [0., 1., 0., 1., 0., 1., 0., 1., 0., 1.]])
```

```
In [6]: b = np.arange(5, 15)
```

```
In [7]: b
```

```
Out[7]: array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [8]: # c = a * b[:, None]  
      c = a * b.reshape((10, 1))
```

```
In [9]: c
```

```
Out[9]: array([[ 5.,  0.,  5.,  0.,  5.,  0.,  5.,  0.,  5.,  0.],  
               [ 0.,  6.,  0.,  6.,  0.,  6.,  0.,  6.,  0.,  6.],  
               [ 7.,  0.,  7.,  0.,  7.,  0.,  7.,  0.,  7.,  0.],  
               [ 0.,  8.,  0.,  8.,  0.,  8.,  0.,  8.,  0.,  8.],  
               [ 9.,  0.,  9.,  0.,  9.,  0.,  9.,  0.,  9.,  0.],  
               [ 0., 10.,  0., 10.,  0., 10.,  0., 10.,  0., 10.],  
               [11.,  0., 11.,  0., 11.,  0., 11.,  0., 11.,  0.],  
               [ 0., 12.,  0., 12.,  0., 12.,  0., 12.,  0., 12.],  
               [13.,  0., 13.,  0., 13.,  0., 13.,  0., 13.,  0.],  
               [ 0., 14.,  0., 14.,  0., 14.,  0., 14.,  0., 14.]])
```

```
In [10]: c.mean(axis=0)
```

```
Out[10]: array([4.5, 5. , 4.5, 5. , 4.5, 5. , 4.5, 5. , 4.5, 5. ])
```

```
In [11]: c.mean(axis=1)
```

```
Out[11]: array([2.5, 3., 3.5, 4., 4.5, 5., 5.5, 6., 6.5, 7.])
```

```
In [12]: c.std(axis=0)
```

```
Out[12]: array([4.9244289, 5.38516481, 4.9244289, 5.38516481, 4.9244289,
 5.38516481, 4.9244289, 5.38516481, 4.9244289, 5.38516481])
```

```
In [13]: c.std(axis=1)
```

```
Out[13]: array([2.5, 3., 3.5, 4., 4.5, 5., 5.5, 6., 6.5, 7.])
```

```
In [14]: d = c[c>0].reshape(10, 5)
```

```
In [15]: d
```

```
Out[15]: array([[ 5.,  5.,  5.,  5.,  5.],
 [ 6.,  6.,  6.,  6.,  6.],
 [ 7.,  7.,  7.,  7.,  7.],
 [ 8.,  8.,  8.,  8.,  8.],
 [ 9.,  9.,  9.,  9.,  9.],
 [10., 10., 10., 10., 10.],
 [11., 11., 11., 11., 11.],
 [12., 12., 12., 12., 12.],
 [13., 13., 13., 13., 13.],
 [14., 14., 14., 14., 14.]])
```

```
In [16]: noise = np.random.normal(scale=0.1, size=(10, 5))
```

```
In [17]: e = d + noise
```

## Exercise 2

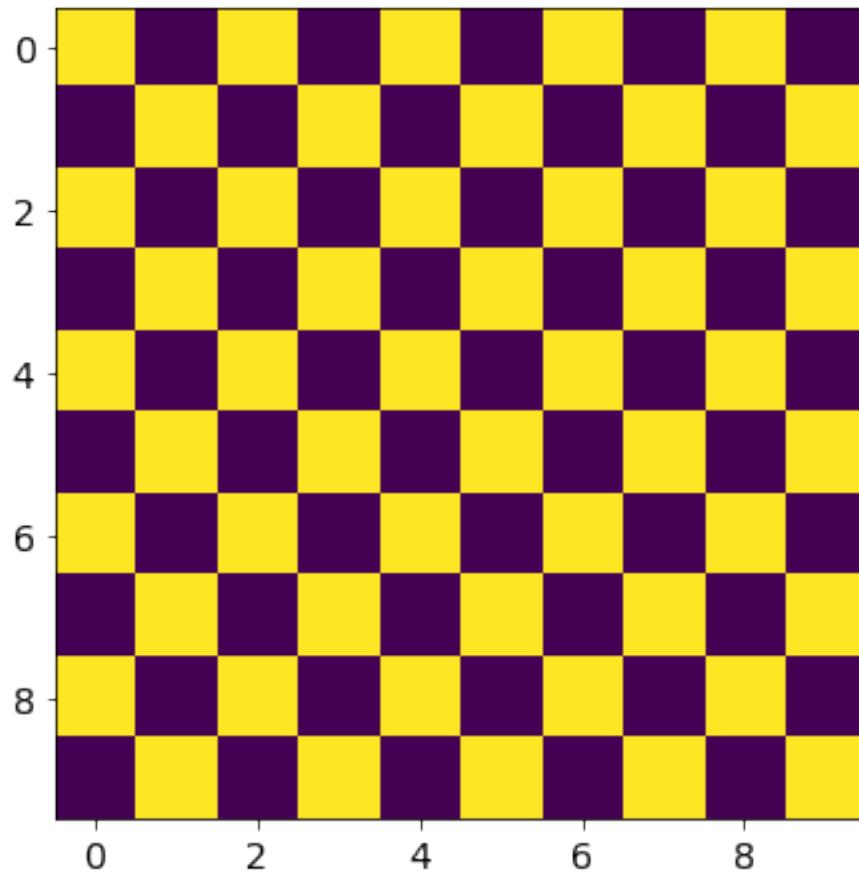
Practice plotting with `matplotlib`

- 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

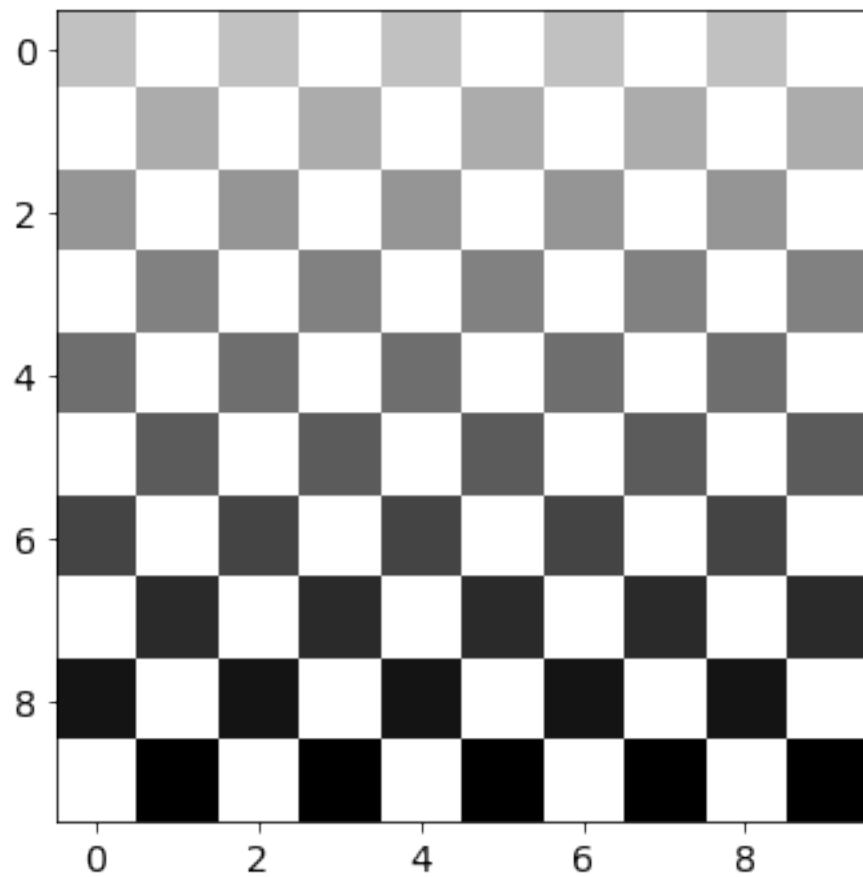
In [18]: `plt.imshow(a)`

Out[18]: <matplotlib.image.AxesImage at 0x7f11d2280f98>



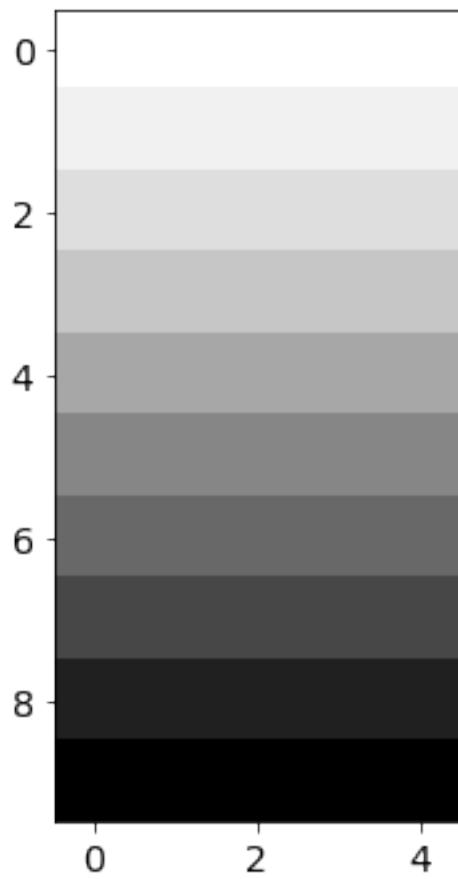
In [19]: `plt.imshow(c, cmap='Greys')`

Out[19]: <matplotlib.image.AxesImage at 0x7f11d220ba58>



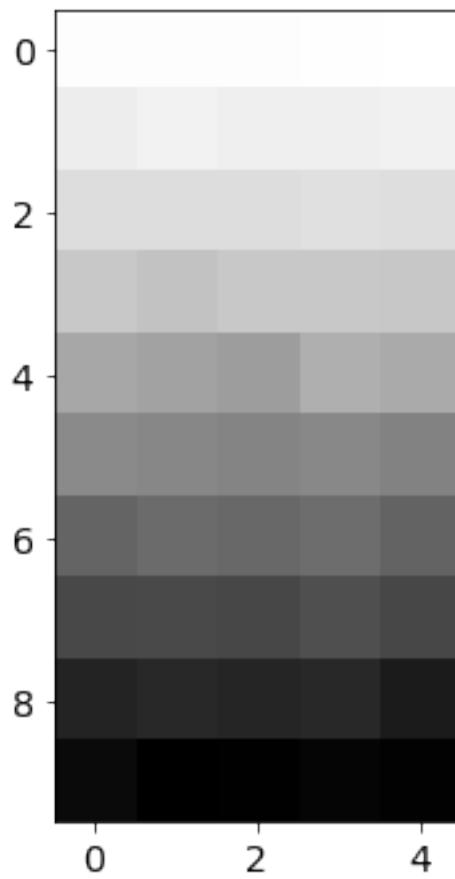
```
In [20]: plt.imshow(d, cmap='Greys')
```

```
Out[20]: <matplotlib.image.AxesImage at 0x7f11d21e8320>
```

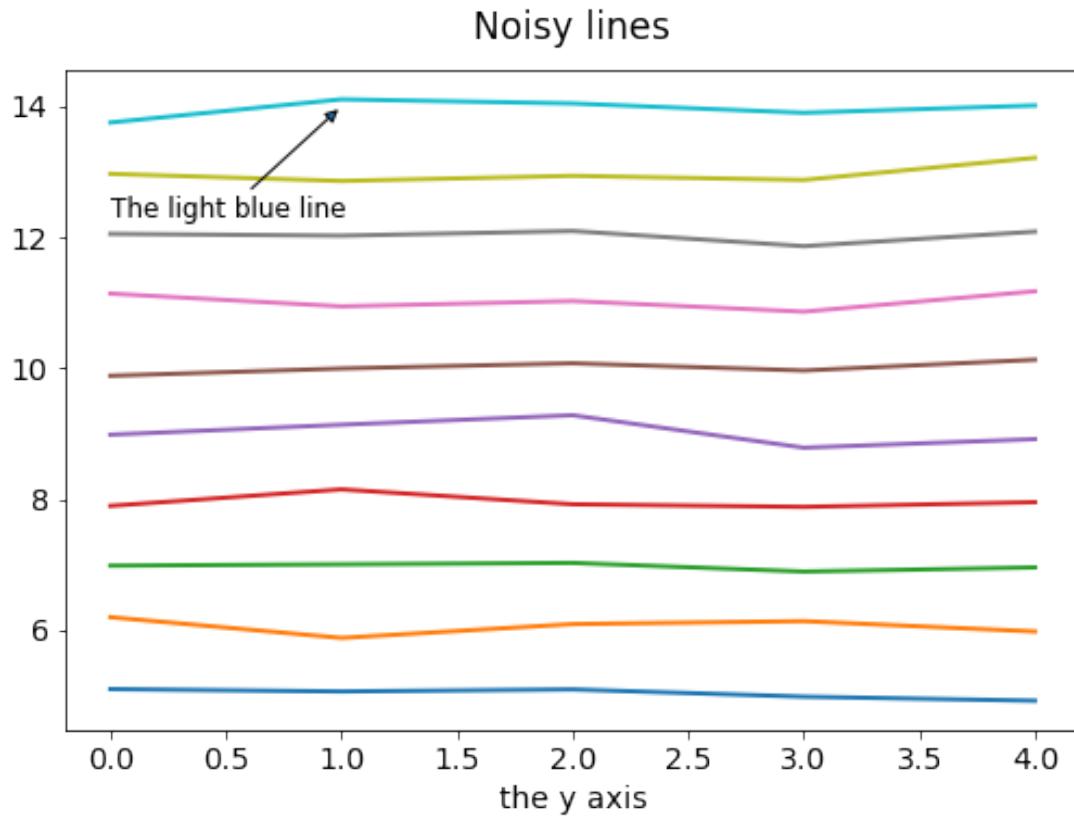


```
In [21]: plt.imshow(e, cmap='Greys')
```

```
Out[21]: <matplotlib.image.AxesImage at 0x7f11d2138080>
```



```
In [22]: plt.plot(e.transpose())
plt.title("Noisy lines")
plt.xlabel("the x axis")
plt.ylabel("the y axis")
plt.annotate(xy=(1, 14), xytext=(0, 12.3),
            s="The light blue line",
            arrowprops={"arrowstyle": '-|>'},
            fontsize=12);
```



### 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):
    ...
```

```
In [23]: def plot_decision_boundary(model, X, y):
    hticks = np.linspace(X.min()-0.1, X.max()+0.1, 101)
    vticks = np.linspace(X.min()-0.1, X.max()+0.1, 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=(7, 7))
```

```
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");
```

## 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

```
In [24]: from keras.models import Sequential
         from keras.layers import Dense
         from keras.optimizers import SGD
```

Using TensorFlow backend.

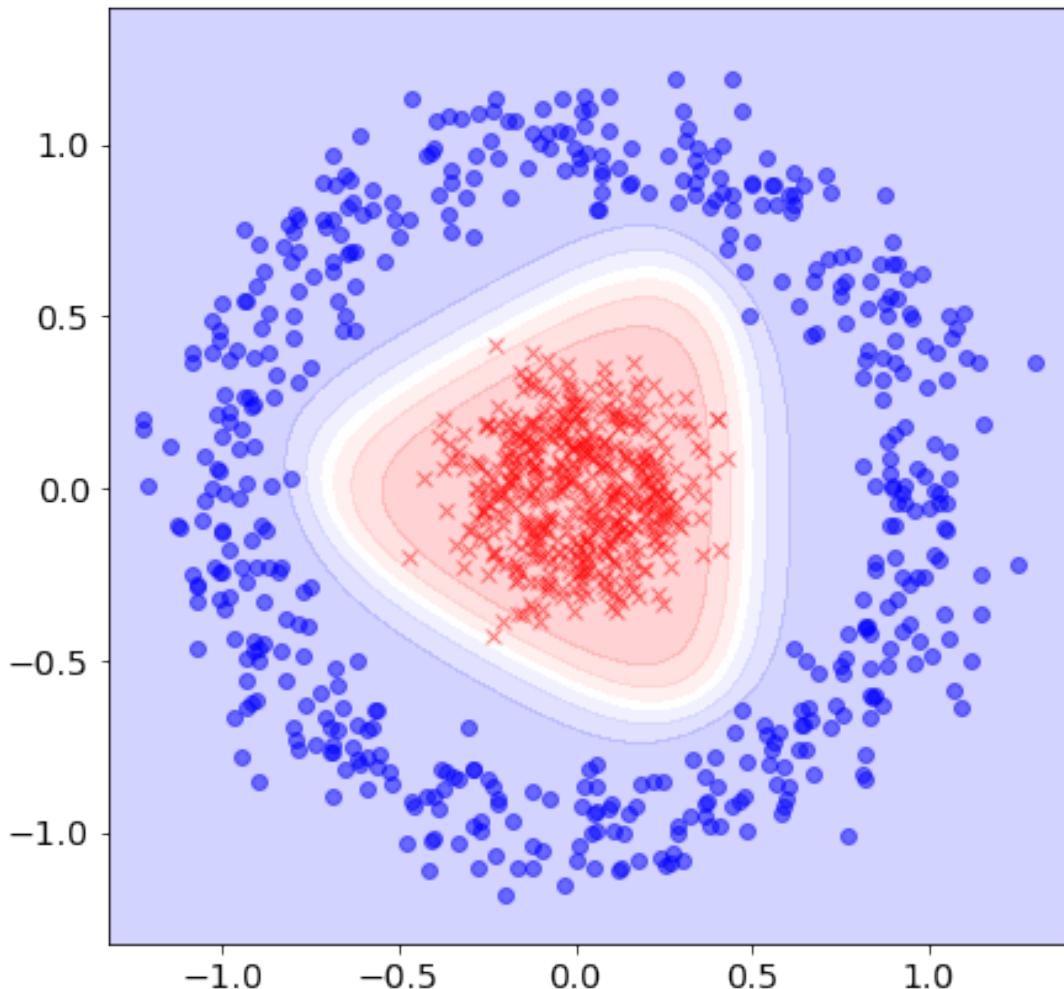
```
In [25]: from sklearn.datasets import make_circles

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

```
In [26]: model = Sequential()
         model.add(Dense(4, input_shape=(2,), activation='tanh'))
         model.add(Dense(1, activation='sigmoid'))
         model.compile(SGD(lr=0.5),
                       'binary_crossentropy',
                       metrics=['accuracy'])
         model.fit(X, y, epochs=30, verbose=0);
```

```
In [27]: plot_decision_boundary(model, X, y)
```

## Blue circles and Red crosses



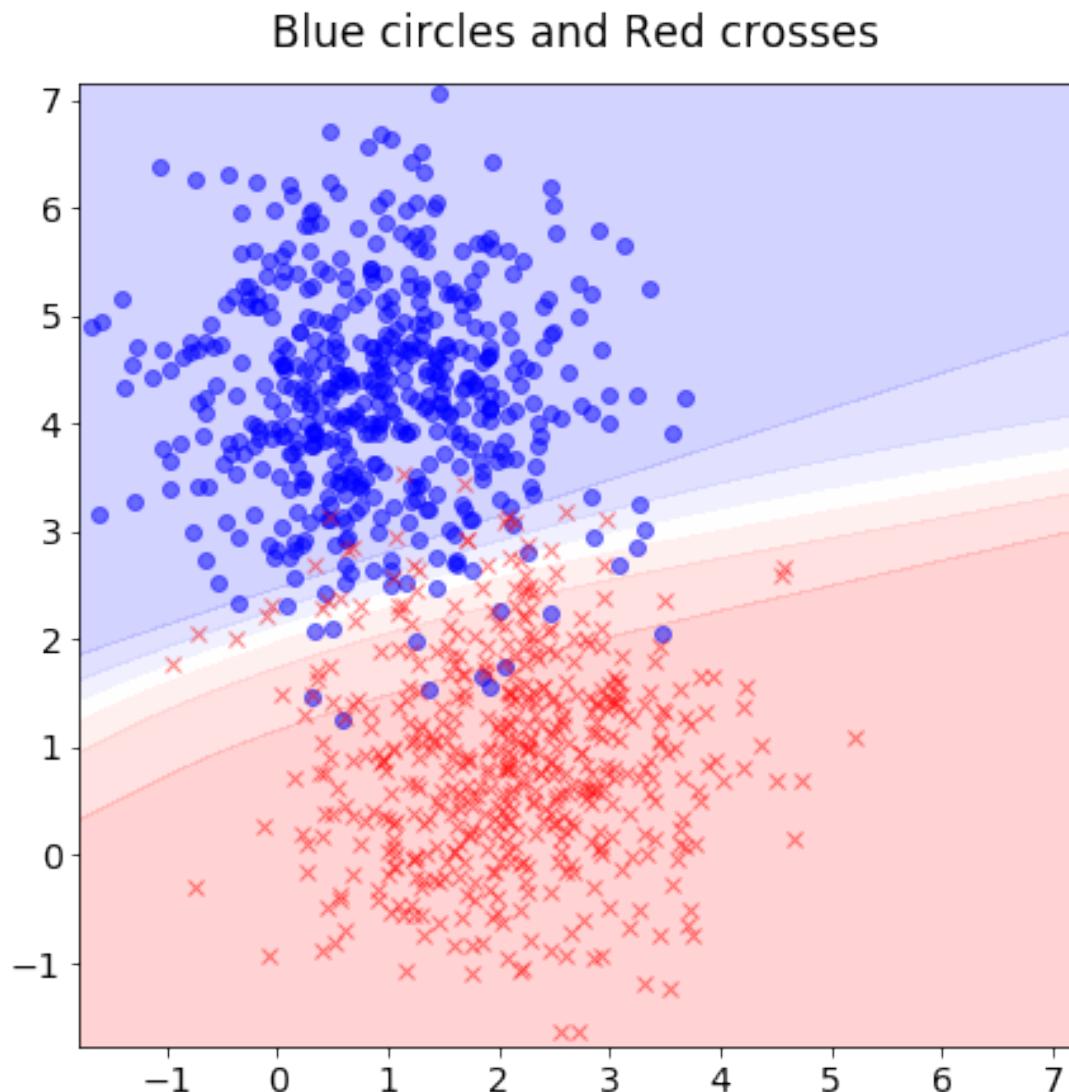
```
In [28]: from sklearn.datasets import make_blobs
```

```
X, y = make_blobs(n_samples=1000,  
                   centers=2,  
                   random_state=0)
```

```
In [29]: model = Sequential()  
model.add(Dense(4, input_shape=(2,), activation='tanh'))  
model.add(Dense(1, activation='sigmoid'))  
model.compile(SGD(lr=0.5),  
              'binary_crossentropy',
```

```
metrics=['accuracy'])  
model.fit(X, y, epochs=30, verbose=0);
```

In [30]: `plot_decision_boundary(model, X, y)`



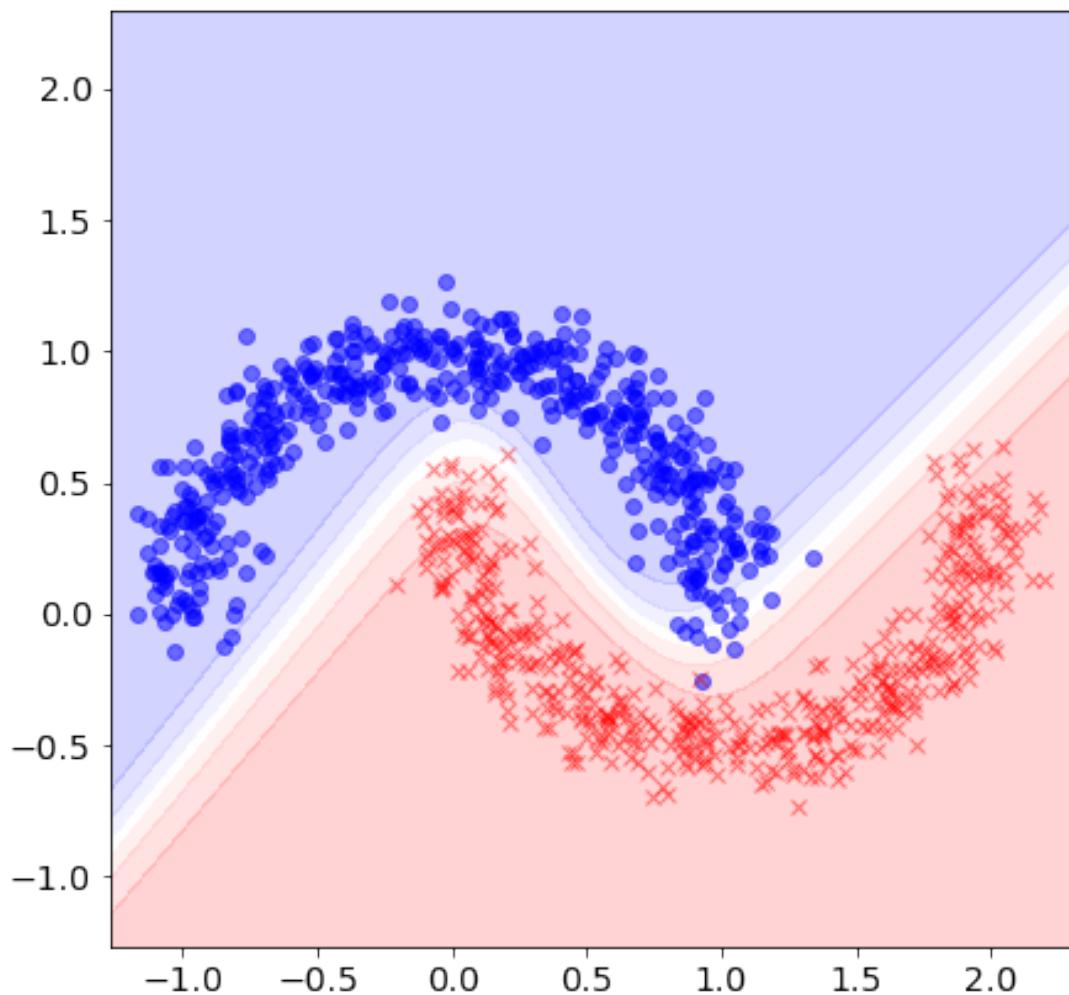
In [31]: `from sklearn.datasets import make_moons`

```
X, y = make_moons(n_samples=1000,  
                   noise=0.1,  
                   random_state=0)
```

```
In [32]: model = Sequential()
model.add(Dense(4, input_shape=(2,), activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
model.compile(SGD(lr=0.5),
              'binary_crossentropy',
              metrics=['accuracy'])
model.fit(X, y, epochs=30, verbose=0);
```

```
In [33]: plot_decision_boundary(model, X, y)
```

Blue circles and Red crosses



# 16

## Data Manipulation Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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:  
[http://pandas.pydata.org/pandas-docs/version/0.20/generated/pandas.to\\_datetime.html](http://pandas.pydata.org/pandas-docs/version/0.20/generated/pandas.to_datetime.html))
- 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

```
In [3]: fname_ = '../data/international-airline-passengers.csv'  
df = pd.read_csv(fname_)
```

```
In [4]: # - inspect it using the .info() and .head() commands  
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
Month           144 non-null object
Thousand Passengers 144 non-null int64
dtypes: int64(1), object(1)
memory usage: 2.3+ KB
```

In [5]: df.head()

Out[5] :

	Month	Thousand Passengers
0	1949-01	112
1	1949-02	118
2	1949-03	132
3	1949-04	129
4	1949-05	121

```
In [6]: # - use the function to_datetime() to change the
#       column type of 'Month' to a datetime type
# - set the index of df to be a datetime index using
#   the column 'Month' and the set_index() method

df['Month'] = pd.to_datetime(df['Month'])
df = df.set_index('Month')
```

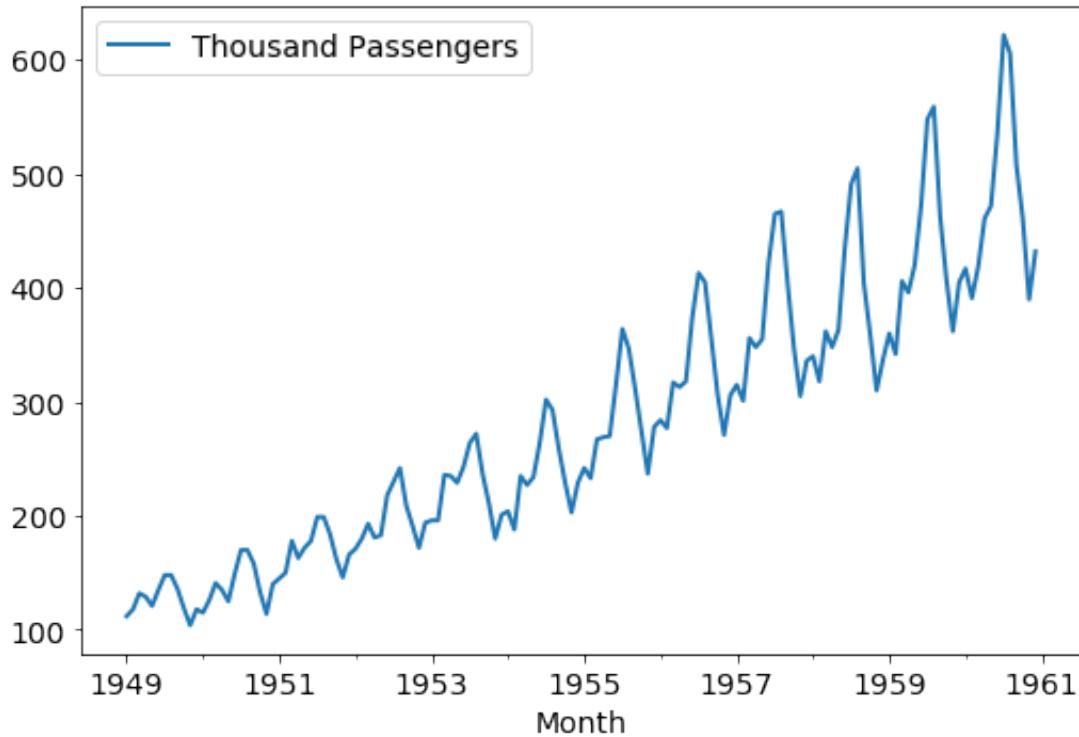
In [7]: df.head()

Out[7] :

Month	Thousand Passengers
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

```
In [8]: # - choose the appropriate plot and display the data
# - choose appropriate scale
# - label the axes
```

```
df.plot();
```



## 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 2 different colors on a new scatter plot
- remember to label the axes

```
In [9]: # - load the dataset: ../data/weight-height.csv
# - inspect it
df = pd.read_csv('../data/weight-height.csv')
df.head()
```

```
Out[9] :
```

	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

In [10]: df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 3 columns):
Gender    10000 non-null object
Height     10000 non-null float64
Weight     10000 non-null float64
dtypes: float64(2), object(1)
memory usage: 234.5+ KB
```

In [11]: df.describe()

Out[11]:

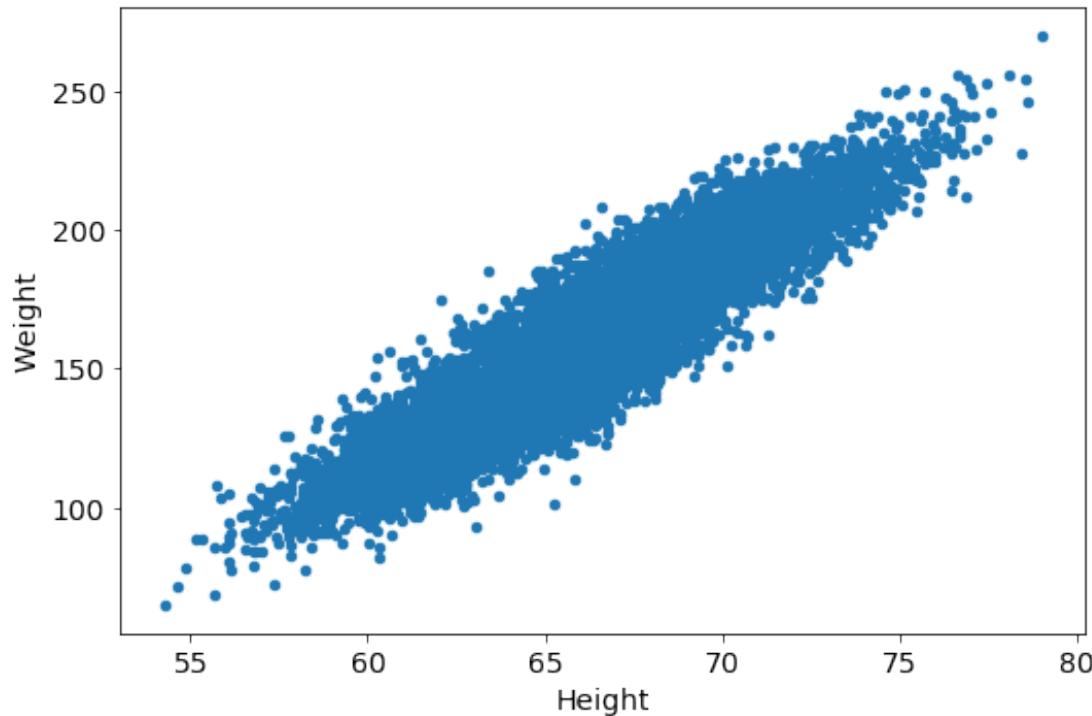
	Height	Weight
count	10000.000000	10000.000000
mean	66.367560	161.440357
std	3.847528	32.108439
min	54.263133	64.700127
25%	63.505620	135.818051
50%	66.318070	161.212928
75%	69.174262	187.169525
max	78.998742	269.989699

In [12]: df['Gender'].value\_counts()

Out[12]:

Gender
Male
Female

```
In [13]: # - plot it using a scatter plot with Weight as a
#       function of Height
df.plot(kind='scatter', x='Height', y='Weight');
```

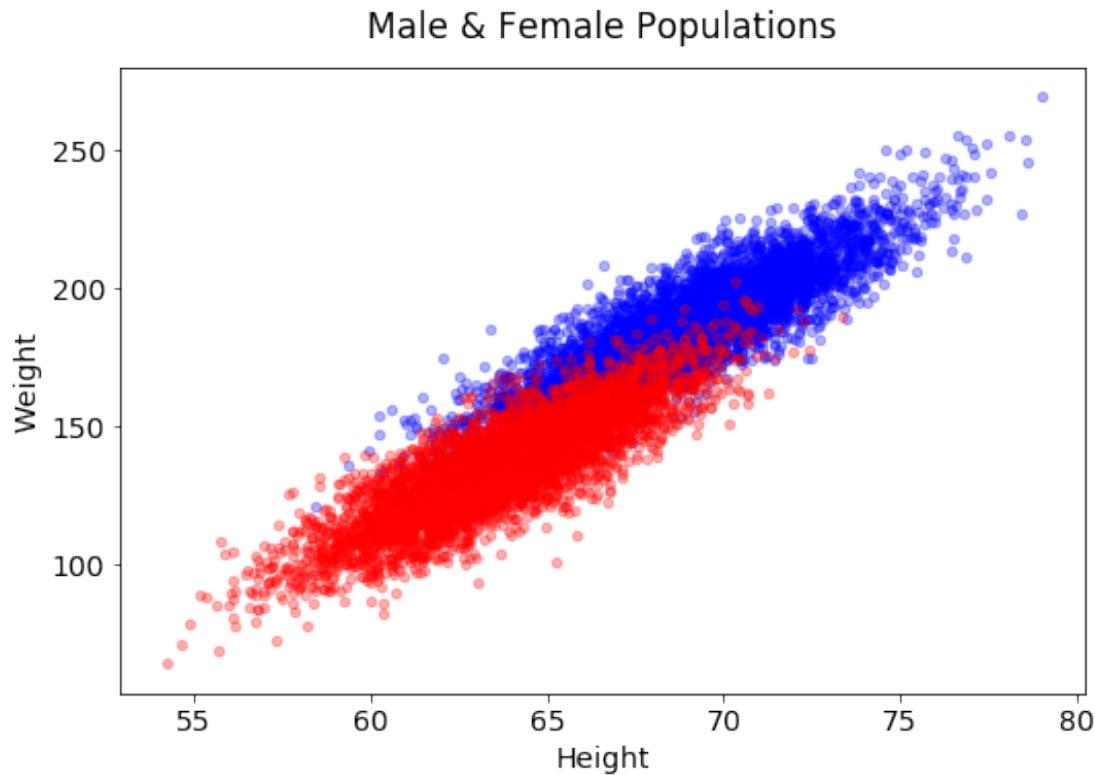


```
In [14]: # - plot the male and female populations with 2
#      different colors on a new scatter plot
# - remember to label the axes

# this can be done in several ways, showing 3 here:
# method 1
males = df[df['Gender'] == 'Male']
females = df.query('Gender == "Female"')
fig, ax = plt.subplots()

males.plot(kind='scatter', x='Height', y='Weight',
           ax=ax, color='blue', alpha=0.3,
           title='Male & Female Populations')

females.plot(kind='scatter', x='Height', y='Weight',
             ax=ax, color='red', alpha=0.3);
```



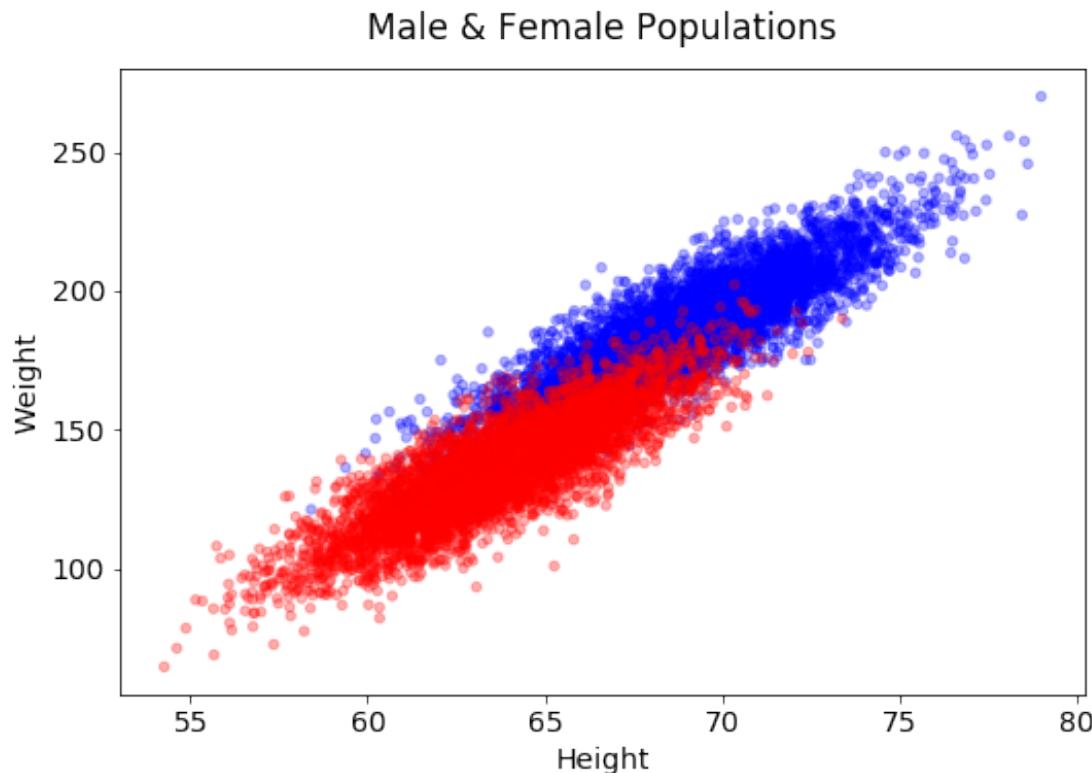
```
In [15]: # method 2
mfmap = {'Male': 'blue', 'Female': 'red'}
df['Gendercolor'] = df['Gender'].map(mfmap)
df.head()
```

Out[15] :

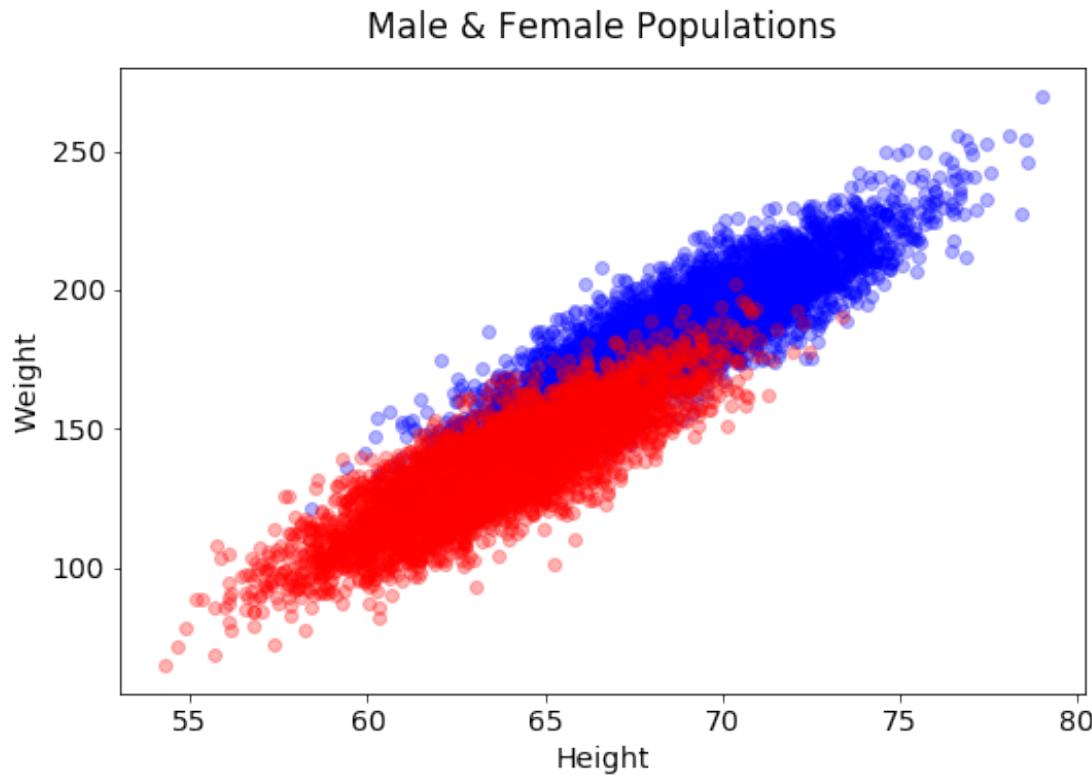
	Gender	Height	Weight	Gendercolor
0	Male	73.847017	241.893563	blue
1	Male	68.781904	162.310473	blue
2	Male	74.110105	212.740856	blue
3	Male	71.730978	220.042470	blue
4	Male	69.881796	206.349801	blue

```
In [16]: df.plot(kind='scatter',
                 x='Height',
                 y='Weight',
                 c=df['Gendercolor'],
```

```
alpha=0.3,  
title='Male & Female Populations');
```



```
In [17]: # method 3  
fig, ax = plt.subplots()  
ax.plot(males['Height'], males['Weight'], 'ob',  
        females['Height'], females['Weight'], 'or',  
        alpha=0.3)  
plt.xlabel('Height')  
plt.ylabel('Weight')  
plt.title('Male & Female Populations');
```



### 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

```
In [18]: males['Height'].plot(kind='hist',
                           bins=50,
                           range=(50, 80),
                           alpha=0.3,
                           color='blue')

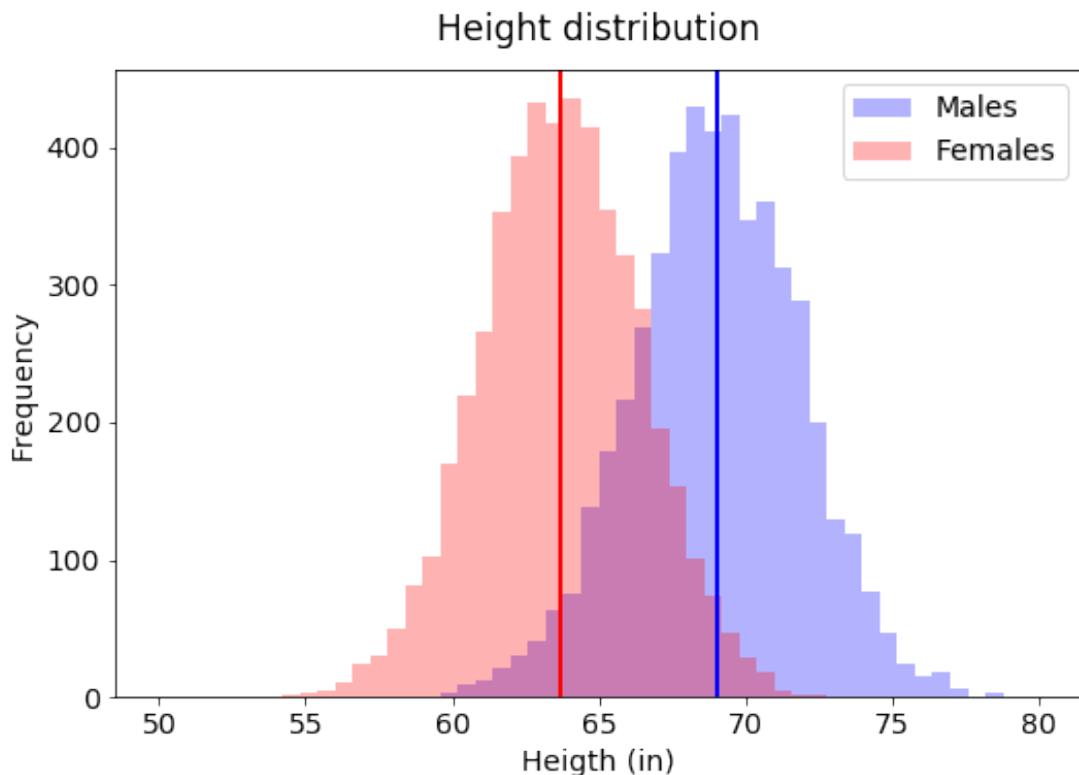
females['Height'].plot(kind='hist',
                       bins=50,
                       range=(50, 80),
                       alpha=0.3,
                       color='red')

plt.title('Height distribution')
```

```
plt.legend(["Males", "Females"])
plt.xlabel("Heighth (in)")

plt.axvline(males['Height'].mean(),
            color='blue', linewidth=2)

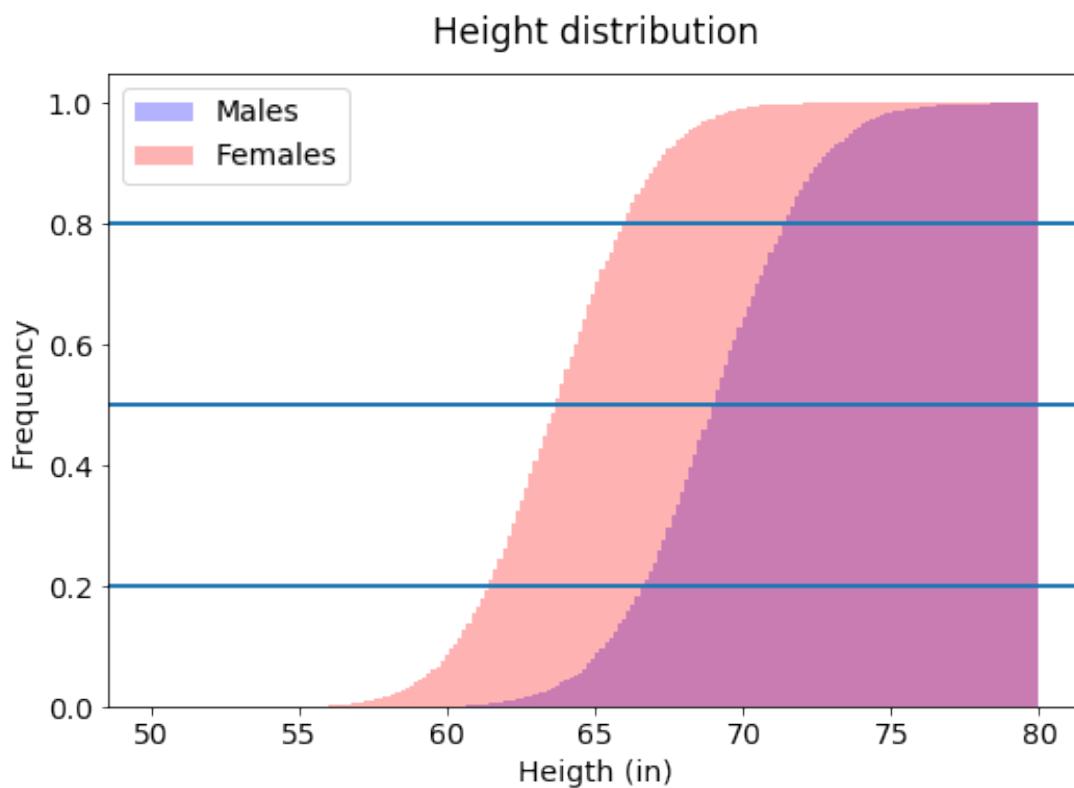
plt.axvline(females['Height'].mean(),
            color='red', linewidth=2);
```



```
In [19]: males['Height'].plot(kind='hist',
                           bins=200,
                           range=(50, 80),
                           alpha=0.3,
                           color='blue',
                           cumulative=True,
                           normed=True)

females['Height'].plot(kind='hist',
                       bins=200,
```

```
range=(50, 80),  
alpha=0.3,  
color='red',  
cumulative=True,  
normed=True)  
  
plt.title('Height distribution')  
plt.legend(["Males", "Females"])  
plt.xlabel("Heigth (in)")  
  
plt.axhline(0.8)  
plt.axhline(0.5)  
plt.axhline(0.2);  
  
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-  
packages/matplotlib/axes/_axes.py:6571: UserWarning: The 'normed' kwarg is  
deprecated, and has been replaced by the 'density' kwarg.  
warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



## 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)

```
In [20]: dfpvt = df.pivot(columns = 'Gender', values = 'Weight')
```

```
In [21]: dfpvt.head()
```

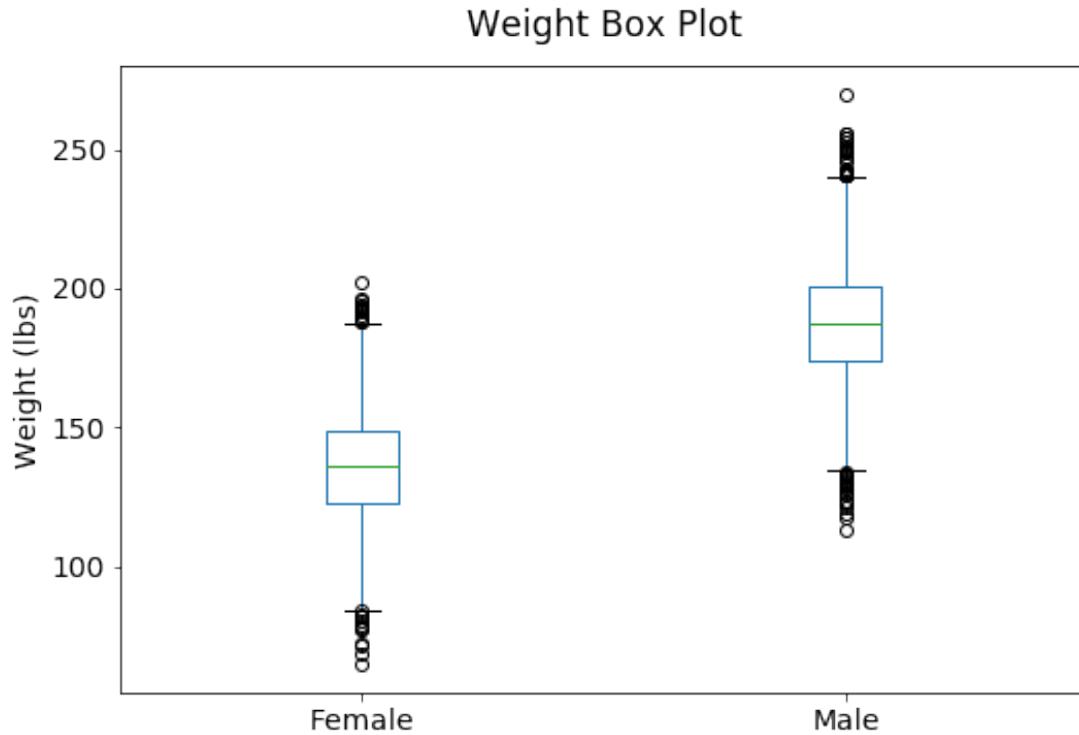
```
Out[21]:
```

Gender	Female	Male
0	NaN	241.893563
1	NaN	162.310473
2	NaN	212.740856
3	NaN	220.042470
4	NaN	206.349801

```
In [22]: dfpvt.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns (total 2 columns):
Female      5000 non-null float64
Male        5000 non-null float64
dtypes: float64(2)
memory usage: 234.4 KB
```

```
In [23]: dfpvt.plot(kind='box')
plt.title('Weight Box Plot')
plt.ylabel("Weight (lbs)");
```



## Exercise 5

- load the dataset: `../data/titanic-train.csv`
- learn about scattermatrix here: <http://pandas.pydata.org/pandas-docs/stable/visualization.html>
- display the data using a scattermatrix

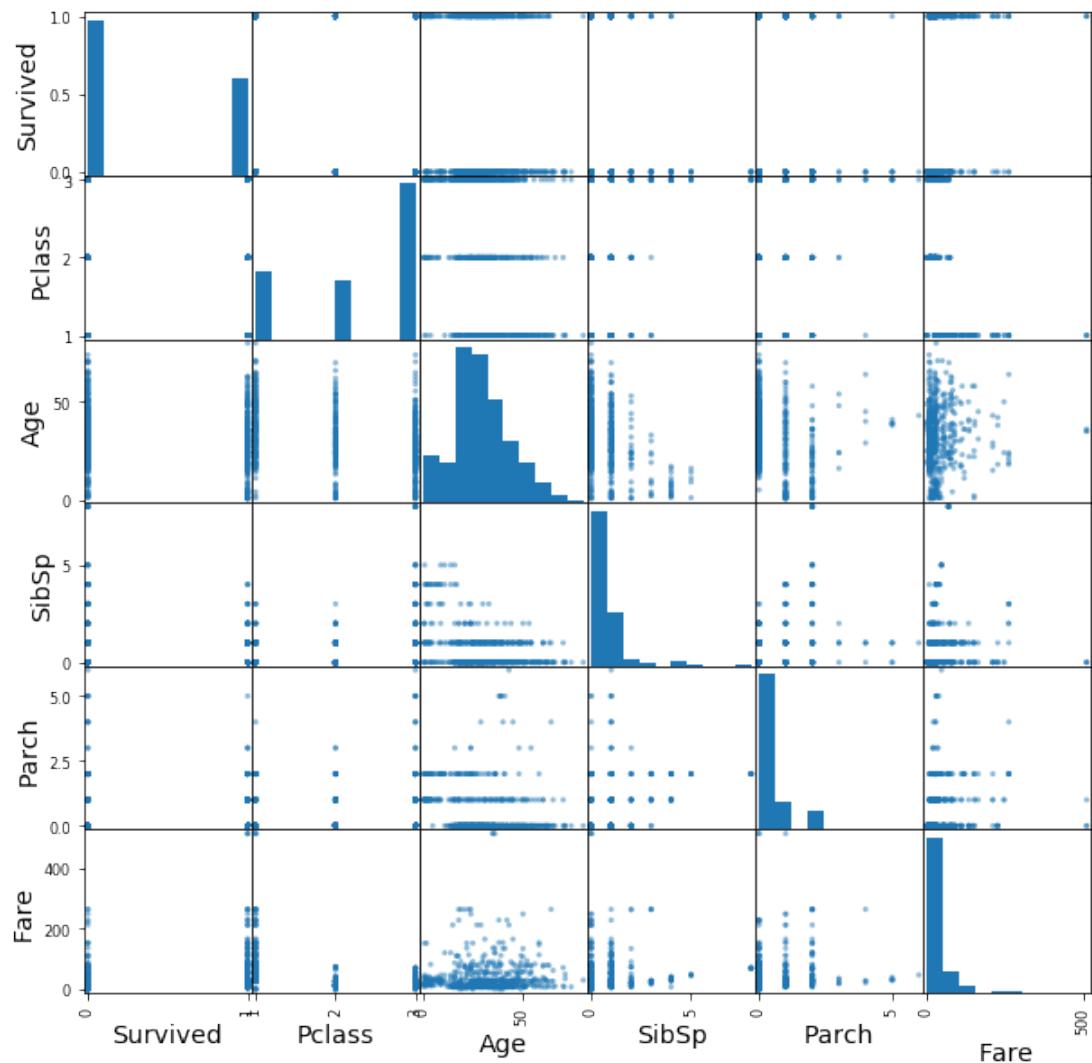
```
In [24]: df = pd.read_csv('../data/titanic-train.csv')
df.head()
```

Out [24] :

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

```
In [25]: from pandas.plotting import scatter_matrix
```

```
In [26]: scatter_matrix(df.drop('PassengerId', axis=1),
figsize=(10, 10));
```





# 17

## Machine Learning Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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

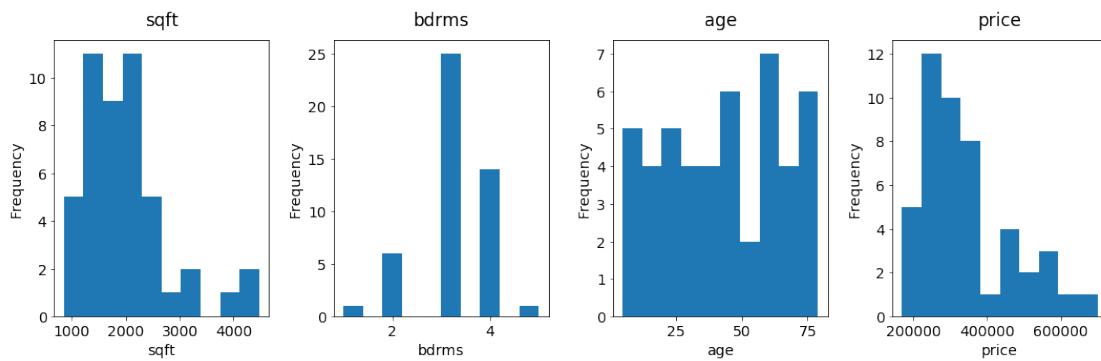
```
In [3]: # Load the dataset ../data/housing-data.csv
df = pd.read_csv('../data/housing-data.csv')
df.head()
```

Out[3] :

	sqft	bdrms	age	price
0	2104	3	70	399900
1	1600	3	28	329900
2	2400	3	44	369000
3	1416	2	49	232000
4	3000	4	75	539900

```
In [4]: # plot the histograms for each feature
plt.figure(figsize=(15, 5))
for i, feature in enumerate(df.columns):
    plt.subplot(1, 4, i+1)
    df[feature].plot(kind='hist', title=feature)
    plt.xlabel(feature)

plt.tight_layout()
```



```
In [5]: # 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 1 column (price)
X = df[['sqft', 'bdrms', 'age']].values
y = df['price'].values
```

```
In [6]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.optimizers import Adam, SGD
```

Using TensorFlow backend.

```
In [7]: # create a linear regression model in Keras
    # with the appropriate number of inputs and output
model = Sequential()
model.add(Dense(1, input_dim=3))
model.compile(Adam(lr=0.8), 'mean_squared_error')
```

```
In [8]: from sklearn.model_selection import train_test_split
```

```
In [9]: # split the data into train and test with a 20% test
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.2)
```

```
In [10]: # 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?
model.fit(X_train, y_train, epochs=20, verbose=0);
```

```
In [11]: df.describe()
```

Out[11] :

	sqft	bdrms	age	price
count	47.000000	47.000000	47.000000	47.000000
mean	2000.680851	3.170213	42.744681	340412.659574
std	794.702354	0.760982	22.873440	125039.899586
min	852.000000	1.000000	5.000000	169900.000000
25%	1432.000000	3.000000	24.500000	249900.000000
50%	1888.000000	3.000000	44.000000	299900.000000
75%	2269.000000	4.000000	61.500000	384450.000000
max	4478.000000	5.000000	79.000000	699900.000000

```
In [12]: # 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
df['sqft1000'] = df['sqft']/1000.0
df['age10'] = df['age']/10.0
df['price100k'] = df['price']/1e5
```

```
In [13]: X = df[['sqft1000', 'bdrms', 'age10']].values
y = df['price100k'].values
```

```
In [14]: X_train, X_test, y_train, y_test = \
train_test_split(X, y, test_size=0.2)
```

```
In [15]: model = Sequential()
model.add(Dense(1, input_dim=3))
model.compile(Adam(lr=0.1), 'mean_squared_error')
model.fit(X_train, y_train, epochs=20, verbose=0);
```

```
In [16]: from sklearn.metrics import r2_score
```

```
In [17]: # once you're satisfied with training, check the
# R2score on the test set

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

r_ = r2_score(y_train, y_train_pred)
print("R2 score on Train set is:\t{:0.3f}\t".format(r_))

r_ = r2_score(y_test, y_test_pred)
print("R2 score on Test set is:\t{:0.3f}\t".format(r_))
```

```
R2 score on Train set is:      0.655
R2 score on Test set is:      0.287
```

## 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.

```
In [18]: # load the dataset at ..../data/HR_comma_sep.csv, inspect
# it with `head()`, `info()` and `describe()`.

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

In [19]: `df.head()`

Out[19] :

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	sales	salary
0	0.38	0.53	2	157	3	0	1	0	sales	low
1	0.80	0.86	5	262	6	0	1	0	sales	medium
2	0.11	0.88	7	272	4	0	1	0	sales	medium
3	0.72	0.87	5	223	5	0	1	0	sales	low
4	0.37	0.52	2	159	3	0	1	0	sales	low

In [20]: `df.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14999 entries, 0 to 14998
Data columns (total 10 columns):
satisfaction_level    14999 non-null float64
last_evaluation        14999 non-null float64
number_project         14999 non-null int64
average_montly_hours  14999 non-null int64
time_spend_company     14999 non-null int64
Work_accident          14999 non-null int64
left                   14999 non-null int64
promotion_last_5years  14999 non-null int64
sales                  14999 non-null object
salary                 14999 non-null object
dtypes: float64(2), int64(6), object(2)
memory usage: 1.1+ MB
```

In [21]: `df.describe()`

Out[21] :

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years
count	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000
mean	0.612834	0.716102	3.803054	201.050337	3.498233	0.144610	0.238083	0.021268
std	0.248631	0.171169	1.232592	49.943099	1.460136	0.351719	0.425924	0.144281
min	0.090000	0.360000	2.000000	96.000000	2.000000	0.000000	0.000000	0.000000
25%	0.440000	0.560000	3.000000	156.000000	3.000000	0.000000	0.000000	0.000000
50%	0.640000	0.720000	4.000000	200.000000	3.000000	0.000000	0.000000	0.000000
75%	0.820000	0.870000	5.000000	245.000000	4.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	7.000000	310.000000	10.000000	1.000000	1.000000	1.000000

In [22]: *# Establish a benchmark: what would be your accuracy  
# score if you predicted everyone stay?*

```
df.left.value_counts() / len(df)
```

Out[22] :

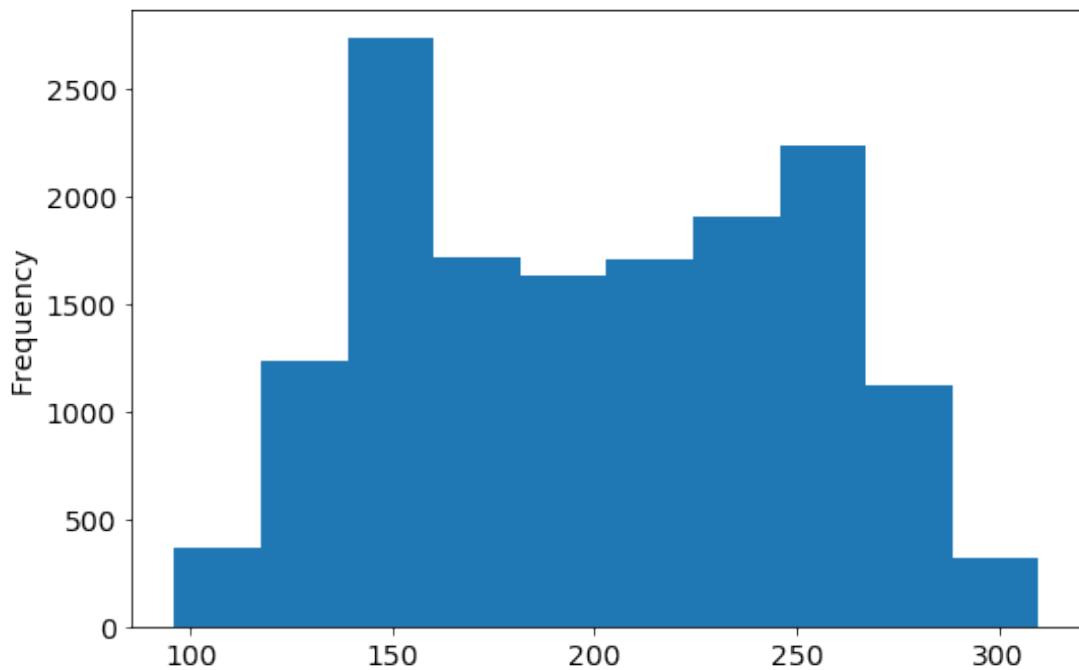
---

	left
o	0.761917
1	0.238083

---

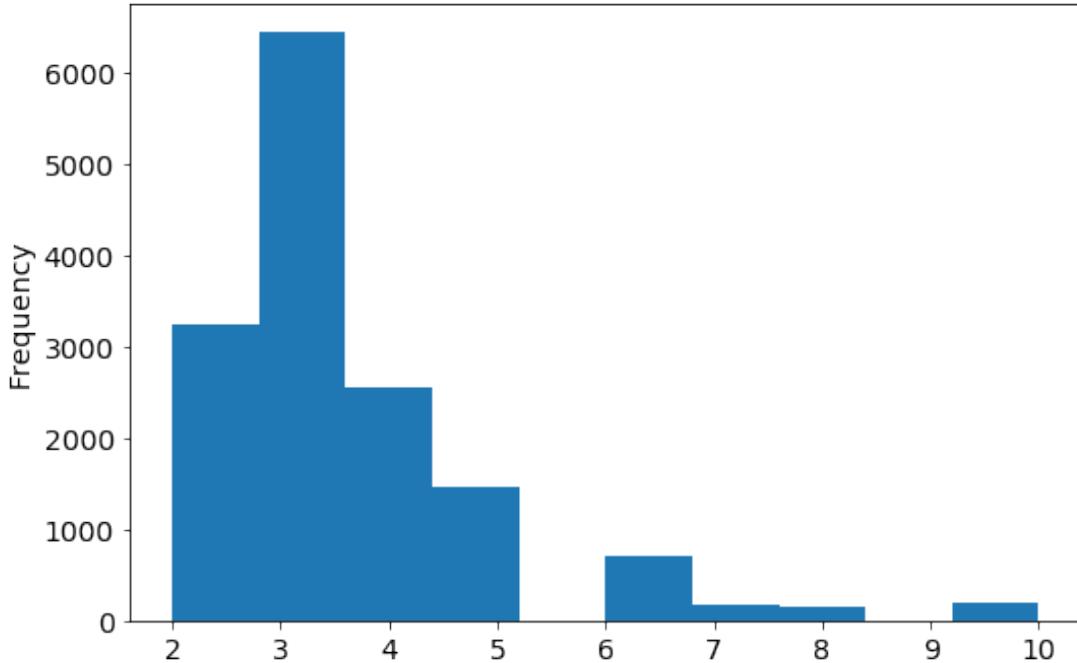
Predicting o all the time would yield an accuracy of 76%

```
In [23]: # Check if any feature needs rescaling.  
# You may plot a histogram of the feature to decide  
# which rescaling method is more appropriate.  
df['average_montly_hours'].plot(kind='hist');
```



```
In [24]: df['average_montly_hours_100'] = \  
        df['average_montly_hours']/100.0
```

```
In [25]: df['time_spend_company'].plot(kind='hist');
```



```
In [26]: # convert the categorical features into binary dummy columns.  
# You will then have to combine them with  
# the numerical features using `pd.concat`.  
df_dummies = pd.get_dummies(df[['sales', 'salary']])
```

```
In [27]: df.columns
```

```
Out[27]: Index(['satisfaction_level', 'last_evaluation', 'number_project',  
                 'average_montly_hours', 'time_spend_company', 'Work_accident', 'left',  
                 'promotion_last_5years', 'sales', 'salary', 'average_montly_hours_100'],  
                dtype='object')
```

```
In [28]: X = pd.concat([df[['satisfaction_level',  
                           'last_evaluation',  
                           'number_project',  
                           'time_spend_company',  
                           'Work_accident',  
                           'promotion_last_5years',  
                           'average_montly_hours_100']],  
                     df_dummies], axis=1).values  
y = df['left'].values
```

```
In [29]: X.shape
```

```
Out[29]: (14999, 20)
```

```
In [30]: # do the usual train/test split with a 20% test size
```

```
X_train, X_test, y_train, y_test = \  
    train_test_split(X, y, test_size=0.2)
```

```
In [31]: # play around with learning rate and optimizer
```

```
model = Sequential()  
model.add(Dense(1, input_dim=20, activation='sigmoid'))  
model.compile(Adam(lr=0.5),  
             'binary_crossentropy',  
             metrics=['accuracy'])
```

```
In [32]: model.fit(X_train, y_train);
```

```
Epoch 1/1  
11999/11999 [=====] - 1s 78us/step - loss: 0.5358 -  
acc: 0.7637
```

```
In [33]: y_test_pred = model.predict_classes(X_test)
```

```
In [34]: from sklearn.metrics import confusion_matrix  
        from sklearn.metrics import classification_report
```

```
In [35]: 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
```

```
In [36]: # check the confusion matrix, precision and recall
```

```
pretty_confusion_matrix(y_test, y_test_pred,  
                         labels=['Stay', 'Leave'])
```

Out [36] :

	Predicted Stay	Predicted Leave
Stay	1850	475
Leave	254	421

In [37]: `print(classification_report(y_test, y_test_pred))`

	precision	recall	f1-score	support
0	0.88	0.80	0.84	2325
1	0.47	0.62	0.54	675
avg / total	0.79	0.76	0.77	3000

In [38]: `from keras.wrappers.scikit_learn import KerasClassifier`

In [39]: *# check if you still get the same results if you use a 5-Fold cross validation on all the data*

```
def build_logistic_regr():
    model = Sequential()
    model.add(Dense(1, input_dim=20, activation='sigmoid'))
    model.compile(Adam(lr=0.5),
                  'binary_crossentropy',
                  metrics=['accuracy'])
    return model

model = KerasClassifier(build_fn=build_logistic_regr,
                       epochs=10, verbose=0)
```

In [40]: `from sklearn.model_selection import cross_val_score, KFold`

```
In [41]: cv = KFold(5, shuffle=True)
scores = cross_val_score(model, X, y, cv=cv)

print("Cross val accuracy is {:.4f} ± {:.4f}".format(
    scores.mean(), scores.std()))
```

Cross val accuracy is 0.7470 ± 0.0744

In [42]: `scores`

```
Out[42]: array([0.779      , 0.59866667, 0.79466667, 0.78      , 0.7825942 ])
```

```
In [43]: # Is the model good enough for your boss?
```

No, the model is not good enough for my boss, since it performs no better than the benchmark.



# 18

## Deep Learning Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

```
In [3]: import seaborn as sns
```

### 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)<sup>2</sup>)

- 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:

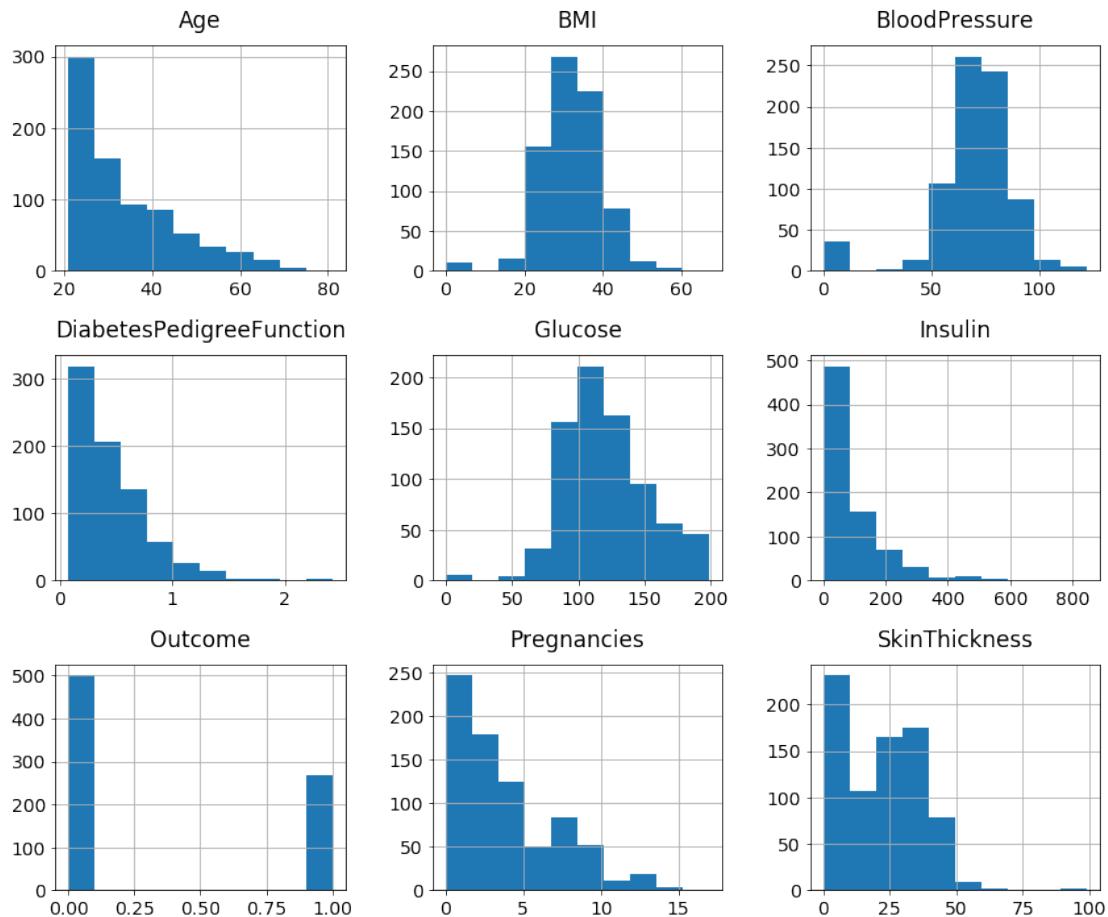
1. Load the ..data/diabetes.csv dataset, use pandas to explore the range of each feature
  - For each feature draw a histogram. Bonus points if you draw all the histograms in the same figure.
  - Explore correlations of features with the outcome column. You can do this in several ways, for example using 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?

```
In [4]: df = pd.read_csv('../data/diabetes.csv')
df.head()
```

Out [4] :

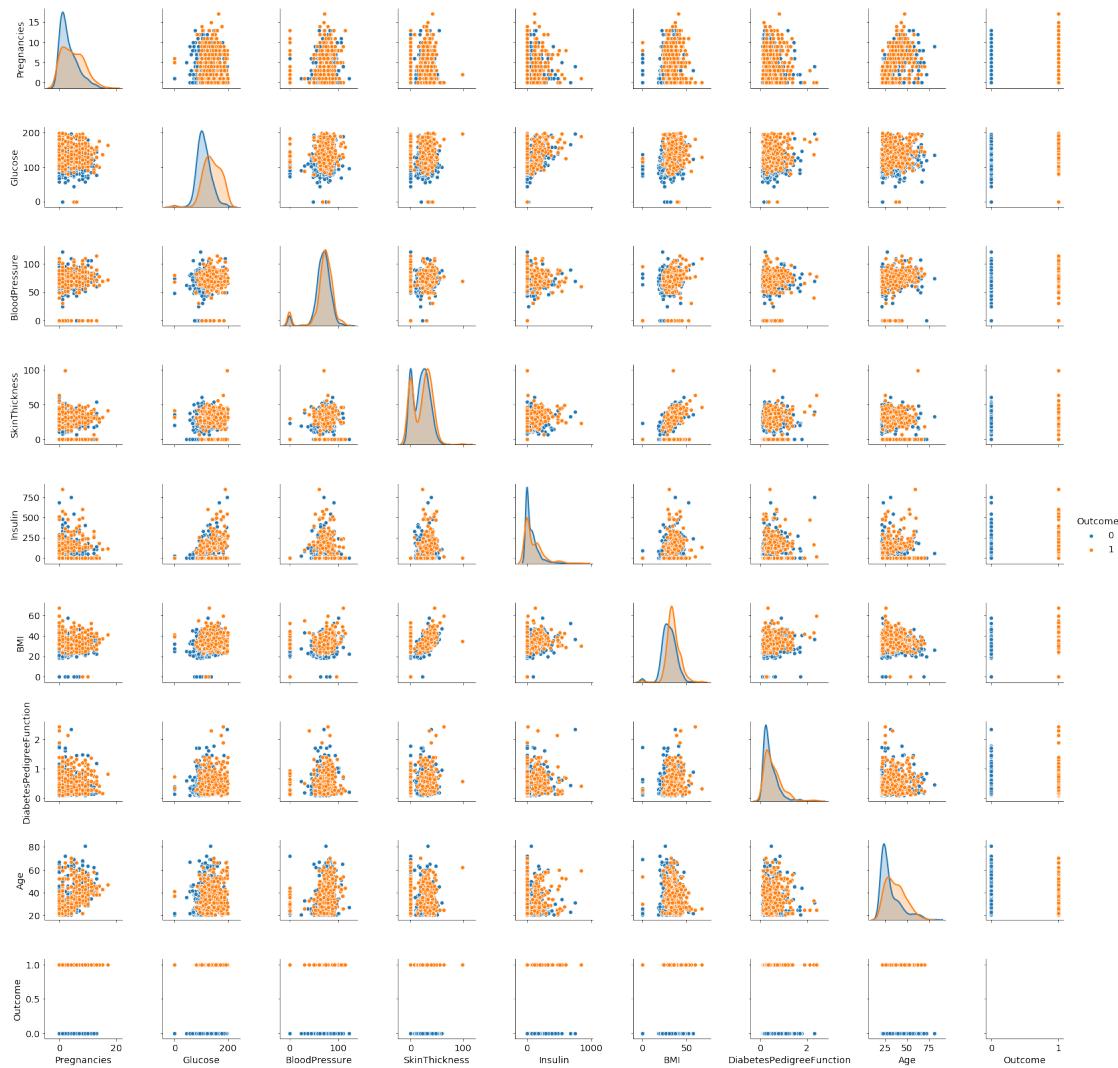
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
In [5]: df.hist(figsize=(12, 10))
plt.tight_layout()
```

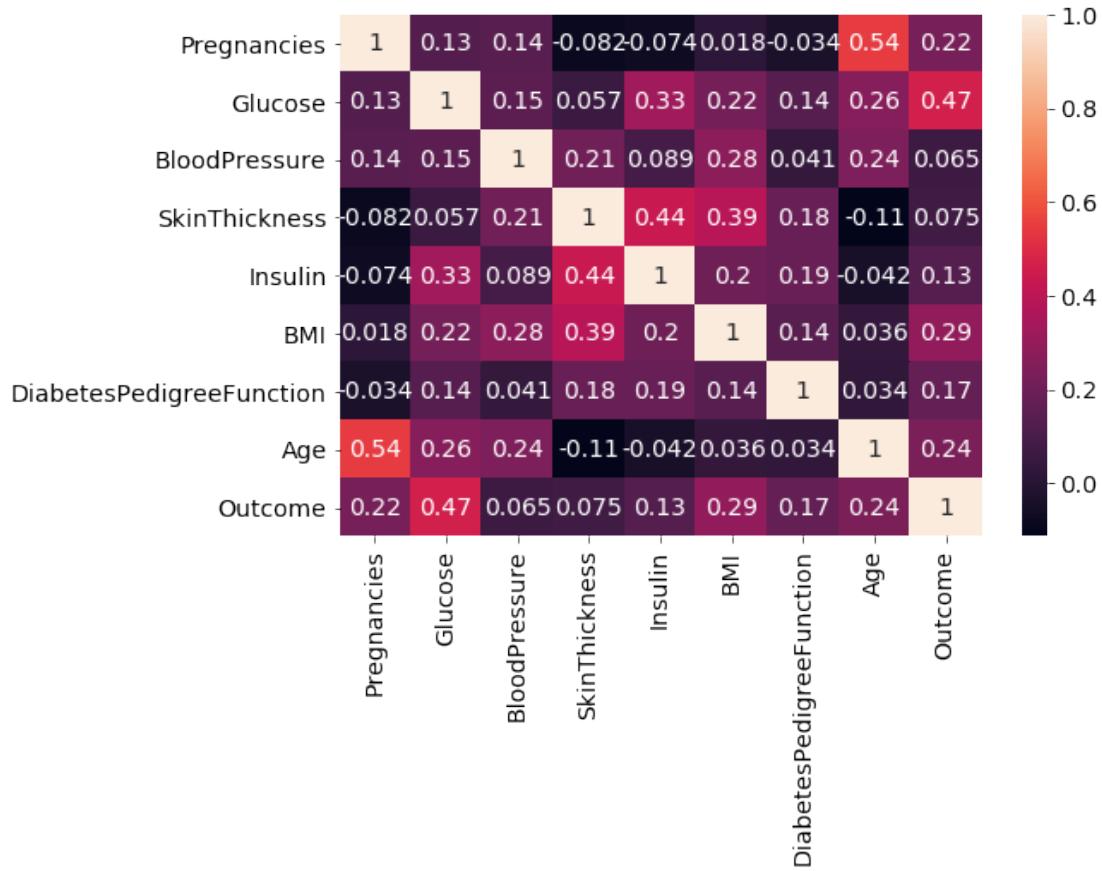


```
In [6]: sns.pairplot(df, hue='Outcome');
```

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kde.py:488: RuntimeWarning: invalid value
encountered in true_divide
    binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kdetools.py:34: RuntimeWarning: invalid value
encountered in double_scalars
    FAC1 = 2*(np.pi*bw/RANGE)**2
```



```
In [7]: sns.heatmap(df.corr(), annot = True);
```



In [8]: df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
Pregnancies          768 non-null int64
Glucose              768 non-null int64
BloodPressure        768 non-null int64
SkinThickness        768 non-null int64
Insulin              768 non-null int64
BMI                  768 non-null float64
DiabetesPedigreeFunction 768 non-null float64
Age                  768 non-null int64
Outcome              768 non-null int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

In [9]: df.describe()

Out [9] :

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

In [10]: `from sklearn.preprocessing import StandardScaler`

In [11]: `from keras.utils import to_categorical`

Using TensorFlow backend.

```
In [12]: sc = StandardScaler()
X = sc.fit_transform(df.drop('Outcome', axis=1))
y = df['Outcome'].values
y_cat = to_categorical(y)
```

## 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

```
In [13]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.optimizers import SGD, Adam
        from sklearn.model_selection import train_test_split
```

```
In [14]: X.shape
```

```
Out[14]: (768, 8)
```

```
In [15]: X_train, X_test, y_train, y_test = \
            train_test_split(X, y_cat,
                            random_state=22, test_size=0.2)
```

```
In [16]: model = Sequential()
        model.add(Dense(32, input_shape=(8,), activation='relu'))
        model.add(Dense(32, activation='relu'))
        model.add(Dense(2, activation='softmax'))
        model.compile(Adam(lr=0.05),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
```

```
In [17]: model.fit(X_train, y_train, epochs=20,
                  verbose=0, validation_split=0.1);
```

```
In [18]: y_pred = model.predict(X_test)
```

```
In [19]: y_test_class = np.argmax(y_test, axis=1)
        y_pred_class = np.argmax(y_pred, axis=1)
```

```
In [20]: from sklearn.metrics import accuracy_score
        from sklearn.metrics import confusion_matrix
        from sklearn.metrics import classification_report
```

```
In [21]: accuracy_score(y_test_class, y_pred_class)
```

```
Out[21]: 0.7272727272727273
```

```
In [22]: print(classification_report(y_test_class, y_pred_class))
```

	precision	recall	f1-score	support
0	0.73	0.92	0.81	100
1	0.71	0.37	0.49	54
avg / total	0.72	0.73	0.70	154

```
In [23]: confusion_matrix(y_test_class, y_pred_class)
```

```
Out[23]: array([[92,  8],
                 [34, 20]])
```

### 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?

```
In [24]: from sklearn.ensemble import RandomForestClassifier
         from sklearn.svm import SVC
         from sklearn.naive_bayes import GaussianNB

         for mod in [RandomForestClassifier(), SVC(), GaussianNB()]:
             mod.fit(X_train, y_train[:, 1])
             y_pred = mod.predict(X_test)
             print("="*80)
             print(mod)
             print("-"*80)
             acc_ = accuracy_score(y_test_class, y_pred)
             print("Accuracy score: {:.3f}".format(acc_))
             print("Confusion Matrix:")
             print(confusion_matrix(y_test_class, y_pred))
             print()

=====
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
```

```
min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
oob_score=False, random_state=None, verbose=0,
warm_start=False)
-----
Accuracy score: 0.747
Confusion Matrix:
[[90 10]
 [29 25]]
=====
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
max_iter=-1, probability=False, random_state=None, shrinking=True,
tol=0.001, verbose=False)
-----
Accuracy score: 0.721
Confusion Matrix:
[[89 11]
 [32 22]]
=====
GaussianNB(priors=None)
-----
Accuracy score: 0.708
Confusion Matrix:
[[87 13]
 [32 22]]
```



# 19

## Deep Learning Internals Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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

In [3]: `df = pd.read_csv('../data/wines.csv')`

In [4]: `df.head()`

Out[4] :

Class	Alcohol	Malic_acid	Ash	Alkalinity_of_ash	Magnesium	Total_phenols	Flavanoids	Nonflavanoid_phenols	Proanthocyanins	Color_intensity	Hue	OD280-OD315_of_diluted_wines	Proline	
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

In [5]: `y = df['Class']`

In [6]: `y.value_counts()`

Out[6] :

Class
2
1
3

In [7]: `y_cat = pd.get_dummies(y)`

In [8]: `y_cat.head()`

Out[8] :

	1	2	3
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0

In [9]: `X = df.drop('Class', axis=1)`

In [10]: X.shape

Out[10]: (178, 13)

In [11]: import seaborn as sns

In [12]: sns.pairplot(df, hue='Class')

```
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kde.py:488: RuntimeWarning: invalid value
encountered in true_divide
    binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/home/ubuntu/miniconda3/envs/ztdlbook/lib/python3.6/site-
packages/statsmodels/nonparametric/kdetools.py:34: RuntimeWarning: invalid value
encountered in double_scalars
    FAC1 = 2*(np.pi*bw/RANGE)**2
```

Out[12]: <seaborn.axisgrid.PairGrid at 0x7f78a836c400>



```
In [13]: from sklearn.preprocessing import StandardScaler
```

```
In [14]: sc = StandardScaler()
```

```
In [15]: Xsc = sc.fit_transform(X)
```

```
In [16]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.optimizers import SGD, Adam, Adadelta, RMSprop
        import keras.backend as K
```

Using TensorFlow backend.

```
In [17]: K.clear_session()
model = Sequential()
model.add(Dense(5, input_shape=(13,),
               kernel_initializer='he_normal',
               activation='relu'))
model.add(Dense(3, activation='softmax'))

model.compile(RMSprop(lr=0.1),
              'categorical_crossentropy',
              metrics=['accuracy'])

model.fit(Xsc, y_cat.values,
          batch_size=8,
          epochs=10,
          verbose=0,
          validation_split=0.2);
```

## 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?

```
In [18]: K.clear_session()
model = Sequential()
model.add(Dense(8, input_shape=(13,),
               kernel_initializer='he_normal',
```

```

        activation='tanh'))
model.add(Dense(5, kernel_initializer='he_normal',
               activation='tanh'))
model.add(Dense(2, kernel_initializer='he_normal',
               activation='tanh'))
model.add(Dense(3, activation='softmax'))

model.compile(RMSprop(lr=0.05),
              'categorical_crossentropy',
              metrics=['accuracy'])

model.fit(Xsc, y_cat.values,
           batch_size=16,
           epochs=20,
           verbose=0);

```

In [19]: model.summary()

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 8)	112
dense_2 (Dense)	(None, 5)	45
dense_3 (Dense)	(None, 2)	12
dense_4 (Dense)	(None, 3)	9

Total params: 178  
Trainable params: 178  
Non-trainable params: 0

In [20]: inp = model.layers[0].input  
out = model.layers[2].output

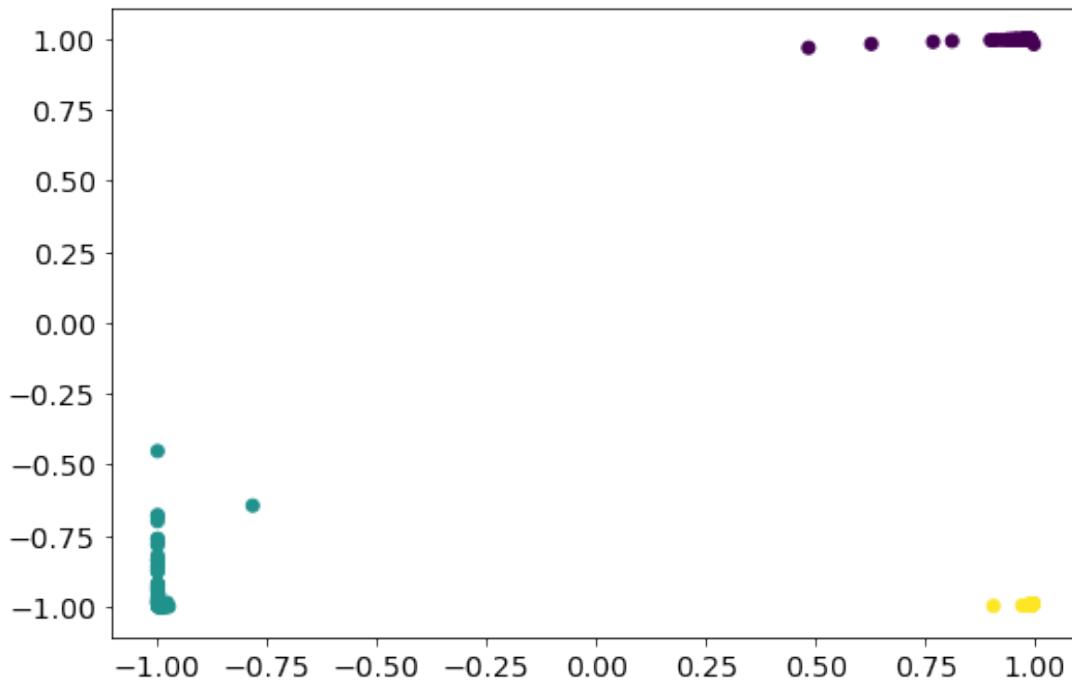
In [21]: features\_function = K.function([inp], [out])

In [22]: features = features\_function([Xsc])[0]

In [23]: features.shape

Out[23]: (178, 2)

```
In [24]: plt.scatter(features[:, 0], features[:, 1], c=y);
```



## 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

```
In [25]: from keras.layers import Input  
       from keras.models import Model
```

```
In [26]: K.clear_session()
```

```

inputs = Input(shape=(13,))
x = Dense(8, kernel_initializer='he_normal',
           activation='tanh')(inputs)
x = Dense(5, kernel_initializer='he_normal',
           activation='tanh')(x)
second_to_last = Dense(2, kernel_initializer='he_normal',
                       activation='tanh')(x)
outputs = Dense(3, activation='softmax')(second_to_last)

model = Model(inputs=inputs, outputs=outputs)

model.compile(RMSprop(lr=0.05),
              'categorical_crossentropy',
              metrics=['accuracy'])

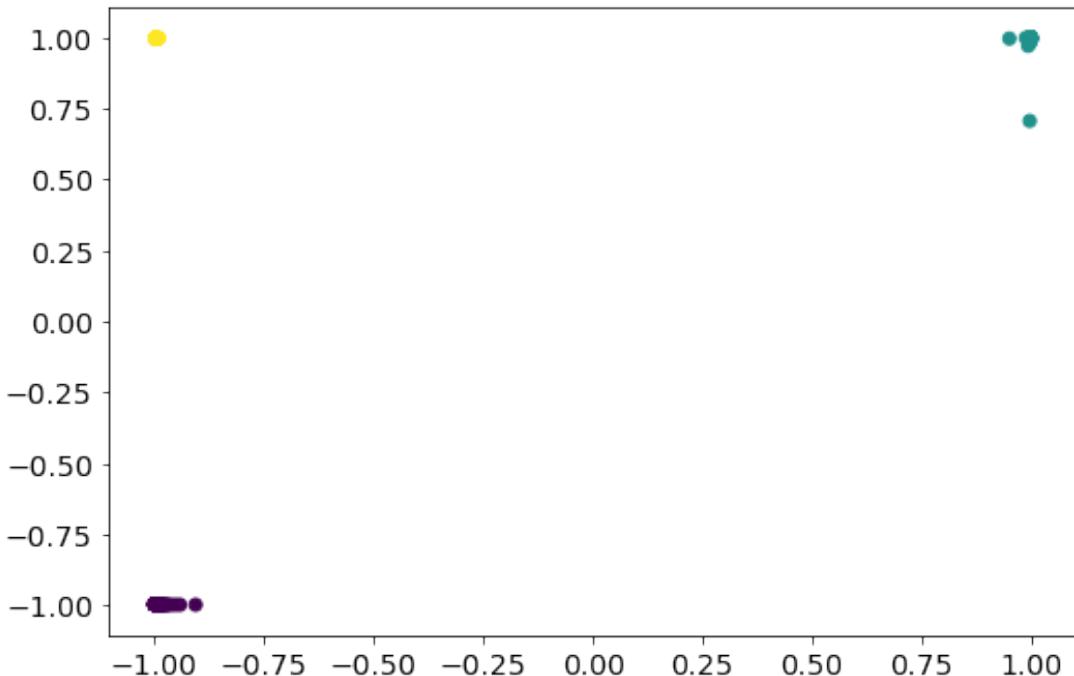
model.fit(Xsc, y_cat.values, batch_size=16,
          epochs=20, verbose=0);

```

In [27]: features\_function = K.function([inputs], [second\_to\_last])

In [28]: features = features\_function([Xsc])[0]

In [29]: plt.scatter(features[:, 0], features[:, 1], c=y);



## 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
- Watch the next video for an overview of tensorboard

```
In [30]: from keras.callbacks import ModelCheckpoint  
        from keras.callbacks import EarlyStopping  
        from keras.callbacks import TensorBoard
```

```
In [31]: checkpointer = ModelCheckpoint(  
            filepath="/tmp/ztdlbook/weights.hdf5",  
            verbose=1, save_best_only=True)
```

```
In [32]: earlystopper = EarlyStopping(  
            monitor='val_loss', min_delta=0, patience=1,  
            verbose=1, mode='auto')
```

```
In [33]: tensorboard = TensorBoard(  
            log_dir='/tmp/ztdlbook/tensorboard/')
```

```
In [34]: from sklearn.model_selection import train_test_split
```

```
In [35]: X_train, X_test, y_train, y_test = \  
            train_test_split(Xsc, y_cat.values,  
                            test_size=0.3, random_state=42)
```

```
In [36]: K.clear_session()
```

```
inputs = Input(shape=(13,))  
  
x = Dense(8, kernel_initializer='he_normal',
```

```

activation='tanh')(inputs)

x = Dense(5, kernel_initializer='he_normal',
           activation='tanh')(x)

second_to_last = Dense(2, kernel_initializer='he_normal',
                       activation='tanh')(x)

outputs = Dense(3, activation='softmax')(second_to_last)

model = Model(inputs=inputs, outputs=outputs)

model.compile(RMSprop(lr=0.05),
              'categorical_crossentropy',
              metrics=['accuracy'])

callbacks_ = [checkpointer, earlystopper, tensorboard]

model.fit(X_train, y_train, batch_size=32,
           epochs=20, verbose=0,
           validation_data=(X_test, y_test),
           callbacks=callbacks_);

```

Epoch 00001: val\_loss improved from inf to 0.51602, saving model to  
 /tmp/ztdlbook/weights.hdf5

Epoch 00002: val\_loss improved from 0.51602 to 0.33564, saving model to  
 /tmp/ztdlbook/weights.hdf5

Epoch 00003: val\_loss improved from 0.33564 to 0.27793, saving model to  
 /tmp/ztdlbook/weights.hdf5

Epoch 00004: val\_loss improved from 0.27793 to 0.19555, saving model to  
 /tmp/ztdlbook/weights.hdf5

Epoch 00005: val\_loss improved from 0.19555 to 0.16347, saving model to  
 /tmp/ztdlbook/weights.hdf5

Epoch 00006: val\_loss did not improve from 0.16347  
 Epoch 00006: early stopping

Run Tensorboard with the command:

```
tensorboard --logdir /tmp/ztdlbook/tensorboard/
```

# 20

## Convolutional Neural Networks Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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 intial 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?

```
In [3]: from keras.utils import np_utils, to_categorical
```

Using TensorFlow backend.

```
In [4]: from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D
from keras.layers import Flatten, Activation
import keras.backend as K
```

```
In [5]: from keras.datasets import mnist
```

```
In [6]: (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
In [7]: X_train.shape
```

```
Out[7]: (60000, 28, 28)
```

```
In [8]: X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
```

```
In [9]: 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(Conv2D(32, (3, 3), kernel_initializer='normal'))
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 #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
activation_1 (Activation)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 32)	9248
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 32)	0
activation_2 (Activation)	(None, 5, 5, 32)	0
flatten_1 (Flatten)	(None, 800)	0
dense_1 (Dense)	(None, 64)	51264
dense_2 (Dense)	(None, 10)	650
Total params:	61,482	
Trainable params:	61,482	
Non-trainable params:	0	

In [10]: `model.fit(X_train, 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 [=====] - 4s 91us/step - loss: 0.3433 -
acc: 0.8954 - val_loss: 0.1345 - val_acc: 0.9585
Epoch 2/5
42000/42000 [=====] - 2s 42us/step - loss: 0.1019 -
acc: 0.9686 - val_loss: 0.1279 - val_acc: 0.9606
Epoch 3/5
42000/42000 [=====] - 2s 42us/step - loss: 0.0688 -
acc: 0.9786 - val_loss: 0.0671 - val_acc: 0.9791
Epoch 4/5
42000/42000 [=====] - 2s 42us/step - loss: 0.0503 -
acc: 0.9844 - val_loss: 0.0752 - val_acc: 0.9769
Epoch 5/5

```

```
42000/42000 [=====] - 2s 42us/step - loss: 0.0403 -
acc: 0.9873 - val_loss: 0.0865 - val_acc: 0.9737
```

In [11]: `model.evaluate(X_test, y_test_cat)`

```
10000/10000 [=====] - 0s 39us/step
```

Out [11]: [0.0788327644346282, 0.9752]

## 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

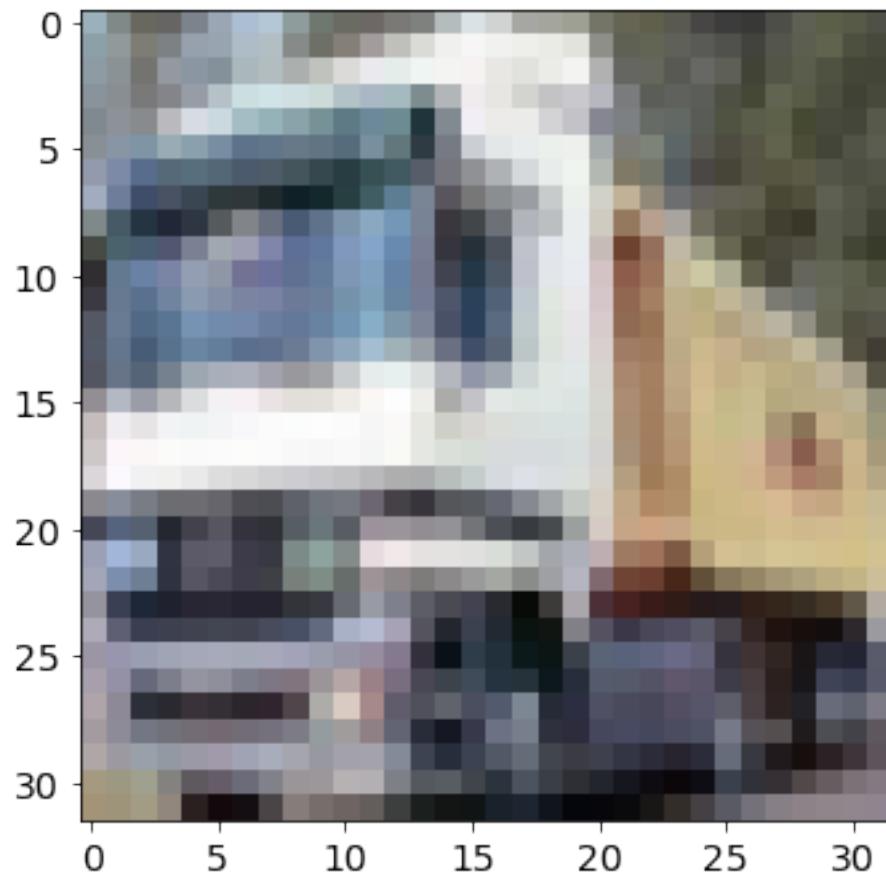
```
In [12]: from keras.datasets import cifar10
```

```
In [13]: (X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

```
In [14]: X_train.shape
```

```
Out[14]: (50000, 32, 32, 3)
```

```
In [15]: plt.imshow(X_train[1]);
```



```
In [16]: X_train = X_train.astype('float32') / 255.0  
X_test = X_test.astype('float32') / 255.0
```

```
In [17]: y_train.shape
```

```
Out[17]: (50000, 1)
```

```
In [18]: y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
```

```
In [19]: y_train_cat.shape
```

```
Out[19]: (50000, 10)
```

```
In [20]: 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(Conv2D(64, (3, 3), padding='same',
                 kernel_initializer='normal',
                 activation='relu'))

model.add(Conv2D(64, (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'))
```

```
In [21]: model.compile(loss='categorical_crossentropy',
                      optimizer='rmsprop',
                      metrics=['accuracy'])
```

```
In [22]: model.summary()
```

```
-----  
Layer (type)          Output Shape         Param #  
=====  
conv2d_3 (Conv2D)      (None, 32, 32, 32)     896  
-----  
conv2d_4 (Conv2D)      (None, 30, 30, 32)     9248  
-----  
max_pooling2d_3 (MaxPooling2D) (None, 15, 15, 32) 0  
-----  
conv2d_5 (Conv2D)      (None, 15, 15, 64)    18496  
-----  
conv2d_6 (Conv2D)      (None, 13, 13, 64)    36928  
-----  
max_pooling2d_4 (MaxPooling2D) (None, 6, 6, 64) 0  
-----  
flatten_2 (Flatten)    (None, 2304)           0  
-----  
dense_3 (Dense)        (None, 512)            1180160  
-----  
dense_4 (Dense)        (None, 10)             5130  
=====  
Total params: 1,250,858  
Trainable params: 1,250,858  
Non-trainable params: 0  
-----
```

```
In [23]: model.fit(X_train, y_train_cat,  
                  batch_size=256,  
                  epochs=2,  
                  validation_data=(X_test, y_test_cat),  
                  shuffle=True);
```

```
Train on 50000 samples, validate on 10000 samples  
Epoch 1/2  
50000/50000 [=====] - 6s 126us/step - loss: 1.8329 -  
acc: 0.3420 - val_loss: 1.6646 - val_acc: 0.3996  
Epoch 2/2  
50000/50000 [=====] - 6s 112us/step - loss: 1.4143 -  
acc: 0.4993 - val_loss: 1.3521 - val_acc: 0.5152
```



# 21

## Time Series and Recurrent Neural Networks Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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.

```
In [3]: df = pd.read_csv('../data/ZonalDemands_2003-2016.csv.bz2',  
                      compression='bz2',  
                      engine='python')
```

```
In [4]: def combine_date_hour(row):
    date = pd.to_datetime(row['Date'])
    hour = pd.Timedelta("%d hours" % row['Hour'])
    return date + hour

idx = df.apply(combine_date_hour, axis=1)
df = df.set_index(idx)

In [5]: split_date = pd.Timestamp('01-01-2014')
train = df.loc[:split_date, ['Total Ontario']].copy()
test = df.loc[split_date:, ['Total Ontario']].copy()

In [6]: offset = 10000
scale = 5000

train_sc = (train - offset) / scale
test_sc = (test - offset) / scale

In [7]: 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):
            col_ = 'T_{}'.format(start_lag + s)
            X[col_] = data.shift(start_lag + s)

    X = X.dropna()
    idx = X.index
    y = data.loc[idx]
    return X, y

In [8]: start_lag=36
window_len=72

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)

In [9]: 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
```

```
In [10]: from keras.models import Sequential
from keras.layers import LSTM, Dense
import keras.backend as K
from keras.optimizers import Adam
```

Using TensorFlow backend.

```
In [11]: K.clear_session()
```

```
model = Sequential()
model.add(LSTM(12, input_shape=(window_len, 1),
              kernel_initializer='normal',
              return_sequences=True))
model.add(LSTM(6, kernel_initializer='normal'))
model.add(Dense(1))

model.compile(optimizer=Adam(lr=0.05),
              loss='mean_squared_error')
```

```
In [12]: model.fit(X_train_t, y_train_t,
                  epochs=5,
                  batch_size=256,
                  verbose=1);
```

```
Epoch 1/5
93445/93445 [=====] - 73s 782us/step - loss: 0.2414
Epoch 2/5
93445/93445 [=====] - 71s 755us/step - loss: 0.1333
Epoch 3/5
93445/93445 [=====] - 71s 765us/step - loss: 0.1157
Epoch 4/5
93445/93445 [=====] - 71s 764us/step - loss: 0.1110
Epoch 5/5
93445/93445 [=====] - 72s 768us/step - loss: 0.0986
```

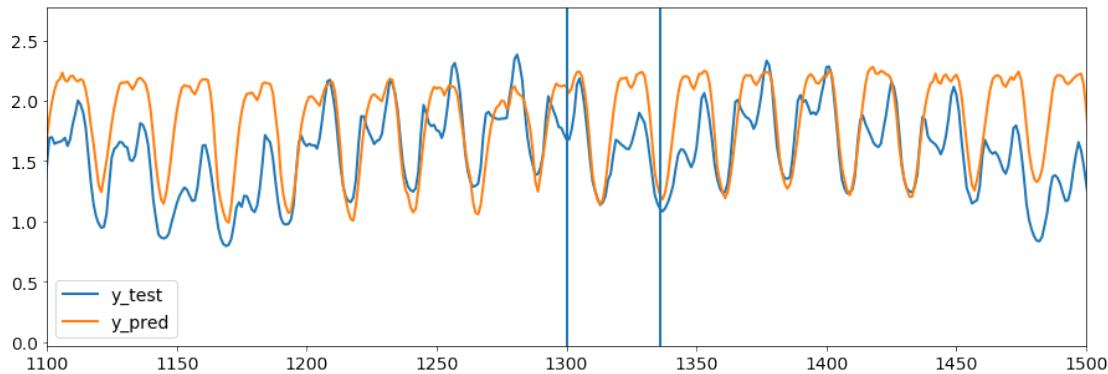
Let's compare the predictions on the test set. We will add a few days of data and put vertical bars to mark an interval of 36 hours:

```
In [13]: 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(1100,1500)
plt.axvline(1300)
plt.axvline(1336);

```



## Exercise 2

Try swapping the LSTM layer with a GRU layer and re-train the model. Does its performance improve on the 36 hours lag task?

In [14]: `from keras.layers import GRU`

In [15]: `K.clear_session()`

```

model = Sequential()
model.add(GRU(12, input_shape=(window_len, 1),
             kernel_initializer='normal',
             return_sequences=True))
model.add(GRU(6, kernel_initializer='normal'))
model.add(Dense(1))

model.compile(optimizer=Adam(lr=0.05),
              loss='mean_squared_error')

```

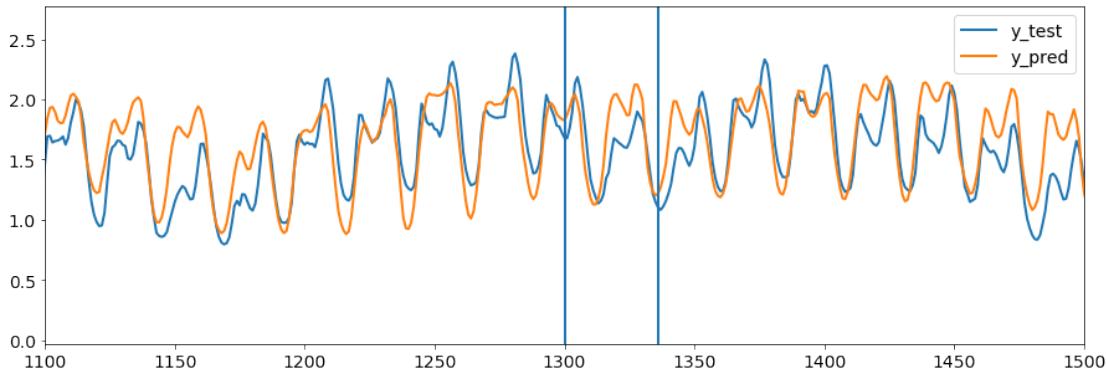
In [16]: `model.fit(X_train_t, y_train_t,
 epochs=5,
 batch_size=256,
 verbose=1);`

```

Epoch 1/5
93445/93445 [=====] - 60s 640us/step - loss: 0.1378
Epoch 2/5
93445/93445 [=====] - 58s 624us/step - loss: 0.0783
Epoch 3/5
93445/93445 [=====] - 58s 624us/step - loss: 0.0718
Epoch 4/5
93445/93445 [=====] - 58s 618us/step - loss: 0.0728
Epoch 5/5
93445/93445 [=====] - 58s 621us/step - loss: 0.0676

```

```
In [17]: 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(1100,1500)
    plt.axvline(1300)
    plt.axvline(1336);
```



GRU not only trains faster, but also seems to reach a better performance than LSTM on this 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?

```
In [18]: X_train = X_train_t.squeeze()
X_test = X_test_t.squeeze()
```

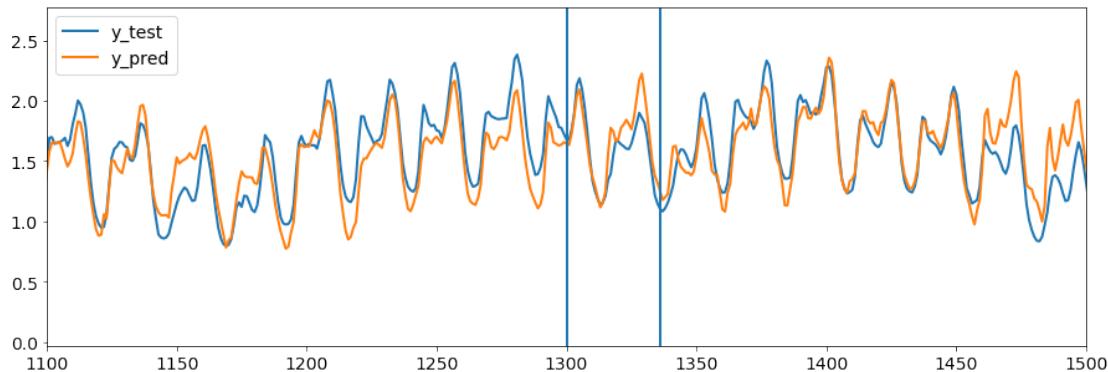
```
In [19]: model = Sequential()
```

```
    model.add(Dense(24, input_dim=window_len, activation='relu'))
    model.add(Dense(12, activation='relu'))
    model.add(Dense(6, activation='relu'))
    model.add(Dense(1))

    model.compile(optimizer='adam', loss='mean_squared_error')
```

```
In [20]: model.fit(X_train, y_train_t,
                  epochs=50,
                  batch_size=256,
                  verbose=0);
```

```
In [21]: y_pred = model.predict(X_test, 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(1100,1500)
plt.axvline(1300)
plt.axvline(1336);
```



## 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.
- 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:

$$\nu_t = 100 \times \frac{x_t - x_{t-1}}{x_{t-1}}$$

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.

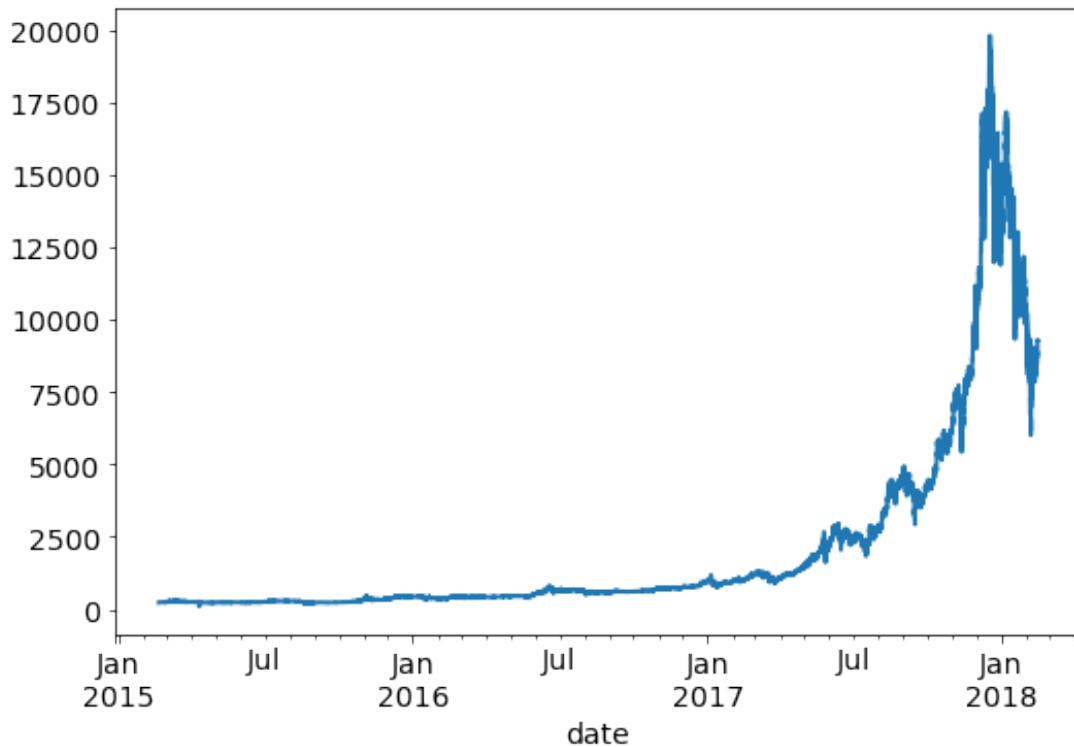
- Create a binary outcome variable that is 1 when `train[weightedAverage] >= 0` and 0 otherwise. This is going to be our label.
- Repeat the same operations on the test data
- Create a model to work with this data. Make sure the input layer has the right `input_shape` and the output layer has 1 node 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.

```
In [22]: df = pd.read_json('../data/poloniex_usdt_btc.json.gz',
                           compression='gzip')
```

```
In [23]: df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)
```

```
In [24]: df['close'].plot();
```



```
In [25]: df_percent = ((df - df.shift()) / df.shift()) * 100.0
```

```
In [26]: df_percent.head()
```

Out [26] :

date	close	high	low	open	quoteVolume	volume	weightedAverage
2015-02-19 19:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2015-02-19 19:30:00	0.000000	0.000000	0.000000	0.000000	-100.000000	-100.000000	0.000000
2015-02-19 20:00:00	6.666667	6.666667	0.000000	0.000000	inf	inf	5.818701
2015-02-19 20:30:00	1.666667	1.666667	8.444444	8.444444	-53.317086	-52.158715	2.481361
2015-02-19 21:00:00	0.000000	0.000000	0.000000	0.000000	-100.000000	-100.000000	0.000000

```
In [27]: df_percent = df_percent.dropna() \
          .replace(-np.inf, 0) \
          .replace(np.inf, 0)
```

```
In [28]: split_date = pd.Timestamp('01-01-2017')
```

```
train = df_percent.loc[:split_date].copy()
test = df_percent.loc[split_date: ].copy()
```

```
In [29]: def create_lagged_Xy_win_t(data, start_lag=1, window_len=1):

    X = data[['close', 'high', 'low', 'open']]
    y = data['weightedAverage'] >= 0

    rows, columns = X.shape
    shape_ = (rows - window_len - 1, window_len, columns)
    X_t = np.zeros(shape_)
    y_t = y.values[window_len + 1:]

    for lag in range(0, window_len):
        all_values = X.shift(start_lag + lag).values
        X_t[:, lag, :] = all_values[window_len + 1:]

    return X_t, y_t
```

```
In [30]: start_lag = 1
         window_len = 36
```

```
In [31]: X_train_t, y_train_t = create_lagged_Xy_win_t(
            train, start_lag, window_len)

X_test_t, y_test_t = create_lagged_Xy_win_t(
            test, start_lag, window_len)
```

```
In [32]: X_train_t.shape
```

```
Out[32]: (23956, 36, 4)
```

```
In [33]: y_train_t.shape
```

```
Out[33]: (23956,)
```

```
In [34]: K.clear_session()
```

```
model = Sequential()
model.add(GRU(24, input_shape=(window_len, 4),
              kernel_initializer='normal',
```

```

        return_sequences=True))
model.add(GRU(18, kernel_initializer='normal',
              return_sequences=True))
model.add(GRU(12, kernel_initializer='normal'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer=Adam(lr=0.05),
              loss='binary_crossentropy',
              metrics=['accuracy'])

```

In [35]: model.summary()

Layer (type)	Output Shape	Param #
gru_1 (GRU)	(None, 36, 24)	2088
gru_2 (GRU)	(None, 36, 18)	2322
gru_3 (GRU)	(None, 12)	1116
dense_1 (Dense)	(None, 1)	13
Total params:	5,539	
Trainable params:	5,539	
Non-trainable params:	0	

In [36]: model.fit(X\_train\_t, y\_train\_t,
 epochs=20,
 batch\_size=512,
 verbose=1);

```

Epoch 1/20
23956/23956 [=====] - 7s 291us/step - loss: 0.6866 -
acc: 0.5474
Epoch 2/20
23956/23956 [=====] - 6s 231us/step - loss: 0.6740 -
acc: 0.5669
Epoch 3/20
23956/23956 [=====] - 6s 230us/step - loss: 0.6578 -
acc: 0.6137
Epoch 4/20
23956/23956 [=====] - 5s 229us/step - loss: 0.6543 -
acc: 0.6145
Epoch 5/20
23956/23956 [=====] - 6s 232us/step - loss: 0.6515 -
acc: 0.6191
Epoch 6/20
23956/23956 [=====] - 6s 231us/step - loss: 0.6611 -

```

```

acc: 0.6088
Epoch 7/20
23956/23956 [=====] - 5s 228us/step - loss: 0.6813 -
acc: 0.5672
Epoch 8/20
23956/23956 [=====] - 6s 232us/step - loss: 0.6870 -
acc: 0.5502
Epoch 9/20
23956/23956 [=====] - 5s 227us/step - loss: 0.6800 -
acc: 0.5711
Epoch 10/20
23956/23956 [=====] - 5s 229us/step - loss: 0.6749 -
acc: 0.5859
Epoch 11/20
23956/23956 [=====] - 5s 228us/step - loss: 0.6758 -
acc: 0.5904
Epoch 12/20
23956/23956 [=====] - 6s 230us/step - loss: 0.6685 -
acc: 0.6049
Epoch 13/20
23956/23956 [=====] - 5s 228us/step - loss: 0.6732 -
acc: 0.5915
Epoch 14/20
23956/23956 [=====] - 5s 228us/step - loss: 0.6796 -
acc: 0.5754
Epoch 15/20
23956/23956 [=====] - 5s 228us/step - loss: 0.6783 -
acc: 0.5792
Epoch 16/20
23956/23956 [=====] - 5s 227us/step - loss: 0.6699 -
acc: 0.5933
Epoch 17/20
23956/23956 [=====] - 5s 228us/step - loss: 0.6674 -
acc: 0.5971
Epoch 18/20
23956/23956 [=====] - 5s 227us/step - loss: 0.6675 -
acc: 0.6008
Epoch 19/20
23956/23956 [=====] - 5s 229us/step - loss: 0.6651 -
acc: 0.6007
Epoch 20/20
23956/23956 [=====] - 5s 226us/step - loss: 0.6639 -
acc: 0.6036

```

In [37]: `model.evaluate(X_test_t, y_test_t)`

```
19633/19633 [=====] - 27s 1ms/step
```

Out [37]: [0.6477084021715723, 0.6354607039320543]

In [38]: `pd.Series(y_test_t).value_counts() / len(y_test_t)`

Out[38] :

---

---

	o
True	0.527326
False	0.472674

---



# 22

## Natural Language Processing and Text Data Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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

Let's reload the sms data we have previously saved:

```
In [3]: df = pd.read_csv('../data/sms_spam.csv')
df.head()
```

Out [3] :

	message	spam
0	Hi Princess! Thank you for the pics. You are v...	o
1	Hello my little party animal! I just thought I...	o
2	And miss vday the parachute and double coins??...	o
3	Maybe you should find something else to do ins...	o
4	What year. And how many miles.	o

Train/Test split on the messages, notice that we use Numpy Arrays, not Pandas Dataframes:

```
In [4]: from sklearn.model_selection import train_test_split
```

```
In [5]: docs_train, docs_test, y_train, y_test = \
    train_test_split(df['message'].values,
                    df['spam'].values,
                    random_state=0)
```

Now let's write the function:

```
In [6]: from sklearn.feature_extraction.text import CountVectorizer
from keras.models import Sequential
from keras.layers import Dense
```

Using TensorFlow backend.

```
In [7]: def train_for_vocab_size(vocab_size):
    vect = CountVectorizer(decode_error='ignore',
                          stop_words='english',
                          max_features=vocab_size)

    vect.fit(docs_train)
    X_train_sparse = vect.transform(docs_train)
    X_train = X_train_sparse.todense()
```

```

X_test_sparse = vect.transform(docs_test)
X_test = X_test_sparse.todense()

input_dim = X_train.shape[1]

model = Sequential()
model.add(Dense(1, input_dim=input_dim, activation='sigmoid'))

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

model.fit(X_train, y_train, epochs=20, verbose=0)

train_acc = model.evaluate(X_train, y_train, verbose=0)[1]
test_acc = model.evaluate(X_test, y_test, verbose=0)[1]
return input_dim, train_acc, test_acc

```

Now let's try a few vocab\_sizes with increasing separation:

```

In [8]: sizes = [2, 3, 5, 10, 30, 50, 100, 300, \
               500, 1000, 3000, 5000, 10000]
idx = []
train_accs = []
test_accs = []

for v in sizes:
    i, tra, tea = train_for_vocab_size(v)

    idx.append(i)
    train_accs.append(tra)
    test_accs.append(tea)

print("Done vocab size: ", i)

Done vocab size:  2
Done vocab size:  3
Done vocab size:  5
Done vocab size:  10
Done vocab size:  30
Done vocab size:  50
Done vocab size:  100
Done vocab size:  300
Done vocab size:  500
Done vocab size:  1000
Done vocab size:  3000
Done vocab size:  5000
Done vocab size:  7150

```

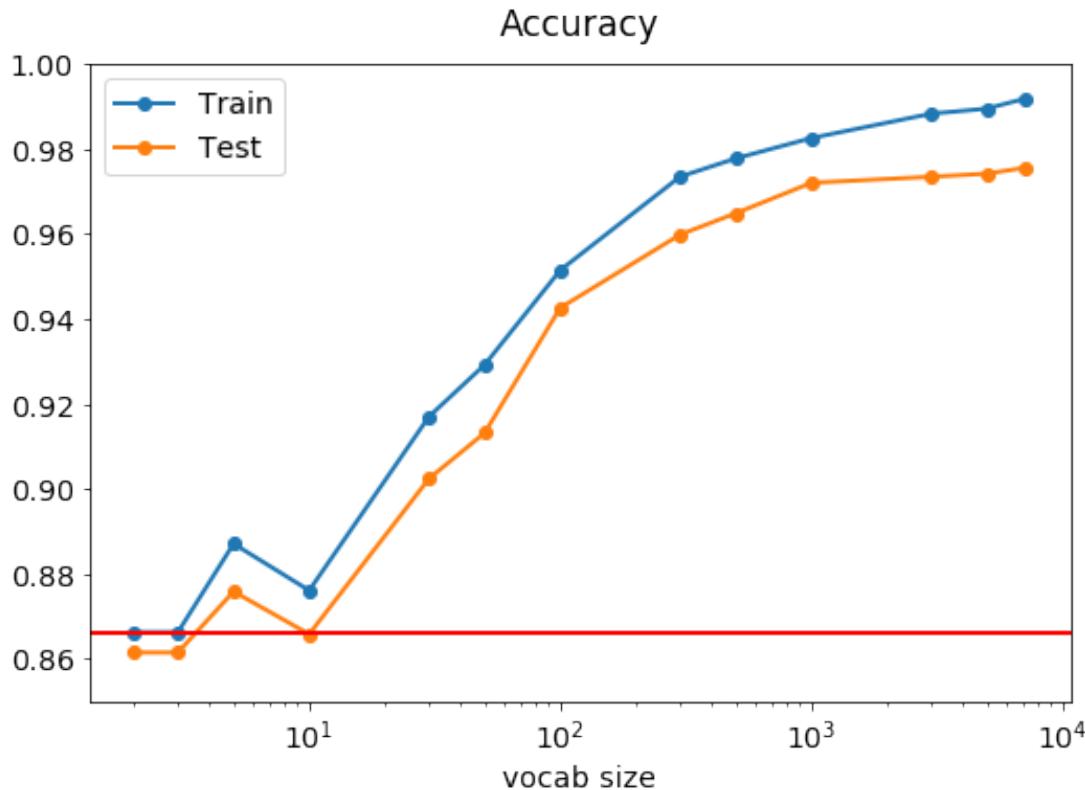
Let's organize the results in a DataFrame

```
In [9]: resdf = pd.DataFrame(train_accs,
                           columns=['Train'],
                           index=idx)

resdf['Test'] = test_accs
```

and let's plot the results using the logarithmic scale for the x axis. Remember that our benchmark accuracy is 86.6%, so we will add a baseline at that level:

```
In [10]: resdf.plot(logx=True, style='-o', title='Accuracy')
plt.xlabel('vocab size')
plt.ylim(0.85, 1)
plt.axhline(0.866, color='red');
```



## 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:

In [11]: `vocab_size=20000`

In [12]: `from keras.datasets import imdb`

In [13]: `(X_train, y_train), (X_test, y_test) = \\\n imdb.load_data(num_words=vocab_size)`

In [14]: `X_train.shape`

Out[14]: `(25000,)`

In [15]: `X_train[0][:10]`

Out[15]: `[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]`

Let's use a maximum review length of 80 words. This seems long enough to express an opinion about the movie:

```
In [16]: maxlen = 80
```

We will pad sequences using the default padding='pre' and truncating='pre' parameters.

```
In [17]: from keras.preprocessing.sequence import pad_sequences
```

```
In [18]: X_train_pad = pad_sequences(X_train, maxlen=maxlen)
X_test_pad = pad_sequences(X_test, maxlen=maxlen)
```

Let's build the model:

```
In [19]: embedding_size = 100
```

```
In [20]: from keras.layers import LSTM, Embedding
```

```
In [21]: model = Sequential()
model.add(Embedding(vocab_size, embedding_size))
model.add(LSTM(64, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

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

TIP: in the above model we have used dropout, which has not yet been formally introduced. For now just know that it's a technique aimed at reducing overfitting.

```
In [22]: model.summary()
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 100)	2000000

```
-----  
lstm_1 (LSTM)           (None, 64)          42240  
dense_14 (Dense)        (None, 1)            65  
=====  
Total params: 2,042,305  
Trainable params: 2,042,305  
Non-trainable params: 0  
-----
```

Let's train the model for a couple of epochs. If you run this model on your laptop it may take a few minutes for each epoch:

```
In [23]: model.fit(X_train_pad, y_train,  
                  batch_size=32,  
                  epochs=2,  
                  validation_split=0.3);
```

```
Train on 17500 samples, validate on 7500 samples  
Epoch 1/2  
17500/17500 [=====] - 75s 4ms/step - loss: 0.4944 -  
acc: 0.7588 - val_loss: 0.4104 - val_acc: 0.8177  
Epoch 2/2  
17500/17500 [=====] - 73s 4ms/step - loss: 0.3117 -  
acc: 0.8762 - val_loss: 0.4061 - val_acc: 0.8187
```

And let's evaluate the training and test accuracies

```
In [24]: train_loss, train_acc = model.evaluate(X_train_pad, y_train)  
  
      print('Train loss:', train_loss)  
      print('Train accuracy:', train_acc)  
  
25000/25000 [=====] - 20s 794us/step  
Train loss: 0.2798962456035614  
Train accuracy: 0.89424
```

```
In [25]: test_loss, test_acc = model.evaluate(X_test_pad, y_test)  
  
      print('Test loss:', test_loss)  
      print('Test accuracy:', test_acc)
```

```
25000/25000 [=====] - 20s 793us/step  
Test loss: 0.40965480022430417
```

```
Test accuracy: 0.81556
```

Not bad! We have a sentiment analysis model that we can unleash on the social media of our choice. Time to go to an investor and raise money! Not quite, but it's nice to see how easy it has become to build a model that would have been unthinkable just a few years ago.

Finally, for the bonus question. Let's get the word index:

```
In [26]: idx = imdb.get_word_index()
```

and let's create the reverse index. Notice that the documentation of `imdb.load_data` reads:

```
"""
```

*Signature: `imdb.load_data(path='imdb.npz', num_words=None, skip_top=0, maxlen=None, seed=113, st`*

*Docstring:*

*Loads the IMDB dataset.*

*path: where to cache the data (relative to `~/.keras/dataset`).*

*num\_words: max number of words to include. Words are ranked by how often they occur (in the training set) and only the most frequent words are kept*

*skip\_top: skip the top N most frequently occurring words (which may not be informative).*

*maxlen: truncate sequences after this length.*

*seed: random seed for sample shuffling.*

*start\_char: The start of a sequence will be marked with this character. Set to 1 because 0 is usually the padding character.*

*oov\_char: words that were cut out because of the `num\_words` or `skip\_top` limit will be replaced with this character.*

*index\_from: index actual words with this index and higher.*

*Tuple of Numpy arrays: `(x\_train, y\_train), (x\_test, y\_test)`.*

```
"""
```

so we will need to shift all indices by three to recover meaningful sentences:

```
In [27]: rev_idx = {v+3:k for k,v in idx.items()}
```

Also, following the documentation let's add the start character and the out-of-vocabulary character:

```
In [28]: rev_idx[1] = 'start_char'  
       rev_idx[2] = 'oov_char'
```

We can then apply the reverse index to recover the text of a review:

```
In [29]: example_review = ' '.join([rev_idx[word] for word in X_train[0]])  
example_review
```

```
Out[29]: "start_char this film was just brilliant casting location scenery story direction every
```

Great! These are indeed movie reviews.



# 23

## Training with GPUs Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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`

```
In [3]: from time import time  
        from keras.datasets import imdb
```

```
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding
from keras.preprocessing.sequence import pad_sequences
from keras.utils import multi_gpu_model
import tensorflow as tf
```

Using TensorFlow backend.

```
In [4]: vocab_size= 2000
        maxlen=80
```

```
In [5]: (X_train, y_train), (X_test, y_test) = \
        imdb.load_data(num_words=vocab_size)
```

```
X_train_pad = pad_sequences(X_train, maxlen=maxlen)
X_test_pad = pad_sequences(X_test, maxlen=maxlen)
```

```
In [6]: def recurrent_model(vocab_size, maxlen):
    print("Defining recurrent model")
    t0 = time()
    model = Sequential()
    model.add(Embedding(vocab_size, 100, input_length=maxlen))
    model.add(LSTM(64, dropout=0.2, recurrent_dropout=0.2))
    model.add(Dense(1, activation='sigmoid'))

    print("{:0.3f} seconds.".format(time() - t0))

    print("Compiling the model...")
    t0 = time()
    model.compile(loss='binary_crossentropy',
                  optimizer='rmsprop',
                  metrics=['accuracy'])

    print("{:0.3f} seconds.".format(time() - t0))
    return model
```

```
In [7]: with tf.device('cpu:0'):
    model = recurrent_model(vocab_size, maxlen)
```

```
Defining recurrent model
0.550 seconds.
Compiling the model...
0.043 seconds.
```

```
In [8]: print("Training recurrent CPU model...")
t0 = time()
model.fit(X_train_pad, y_train,
           batch_size=1024,
           epochs=2,
           shuffle=True)
print("{:0} seconds.".format(time() - t0))
```

```
Training recurrent CPU model...
Epoch 1/2
25000/25000 [=====] - 11s 436us/step - loss: 0.6274 -
acc: 0.6694
Epoch 2/2
25000/25000 [=====] - 9s 353us/step - loss: 0.4967 -
acc: 0.7871
20.850658893585205 seconds.
```

```
In [9]: with tf.device('gpu:0'):
    model = recurrent_model(vocab_size, maxlen)
```

```
Defining recurrent model
0.539 seconds.
Compiling the model...
0.040 seconds.
```

```
In [10]: print("Training recurrent CPU model...")
t0 = time()
model.fit(X_train_pad, y_train,
           batch_size=1024,
           epochs=2,
           shuffle=True)
print("{:0} seconds.".format(time() - t0))
```

```
Training recurrent CPU model...
Epoch 1/2
25000/25000 [=====] - 5s 185us/step - loss: 0.6211 -
acc: 0.6714
Epoch 2/2
25000/25000 [=====] - 3s 138us/step - loss: 0.4793 -
acc: 0.7878
9.354848384857178 seconds.
```

```
In [11]: NGPU = 2
```

```
In [12]: with tf.device('cpu:0'):
    model = recurrent_model(vocab_size, maxlen)

    model = multi_gpu_model(model, NGPU)

Defining recurrent model
0.358 seconds.
Compiling the model...
0.041 seconds.
```

```
In [13]: model.compile(loss='binary_crossentropy',
                      optimizer='rmsprop',
                      metrics=['accuracy'])
```

```
In [14]: print("Training recurrent GPU model on 2 GPUs...")
t0 = time()
model.fit(X_train_pad, y_train,
           batch_size=1024*NGPU,
           epochs=2,
           shuffle=True)
print("{:0} seconds.".format(time() - t0))
```

```
Training recurrent GPU model on 2 GPUs...
Epoch 1/2
25000/25000 [=====] - 4s 170us/step - loss: 0.6695 -
acc: 0.6125
Epoch 2/2
25000/25000 [=====] - 2s 94us/step - loss: 0.5399 -
acc: 0.7550
8.971489906311035 seconds.
```

## 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.

```
In [15]: import keras.backend as K
```

```
In [16]: K.clear_session() # freeing the graph
```

```
In [17]: model = Sequential()
    with tf.device('gpu:0'):
        model.add(Embedding(input_dim=vocab_size,
                            output_dim=100,
                            input_length=maxlen))
        model.add(LSTM(64, dropout=0.2,
                      recurrent_dropout=0.2,
                      return_sequences=True))
    with tf.device('gpu:1'):
        model.add(LSTM(64, dropout=0.2,
                      recurrent_dropout=0.2,
                      return_sequences=True))
        model.add(LSTM(64, dropout=0.2,
                      recurrent_dropout=0.2))
    with tf.device('cpu:0'):
        model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy',
                  optimizer='rmsprop',
                  metrics=['accuracy'])

    print("{:0.3f} seconds.".format(time() - t0))
```

```
print("Compiling the model...")
t0 = time()
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

print("{:0.3f} seconds.".format(time() - t0))
```

```
10.114 seconds.
Compiling the model...
```

0.041 seconds.

```
In [18]: print("Training distributed recurrent model...")
t0 = time()
model.fit(X_train_pad, y_train,
           batch_size=1024,
           epochs=2,
           shuffle=True)
print("{:0} seconds.".format(time() - t0))
```

```
Training distributed recurrent model...
Epoch 1/2
25000/25000 [=====] - 13s 512us/step - loss: 0.6394 -
acc: 0.6416
Epoch 2/2
25000/25000 [=====] - 10s 416us/step - loss: 0.4968 -
acc: 0.7649
26.747453689575195 seconds.
```

# 24

## Performance Improvement Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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  $64 \times 64$  as input and return a binary output ( $\text{female}=0/\text{male}=1$ )
- 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  $64\times 64$ .
- 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](#)

In [3]: %%bash

```
if [ ! -d ../data/male_female ]; then
    A=https://www.zerotodeeplearning.com/
    B=media/z2dl/45bzty/
    C=male_female.tgz
    wget $A$B$C -O male_female.tgz
    tar -xzvf male_female.tgz --directory ../data/
    rm male_female.tgz
fi
```

In [4]: data\_path = '../data/male\_female/'

In [5]:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import BatchNormalization
from itertools import islice
from keras import backend as K
from keras.utils import multi_gpu_model
from keras.preprocessing.image import ImageDataGenerator
import tensorflow as tf
```

Using TensorFlow backend.

In [6]:

```
def create_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3),
                    input_shape=(64, 64, 3),
                    activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
```

```

model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization())

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
return model

```

In [7]: gpus = K.tensorflow\_backend.\_get\_available\_gpus()  
gpus

Out[7]: ['/job:localhost/replica:0/task:0/device:GPU:0',  
 '/job:localhost/replica:0/task:0/device:GPU:1']

In [8]: NGPU = len(gpus)

In [9]: if NGPU <= 1:  
 model = create\_model()  
 ncopies = 1 # for batch size  
else:  
 with tf.device("/cpu:0"):  
 model = create\_model()  
 model = multi\_gpu\_model(model, gpus=NGPU)  
 ncopies = NGPU

In [10]: model.summary()

---



---

Layer (type)	Output Shape	Param #	Connected to
conv2d_1_input (InputLayer)	(None, 64, 64, 3)	0	
lambda_1 (Lambda)	(None, 64, 64, 3)	0	
conv2d_1_input[0][0]			

---

```
-----  
lambda_2 (Lambda)           (None, 64, 64, 3)    0  
conv2d_1_input[0][0]  
-----  
  
sequential_1 (Sequential)   (None, 1)            352129    lambda_1[0][0]  
                             lambda_2[0][0]  
-----  
  
dense_2 (Concatenate)       (None, 1)            0  
sequential_1[1][0]  
sequential_1[2][0]  
=====  
=====  
Total params: 352,129  
Trainable params: 351,809  
Non-trainable params: 320  
-----  
-----
```

```
In [11]: model.compile(optimizer='adam',  
                      loss='binary_crossentropy',  
                      metrics=['accuracy'])
```

```
In [12]: batch_size = 16
```

```
In [13]: train_gen = ImageDataGenerator(rescale=1./255,  
                                      width_shift_range=0.1,  
                                      height_shift_range=0.1,  
                                      rotation_range=10,  
                                      shear_range=0.2,  
                                      zoom_range=0.2,  
                                      horizontal_flip=True)  
  
test_gen = ImageDataGenerator(rescale=1./255)
```

```
In [14]: train = train_gen.flow_from_directory(  
          data_path + '/train', target_size=(64, 64),  
          batch_size=batch_size * ncopies,  
          class_mode='binary')  
  
test = test_gen.flow_from_directory(  
          data_path + '/test', target_size=(64, 64),  
          batch_size=batch_size * ncopies,  
          class_mode='binary')
```

```
Found 11663 images belonging to 2 classes.  
Found 2920 images belonging to 2 classes.
```

```
In [15]: test.class_indices
```

```
Out[15]: {'0_female': 0, '1_male': 1}
```

```
In [16]: label_to_class = {0: 'female', 1: 'male'}
```

```
In [17]: model.fit_generator(train,  
                           steps_per_epoch=600,  
                           epochs=3);
```

```
Epoch 1/3  
600/600 [=====] - 59s 99ms/step - loss: 0.6250 - acc:  
0.6959  
Epoch 2/3  
600/600 [=====] - 58s 97ms/step - loss: 0.4662 - acc:  
0.7706  
Epoch 3/3  
600/600 [=====] - 58s 96ms/step - loss: 0.4241 - acc:  
0.7938
```

```
In [18]: model.evaluate_generator(test)
```

```
Out[18]: [0.39503416067933383, 0.809931506849315]
```

```
In [19]: X_test = []  
y_test = []  
for ts in islice(test, 50):  
    X_test.append(ts[0])  
    y_test.append(ts[1])  
  
X_test = np.concatenate(X_test)  
y_test = np.concatenate(y_test)
```

```
In [20]: y_test
```

```
Out[20]: array([1., 0., 0., ..., 0., 0., 0.], dtype=float32)
```

```
In [21]: y_pred = model.predict(X_test).ravel().round(0)
y_pred
```

```
Out[21]: array([1., 0., 0., ..., 0., 0., 0.], dtype=float32)
```

```
In [22]: wrong_idx = np.argwhere(y_test != y_pred).ravel()
wrong_idx
```

```
Out[22]: array([ 3,    7,   12,   24,   25,   27,   28,   29,   43,   45,   49,
      52,   54,   57,   76,   80,   89,   90,   99,  101,  102,  109,
     115,  116,  117,  120,  131,  139,  153,  155,  157,  162,  170,
     172,  177,  182,  193,  195,  199,  202,  204,  205,  210,  212,
     213,  216,  225,  226,  228,  236,  245,  246,  255,  260,  267,
     273,  287,  289,  292,  293,  301,  303,  310,  311,  314,  336,
     347,  351,  354,  359,  368,  375,  380,  381,  384,  387,  390,
     402,  407,  408,  409,  411,  431,  442,  446,  449,  462,  464,
     467,  468,  481,  484,  486,  488,  491,  492,  493,  496,  497,
     499,  503,  507,  509,  512,  531,  532,  533,  546,  550,  555,
     564,  566,  570,  571,  580,  584,  604,  606,  607,  608,  616,
     618,  620,  624,  631,  636,  640,  641,  645,  652,  660,  668,
     669,  676,  680,  681,  684,  685,  692,  693,  695,  710,  724,
     726,  730,  732,  740,  743,  747,  789,  797,  803,  806,  811,
     814,  817,  818,  819,  824,  840,  841,  842,  847,  858,  867,
     870,  877,  879,  886,  887,  895,  897,  900,  908,  913,  914,
     919,  921,  935,  942,  943,  945,  949,  956,  959,  961,  965,
     968,  978,  984,  986,  991,  996,  1018,  1020,  1037,  1042,  1085,
    1089,  1093,  1095,  1104,  1105,  1106,  1108,  1111,  1112,  1118,  1120,
    1122,  1139,  1142,  1143,  1150,  1159,  1166,  1167,  1173,  1186,  1191,
    1196,  1197,  1206,  1208,  1210,  1221,  1222,  1223,  1239,  1243,  1244,
    1251,  1254,  1256,  1257,  1258,  1273,  1275,  1277,  1287,  1289,  1305,
    1307,  1313,  1316,  1326,  1333,  1335,  1337,  1340,  1341,  1345,  1348,
    1365,  1372,  1373,  1374,  1377,  1398,  1399,  1407,  1410,  1411,  1417,
    1418,  1429,  1430,  1435,  1436,  1444,  1448,  1450,  1452,  1454,  1455,
    1474,  1480,  1485,  1490,  1497,  1500,  1503,  1525,  1530,  1531,  1533,
    1536,  1540,  1546,  1551,  1566,  1567,  1571,  1580,  1581,  1583,  1585,
    1592])
```

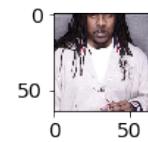
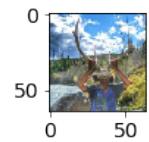
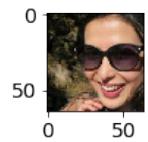
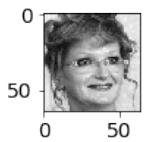
```
In [23]: plt.figure(figsize=(10, 10))
```

```
i = 1

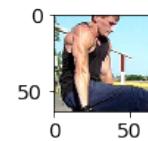
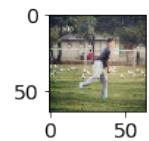
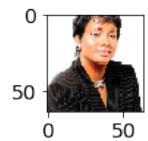
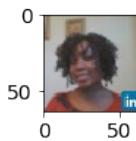
for idx in wrong_idx[:16]:
    plt.subplot(4, 4, i)
    plt.imshow(X_test[idx])
    label = label_to_class[int(y_test[idx])]
```

```
pred = label_to_class[int(y_pred[idx])]  
plt.title("Label: {} Pred: {}".format(label, pred))  
i += 1  
  
plt.tight_layout()
```

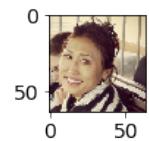
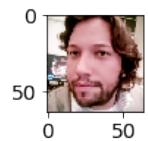
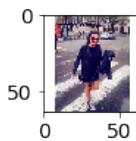
Label: female Pred: female Label: female Pred: female Label: male Pred: female



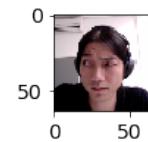
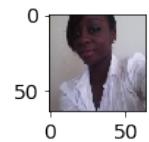
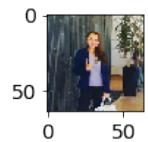
Label: female Pred: male Label: female Pred: male Label: male Pred: female



Label: female Pred: male Label: male Pred: female Label: female Pred: male Label: male Pred: female



Label: male Pred: female Label: female Pred: male Label: female Pred: male Label: male Pred: female



The model still has a lot to learn about humans



# 25

## Pretrained Models for Images Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### 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](#).)

```
In [3]: from urllib.request import urlretrieve  
        from keras.preprocessing import image  
        from keras.applications.xception import Xception  
        from keras.applications.xception import preprocess_input  
        from keras.applications.xception import decode_predictions
```

Using TensorFlow backend.

```
In [4]: img_url = ('https://upload.wikimedia.org/wikipedia/' +
    'commons/thumb/7/73/Elbilfestival_i_Geiran' +
    'ger_two_Tesla_Model_S_electric_cars.jpg/' +
    '1920px-Elbilfestival_i_Geiranger_two_Tes' +
    'la_Model_S_electric_cars.jpg')
```

```
In [5]: def load_image_from_url(url, target_size=(299, 299)):
    path, response = urlretrieve(
        img_url, filename='/tmp/temp_img.jpg')

    img = image.load_img(path, target_size=target_size)

    img_tensor = np.expand_dims(
        image.img_to_array(img), axis=0)

    return img, img_tensor
```

```
In [6]: img, img_tensor = load_image_from_url(img_url)
```

```
In [7]: img_scaled = preprocess_input(np.copy(img_tensor))
```

```
In [8]: img
```

```
Out[8]:
```



```
In [9]: model = Xception(weights='imagenet')
```

```
In [10]: preds = model.predict(img_scaled)
```

```
In [11]: decode_predictions(preds, top=3)[0]
```

```
Out[11]: [('n03100240', 'convertible', 0.32023576),  
          ('n04285008', 'sports_car', 0.16641538),  
          ('n03459775', 'grille', 0.08980309)]
```

## 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?

```
In [12]: from keras.applications.vgg16 import VGG16  
from keras.applications.vgg16 import preprocess_input  
from keras.applications.vgg16 import decode_predictions
```

```
In [13]: model = VGG16(weights='imagenet')

In [14]: img, img_tensor = load_image_from_url(
           img_url, target_size=(224, 224))

In [15]: img_scaled = preprocess_input(np.copy(img_tensor))

In [16]: preds = model.predict(img_scaled)

In [17]: decode_predictions(preds, top=3)[0]

Out[17]: [('n03594945', 'jeep', 0.19831586),
           ('n03770679', 'minivan', 0.15666518),
           ('n03100240', 'convertible', 0.111178935)]
```

### 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.

```
In [18]: from keras.models import Model
        from keras.layers import Dense, Dropout
        from keras.preprocessing.image import ImageDataGenerator
        from keras.applications.xception import preprocess_input
        from keras.optimizers import SGD

In [19]: img_size = 299

In [20]: base_model = Xception(include_top=False, weights='imagenet',
                           input_shape=(img_size, img_size, 3),
                           pooling='avg')

In [21]: x = base_model.output
        x = Dense(256, activation='relu')(x)
        x = Dropout(0.5)(x)
        predictions = Dense(3, activation='softmax')(x)

In [22]: model = Model(inputs=base_model.input,
                     outputs=predictions)
```

```
In [23]: for layer in base_model.layers:  
    layer.trainable = False  
  
In [24]: model.compile(optimizer='rmsprop',  
                      loss='categorical_crossentropy',  
                      metrics=['accuracy'])  
  
In [25]: train_datagen = ImageDataGenerator(  
         preprocessing_function=preprocess_input,  
         rotation_range=15,  
         width_shift_range=0.2,  
         height_shift_range=0.2,  
         shear_range=5,  
         zoom_range=0.2,  
         horizontal_flip=True,  
         fill_mode='nearest')  
  
In [26]: batch_size = 32
```

```
In [27]: train_path = '../data/sports/train/'
```

```
In [28]: 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.

```
In [29]: model.fit_generator(  
         train_generator,  
         steps_per_epoch=65,  
         epochs=1)
```

```
Epoch 1/1  
65/65 [=====] - 47s 730ms/step - loss: 0.5013 - acc:  
0.8046
```

```
Out[29]: <keras.callbacks.History at 0x7ff4f8725898>
```

```
In [30]: for i, layer in enumerate(base_model.layers):
    print(i, layer.name)

0 input_3
1 block1_conv1
2 block1_conv1_bn
3 block1_conv1_act
4 block1_conv2
5 block1_conv2_bn
6 block1_conv2_act
7 block2_sepconv1
8 block2_sepconv1_bn
9 block2_sepconv2_act
10 block2_sepconv2
11 block2_sepconv2_bn
12 conv2d_5
13 block2_pool
14 batch_normalization_5
15 add_13
16 block3_sepconv1_act
17 block3_sepconv1
18 block3_sepconv1_bn
19 block3_sepconv2_act
20 block3_sepconv2
21 block3_sepconv2_bn
22 conv2d_6
23 block3_pool
24 batch_normalization_6
25 add_14
26 block4_sepconv1_act
27 block4_sepconv1
28 block4_sepconv1_bn
29 block4_sepconv2_act
30 block4_sepconv2
31 block4_sepconv2_bn
32 conv2d_7
33 block4_pool
34 batch_normalization_7
35 add_15
36 block5_sepconv1_act
37 block5_sepconv1
38 block5_sepconv1_bn
39 block5_sepconv2_act
40 block5_sepconv2
41 block5_sepconv2_bn
42 block5_sepconv3_act
43 block5_sepconv3
44 block5_sepconv3_bn
45 add_16
46 block6_sepconv1_act
47 block6_sepconv1
48 block6_sepconv1_bn
49 block6_sepconv2_act
50 block6_sepconv2
51 block6_sepconv2_bn
52 block6_sepconv3_act
53 block6_sepconv3
54 block6_sepconv3_bn
```

```
55 add_17
56 block7_sepconv1_act
57 block7_sepconv1
58 block7_sepconv1_bn
59 block7_sepconv2_act
60 block7_sepconv2
61 block7_sepconv2_bn
62 block7_sepconv3_act
63 block7_sepconv3
64 block7_sepconv3_bn
65 add_18
66 block8_sepconv1_act
67 block8_sepconv1
68 block8_sepconv1_bn
69 block8_sepconv2_act
70 block8_sepconv2
71 block8_sepconv2_bn
72 block8_sepconv3_act
73 block8_sepconv3
74 block8_sepconv3_bn
75 add_19
76 block9_sepconv1_act
77 block9_sepconv1
78 block9_sepconv1_bn
79 block9_sepconv2_act
80 block9_sepconv2
81 block9_sepconv2_bn
82 block9_sepconv3_act
83 block9_sepconv3
84 block9_sepconv3_bn
85 add_20
86 block10_sepconv1_act
87 block10_sepconv1
88 block10_sepconv1_bn
89 block10_sepconv2_act
90 block10_sepconv2
91 block10_sepconv2_bn
92 block10_sepconv3_act
93 block10_sepconv3
94 block10_sepconv3_bn
95 add_21
96 block11_sepconv1_act
97 block11_sepconv1
98 block11_sepconv1_bn
99 block11_sepconv2_act
100 block11_sepconv2
101 block11_sepconv2_bn
102 block11_sepconv3_act
103 block11_sepconv3
104 block11_sepconv3_bn
105 add_22
106 block12_sepconv1_act
107 block12_sepconv1
108 block12_sepconv1_bn
109 block12_sepconv2_act
110 block12_sepconv2
111 block12_sepconv2_bn
112 block12_sepconv3_act
113 block12_sepconv3
```

```

114 block12_sepconv3_bn
115 add_23
116 block13_sepconv1_act
117 block13_sepconv1
118 block13_sepconv1_bn
119 block13_sepconv2_act
120 block13_sepconv2
121 block13_sepconv2_bn
122 conv2d_8
123 block13_pool
124 batch_normalization_8
125 add_24
126 block14_sepconv1
127 block14_sepconv1_bn
128 block14_sepconv1_act
129 block14_sepconv2
130 block14_sepconv2_bn
131 block14_sepconv2_act
132 global_average_pooling2d_1

```

In [31]: # we chose to train the top 2 separable convolution  
# blocks, i.e. we will freeze the first 126 layers  
# and unfreeze the rest:

```

split_layer = 126

for layer in model.layers[:split_layer]:
    layer.trainable = False
for layer in model.layers[split_layer:]:
    layer.trainable = True

```

In [32]: model.compile(optimizer=SGD(lr=0.0001, momentum=0.9),  
loss='categorical\_crossentropy',  
metrics=['accuracy'])

In [33]: model.fit\_generator(  
train\_generator,  
steps\_per\_epoch=65,  
epochs=1);

```

Epoch 1/1
65/65 [=====] - 47s 730ms/step - loss: 0.2432 - acc: 0.9110

```

# 26

## Pretrained Embeddings for Text Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### Exercise 1

Compare the representations of Word2Vec, Glove and FastText. In the `data/embeddings` folder we provided you with two additional scripts to download FastText and Word2Vec. Go ahead and download each of them into the `data/embeddings`. Then load each of the 3 embeddings in a separate Gensim model and complete the following steps:

1. define a list of words containing the following words: ‘good’, ‘bad’, ‘fast’, ‘tensor’, ‘teacher’, ‘student’.
- create a function called `get_top_5(words, model)` that retrieves the top 5 most similar words to the list of words and compare what the 3 different embeddings give you
- apply the same function to each word in the list separately and compare the lists of the 3 embeddings.
- explore the following word analogies:

man:king=woman:? ==> expected queen  
france:paris=germany:? ==> expected berlin  
teacher:teach=student:? ==> expected learn  
cat:kitten=dog:? ==> expected puppy  
english:friday=italiano:? ==> expected venerdi

Can word analogies be used for translation?

Note that loading the vector may take several minutes depending on your computer.

```
In [3]: import gensim
```

```
In [4]: from gensim.models import KeyedVectors
```

```
In [5]: w2v_path = '../data/embeddings/GoogleNews-vectors-negative300.bin'
w2v_model = KeyedVectors.load_word2vec_format(
    w2v_path, binary=True)
```

```
In [6]: glove_path = '../data/embeddings/glove.6B.50d.txt.vec'
glove_model = KeyedVectors.load_word2vec_format(
    glove_path, binary=False)
```

```
In [7]: fasttext_path = '../data/embeddings/wiki-news-300d-1M.vec'
fasttext_model = KeyedVectors.load_word2vec_format(
    fasttext_path, binary=False)
```

```
In [8]: word_list = ['good', 'bad',
                  'fast', 'tensor',
                  'teacher', 'student']
```

```
In [9]: def get_top_5(words, model):
    res = model.most_similar(positive=words, topn=5)
    return [r[0] for r in res]
```

```
In [10]: for word in word_list:
    print(word)
    print("W2V      : ", get_top_5([word], w2v_model))
    print("Glove    : ", get_top_5([word.lower()], glove_model))
    print("FastText: ", get_top_5([word], fasttext_model))
    print()
```

```
good
W2V      : ['great', 'bad', 'terrific', 'decent', 'nice']
Glove    : ['better', 'really', 'always', 'sure', 'something']
FastText: ['bad', 'excellent', 'decent', 'nice', 'great']
```

```
bad
W2V      : ['good', 'terrible', 'horrible', 'Bad', 'lousy']
```

```

Glove   : ['worse', 'unfortunately', 'too', 'really', 'little']
FastText: ['good', 'terrible', 'horrible', 'lousy', 'awful']

fast
W2V     : ['quick', 'rapidly', 'Fast', 'quickly', 'slow']
Glove   : ['slow', 'faster', 'pace', 'turning', 'better']
FastText: ['slow', 'rapid', 'quick', 'Fast', 'faster']

tensor
W2V     : ['uniaxial', ' $\tau$ ', ' $\theta$ ', ' $\varphi$ ', 'wavefunction']
Glove   : ['scalar', 'tensors', 'coefficients', 'coefficient', 'formula_12']
FastText: ['tensors', 'Tensor', 'stress-energy', 'pseudotensor', 'tensorial']

teacher
W2V     : ['teachers', 'Teacher', 'guidance_counselor', 'elementary',
'PE_teacher']
Glove   : ['student', 'graduate', 'teaching', 'taught', 'teaches']
FastText: ['teachers', 'educator', 'Teacher', 'student', 'pupil']

student
W2V     : ['students', 'Student', 'teacher', 'stu_dent', 'faculty']
Glove   : ['teacher', 'students', 'teachers', 'graduate', 'school']
FastText: ['students', 'teacher', 'Student', 'university', 'graduate']

```

```
In [11]: def word_analogy(model,
                         thing='man',
                         is_to='king',
                         like='woman'):
    res = model.most_similar(positive=[is_to, like],
                             negative=[thing],
                             topn=3)
    return [r[0] for r in res]
```

```
In [12]: word_analogies = ['man:king=woman:queen',
                           'france:paris=germany:berlin',
                           'teacher:teach=student:learn',
                           'cat:kitten=dog?:?',
                           'english:friday=italiano?:?']

for analogy in word_analogies:
    first, second = analogy.split('=')
    thing, is_to = first.split(':')
    like, answer = second.split(':')

    print(analogy)
    print("W2V     : ", word_analogy(
        w2v_model, thing, is_to, like))
    print("Glove   : ", word_analogy(
        glove_model, thing, is_to, like))
```

```

print("FastText: ", word_analogy(
    fasttext_model, thing, is_to, like))
print()

man:king=woman:queen
W2V     : ['queen', 'monarch', 'princess']
Glove   : ['queen', 'throne', 'prince']
FastText: ['queen', 'monarch', 'princess']

france:paris=germany:berlin
W2V     : ['berlin', 'german', 'lindsay_lohan']
Glove   : ['berlin', 'frankfurt', 'vienna']
FastText: ['berlin', 'munich', 'dresden']

teacher:teach=student:learn
W2V     : ['educate', 'learn', 'teaches']
Glove   : ['students', 'teachers', 'teaching']
FastText: ['learn', 'educate', 'attend']

cat:kitten=dog:?
W2V     : ['puppy', 'pup', 'pit_bull']
Glove   : ['puppy', 'rottweiler', 'retriever']
FastText: ['puppy', 'puppies', 'pup']

english:friday=italiano:?
W2V     : ['noche', 'fatto', 'la_versione']
Glove   : ['exxonmobil', 'eni', 'newmont']
FastText: ['dopo', 'meglio', 'lavoro']

```

## Exercise 2

The [Reuters Newswire topic classification dataset](#) is a dataset of 11,228 newswires from Reuters, labeled over 46 topics. This dataset is provided in the `keras.datasets` module and it's easy to use.

Let's compare the performance of a model using pre-trained embeddings with a model using random embeddings on the topic classification task.

- Load the data from `keras.datasets.reuters`
- Retrieve the word index and create the `reverse_word_idx` as it was done for IMDB in [Chapter 8](#).
- Augment the reverse word index with `pad_char`, `start_char` and `oov_char` at indices 0, 1, 2 respectively.
- Check the maximum length of a newswire and use the `pad_sequences` function to pad everything to that 100 words.
- Create and train two models, one using pre-trained embeddings and the other using a randomly initialized embedding
- Compare their performance on this dataset using a recurrent model. In particular check which of the two models shows the worst overfitting.

```
In [13]: from keras.datasets import reuters
```

Using TensorFlow backend.

```
In [14]: vocab_size=20000
```

```
In [15]: (X_train, y_train), (X_test, y_test) = \  
        reuters.load_data(num_words=vocab_size, index_from=2)
```

```
In [16]: reuters_word_idx = reuters.get_word_index()
```

```
In [17]: reverse_reuters_word_idx = {index+2:word for word, index  
                                    in reuters_word_idx.items()}
```

```
In [18]: reverse_reuters_word_idx[0] = 'pad_char'  
        reverse_reuters_word_idx[1] = 'start_char'  
        reverse_reuters_word_idx[2] = 'oov_char'
```

```
In [19]: from keras.preprocessing.sequence import pad_sequences
```

```
In [20]: ' '.join([reverse_reuters_word_idx[i] for i in X_train[0]])
```

```
Out[20]: 'start_char oov_char oov_char said as a result of its december acquisition of space co
```

```
In [21]: max([len(seq) for seq in X_train])
```

```
Out[21]: 2376
```

```
In [22]: maxlen=100
```

```
In [23]: X_train_pad = pad_sequences(X_train, maxlen=maxlen)  
        X_test_pad = pad_sequences(X_test, maxlen=maxlen)
```

```
In [24]: embedding_model = fasttext_model
```

```
In [25]: embedding_size = embedding_model.vector_size
```

```
In [26]: reuters_emb_weights = np.zeros((vocab_size, embedding_size))
```

```
for i in range(1, vocab_size):
    word = reverse_reuters_word_idx[i]
    try:
        reuters_emb_weights[i] = embedding_model[word]
    except:
        # print(word, "not found in pre-trained embedding")
        reuters_emb_weights[i] = np.random.random(
            size=embedding_size)
    pass
```

```
In [27]: from keras.layers import LSTM, Embedding, Dense
from keras.models import Sequential
```

```
In [28]: random_emb_layer = Embedding(vocab_size,
                                      embedding_size,
                                      mask_zero=True,
                                      input_length=maxlen)

fixed_emb_layer = Embedding(vocab_size,
                            embedding_size,
                            weights=[reuters_emb_weights],
                            mask_zero=True,
                            trainable=False,
                            input_length=maxlen)
```

```
In [29]: def build_train_eval(embedding_layer):
    model = Sequential([
        embedding_layer,
        LSTM(64, dropout=0.2, recurrent_dropout=0.2),
        Dense(46, activation='softmax')
    ])

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

    h = model.fit(X_train_pad, y_train,
                  batch_size=32,
                  epochs=5,
                  validation_split=0.1)

    train_loss, train_acc = model.evaluate(X_train_pad,
                                           y_train)
```

```
print('Train loss:', train_loss)
print('Train accuracy:', train_acc)

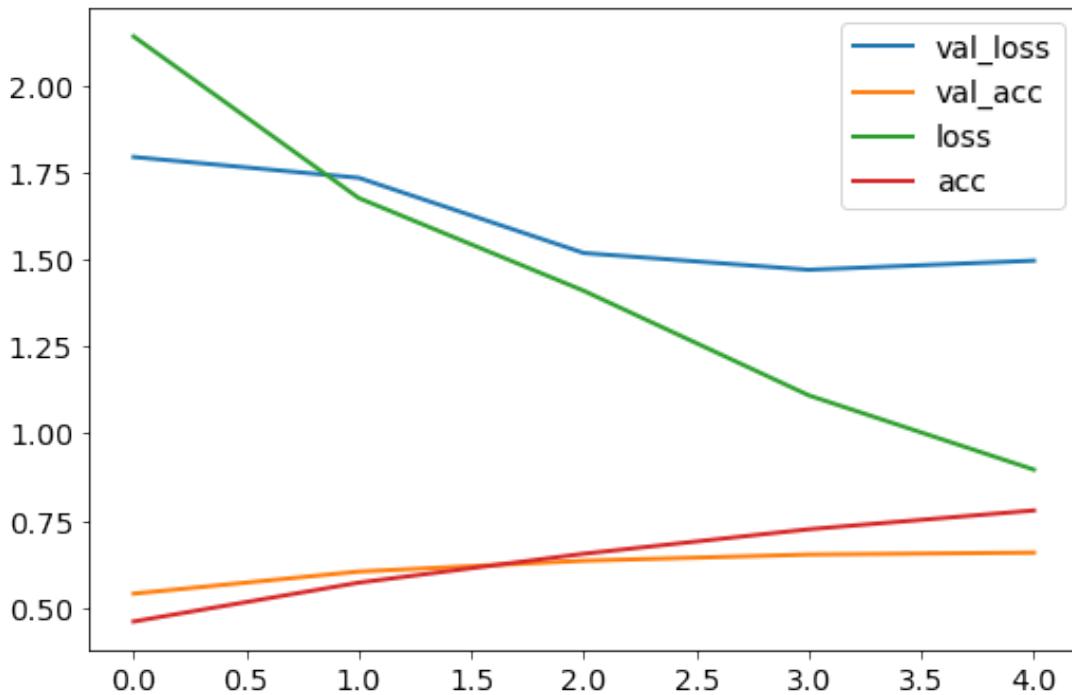
test_loss, test_acc = model.evaluate(X_test_pad,
                                      y_test)
print('Test loss:', test_loss)
print('Test accuracy:', test_acc)

return h, model
```

```
In [30]: history, random_model = build_train_eval(random_emb_layer)
```

```
Train on 8083 samples, validate on 899 samples
Epoch 1/5
8083/8083 [=====] - 55s 7ms/step - loss: 2.1399 - acc: 0.4595 - val_loss: 1.7936 - val_acc: 0.5395
Epoch 2/5
8083/8083 [=====] - 52s 6ms/step - loss: 1.6758 - acc: 0.5711 - val_loss: 1.7338 - val_acc: 0.6029
Epoch 3/5
8083/8083 [=====] - 52s 6ms/step - loss: 1.4093 - acc: 0.6541 - val_loss: 1.5178 - val_acc: 0.6340
Epoch 4/5
8083/8083 [=====] - 52s 6ms/step - loss: 1.1089 - acc: 0.7245 - val_loss: 1.4696 - val_acc: 0.6518
Epoch 5/5
8083/8083 [=====] - 52s 6ms/step - loss: 0.8956 - acc: 0.7784 - val_loss: 1.4959 - val_acc: 0.6574
8982/8982 [=====] - 18s 2ms/step
Train loss: 0.7549172357683642
Train accuracy: 0.8184146069386732
2246/2246 [=====] - 5s 2ms/step
Test loss: 1.51039785576631
Test accuracy: 0.6380231522707035
```

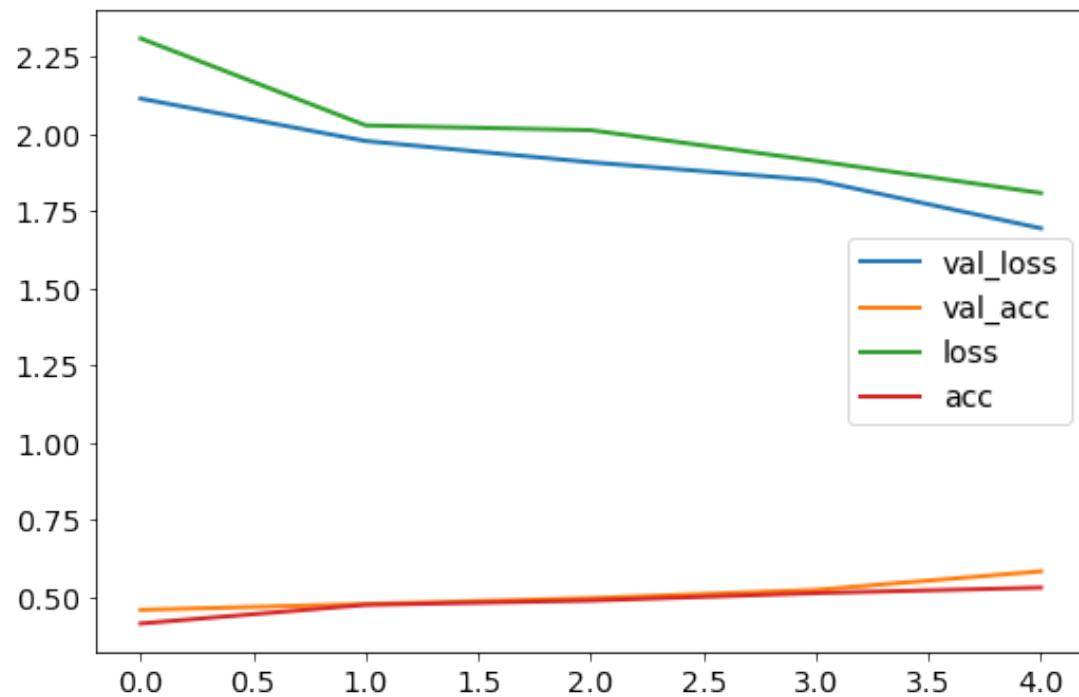
```
In [31]: pd.DataFrame(history.history).plot();
```



```
In [32]: history, fixed_model = build_train_eval(fixed_emb_layer)
```

```
Train on 8083 samples, validate on 899 samples
Epoch 1/5
8083/8083 [=====] - 48s 6ms/step - loss: 2.3074 - acc: 0.4153 - val_loss: 2.1127 - val_acc: 0.4594
Epoch 2/5
8083/8083 [=====] - 47s 6ms/step - loss: 2.0262 - acc: 0.4756 - val_loss: 1.9756 - val_acc: 0.4783
Epoch 3/5
8083/8083 [=====] - 47s 6ms/step - loss: 2.0106 - acc: 0.4895 - val_loss: 1.9068 - val_acc: 0.4972
Epoch 4/5
8083/8083 [=====] - 47s 6ms/step - loss: 1.9115 - acc: 0.5137 - val_loss: 1.8489 - val_acc: 0.5239
Epoch 5/5
8083/8083 [=====] - 47s 6ms/step - loss: 1.8075 - acc: 0.5312 - val_loss: 1.6934 - val_acc: 0.5840
8982/8982 [=====] - 18s 2ms/step
Train loss: 1.650323630308314
Train accuracy: 0.5871743487106668
2246/2246 [=====] - 5s 2ms/step
Test loss: 1.695558883627928
Test accuracy: 0.570792520062157
```

```
In [33]: pd.DataFrame(history.history).plot();
```





# 27

## Serving Deep Learning Models Exercises Solutions

```
In [1]: with open('../course/common.py') as fin:  
    exec(fin.read())
```

```
In [2]: with open('../course/matplotlibconf.py') as fin:  
    exec(fin.read())
```

### Exercise 1

Let's deploy an image recognition API using Tensorflow Serving. The main difference from the API we have deployed in this chapter is that we will have to deal with how to pass an image to the model through tensorflow serving. Since this chapter focuses on deployment, we will take a shortcut and deploy a pre-trained model that uses Imagenet. In particular we will deploy the Xception model. If you are unsure about how to use pre-trained model, please go back to [Chapter 11](#) for a refresher.

Here are the steps you will need to complete:

- load the model in keras
- export the model for tensorflow serving:
  - set the learning phase to zero
  - choose the right inputs and outputs
  - choose the right prediction signature
- run the model server (using Docker or using the native server)
- test your api on the provided image `./13_penguin.jpg`:

- load 13\_penguin.jpg using the keras.preprocessing.image module
- pre-processes the image with the appropriate function
- convert the image to a numpy array
- serialize the array to protobuf
- send the image to the server using a PredictRequest and a PredictionServiceStub
- retrieve the prediction and convert it back to a numpy array
- decode the prediction with keras decode\_predictions function

The response should be: 'king\_penguin'.

```
In [3]: import os
        from os.path import join
        import shutil

        import tensorflow as tf
        import keras.backend as K

        from keras.preprocessing import image
        from keras.applications.xception import Xception
        from keras.applications.xception import preprocess_input
        from keras.applications.xception import decode_predictions

        from grpc import insecure_channel

        from tensorflow.python.saved_model.builder \
            import SavedModelBuilder
        from tensorflow.python.saved_model.signature_def_utils \
            import predict_signature_def
        from tensorflow_serving.apis.prediction_service_pb2_grpc \
            import PredictionServiceStub
        from tensorflow_serving.apis.predict_pb2 \
            import PredictRequest
        from tensorflow.contrib.util import make_tensor_proto
        from tensorflow.contrib.util import make_ndarray
```

Using TensorFlow backend.

### Save Xception as tensorflow model

```
In [4]: model = Xception(weights='imagenet')
```

```
In [5]: K.set_learning_phase(0)
```

```
In [6]: base_path = '/tmp/ztdl_models/xception'
        sub_path = 'tf-serving'
        version = 1

In [7]: export_path = join(base_path, sub_path, str(version))
        shutil.rmtree(export_path, ignore_errors=True)

In [8]: builder = SavedModelBuilder(export_path)

In [9]: signature = predict_signature_def(
        inputs={"inputs": model.input},
        outputs={"outputs": model.output})

In [10]: sess = K.get_session()

In [11]: builder.add_meta_graph_and_variables(
        sess=sess,
        tags=[tf.saved_model.tag_constants.SERVING],
        signature_def_map={'predict': signature})
```

INFO:tensorflow:No assets to save.  
INFO:tensorflow:No assets to write.

```
In [12]: builder.save()
```

INFO:tensorflow:SavedModel written to:  
/tmp/ztdl\_models/xception/tfserving/1/saved\_model.pb

```
Out[12]: b'/tmp/ztdl_models/xception/tfserving/1/saved_model.pb'
```

## Start Server

```
docker run \
-v /tmp/ztdl_models/xception/tfserving:/models/xception \
-e MODEL_NAME=xception \
-e MODEL_PATH=/models/xception \
-p 8502:8500 \
-p 8503:8501 \
-t tensorflow/serving
```

**Convert image to protobuf**

```
In [13]: img = image.load_img(  
    './13_penguin.jpg', target_size=(299, 299))  
  
In [14]: img_tensor = np.expand_dims(  
    image.img_to_array(img), axis=0)  
  
In [15]: img_scaled = preprocess_input(img_tensor)  
  
In [16]: data_pb = make_tensor_proto(  
    img_scaled, dtype='float', shape=img_scaled.shape)
```

**Send request and retrieve response**

```
In [17]: channel = insecure_channel('localhost:8502')  
  
In [18]: stub = PredictionServiceStub(channel)  
  
In [19]: request = PredictRequest()  
  
In [20]: request.model_spec.name = 'xception'  
  
In [21]: request.model_spec.signature_name = 'predict'  
  
In [22]: request.inputs['inputs'].CopyFrom(data_pb)  
  
In [23]: result_future = stub.Predict.future(request, 5.0)  
  
In [24]: result = result_future.result()
```

**Decode predictions**

```
In [25]: scores = make_ndarray(result.outputs['outputs'])  
  
In [26]: preds = decode_predictions(scores, top=1)[0][0][1]
```

```
In [27]: preds
```

```
Out[27]: 'king_penguin'
```

## Exercise 2

The above method of serving a pre-trained model has an issue: we are doing pre-processing and prediction decoding on the client side. This is actually not a best practice, because it requires the client to be aware of what kind of pre-processing and decoding functions the model needs.

We would like a server that takes the image as it is and returns a string with the name of the object in the image.

The easy way to do this is to use the Flask app implementation we have shown in this chapter and move pre-processing and decoding on the server side.

Go ahead and build a Flask version of the API that takes an image url as a json string, applies pre-processing, runs and decodes the prediction and returns a string with the response.

You will not use tensorflow serving for this exercise.

Once your script is ready, save it as `13_flask_serve_xception.py`, run it as:

```
python 13_flask_serve_xception.py
```

and test the prediction with the following command:

```
curl -d 'http://bit.ly/2wb7uqN' \
      -H "Content-Type: application/json" \
      -X POST http://localhost:5000
```

If you've done things correctly, this should return:

```
"king_penguin"
```

**Disclaimer: this script is not meant for production purposes. Retrieving a file from a URL is not secure and you should avoid building an API that retrieves a file from a URL provided from the client. Here we used the url retrieval trick in order to make the curl command shorter.**

```
In [28]: !cat 13_flask_serve_xception.py
```

```
import os
import json
import numpy as np

from flask import Flask
from flask import request, jsonify

import tensorflow as tf
from urllib.request import urlretrieve
from keras.preprocessing import image
from keras.applications.xception import Xception
from keras.applications.xception import preprocess_input
from keras.applications.xception import decode_predictions
import keras.backend as K

loaded_model = None
graph = None

app = Flask(__name__)

def load_model():
    """
    Load model and tensorflow graph
    into global variables.
    """
    # global variables
    global loaded_model
    global graph

    loaded_model = Xception(weights='imagenet')

    # get the tensorflow graph
    graph = tf.get_default_graph()
    print("Model loaded.")

def load_image_from_url(url, target_size=(299, 299)):
    path, response = urlretrieve(url, filename='/tmp/temp_img')
    img = image.load_img(path, target_size=target_size)
    img_tensor = np.expand_dims(image.img_to_array(img), axis=0)
    return img, img_tensor

def preprocess(data):
    url = data.decode('utf-8')
    img, img_tensor = load_image_from_url(url)
    img_scaled = preprocess_input(img_tensor)
    return img_scaled

@app.route('/', methods=["POST"])
def predict():
    """
    Generate predictions with the model
    when receiving data as a POST request
    """
    if request.method == "POST":
        # get url from the request
        data = request.data
```

```
# preprocess the data
processed = preprocess(data)

# run predictions using the global tf graph
with graph.as_default():
    preds = loaded_model.predict(processed)

# obtain predicted classes from predicted probabilities
result = decode_predictions(preds, top=1)[0][0][1]

# print in backend
print("Received data:", data)
print("Predicted labels:", result)

return jsonify(result)

if __name__ == "__main__":
    print("* Loading model and starting Flask server...")
    load_model()
    app.run(host='0.0.0.0', debug=True)

# Test this with the following command:
# curl -d 'http://bit.ly/2wb7uqN' -H "Content-Type: application/json" -X POST
# http://localhost:5000
```