

Hands-on Machine Learning with Scikit-Learn & TensorFlow

CONCEPTS, TOOLS, AND TECHNIQUES
TO BUILD INTELLIGENT SYSTEMS



Early Release

RAW & UNEDITED

Aurélien Géron

Hands-on Machine Learning

with Scikit-Learn & TensorFlow

Aurélien Géron

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Hands-On Machine Learning with Scikit-Learn and TensorFlow

by Aurélien Géron

Copyright © 2016 Aurélien Géron. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Nicole Tache

Production Editor: FILL IN PRODUCTION EDITOR

TOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition

2016-07-15: First Early Release

2016-08-12: Second Early Release

2016-09-09: Third Early Release

2016-09-16: Fourth Early Release

2016-09-26: Fifth Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491962220> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Hands-On Machine Learning with Scikit-Learn and TensorFlow, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-96222-0

[FILL IN]

Table of Contents

Preface.....	xı
--------------	----

Part I. The Fundamentals of Machine Learning

1. The Machine Learning landscape.....	21
What is Machine Learning?	22
Why use Machine Learning?	22
Types of Machine Learning systems	25
Supervised/Unsupervised learning	26
Batch and Online learning	33
Instance based <i>vs</i> Model based learning	36
Main challenges of Machine Learning	43
Insufficient quantity of training data	43
Non representative training data	45
Poor quality data	46
Irrelevant features	46
Overfitting the training data	47
Underfitting the training data	49
Stepping back	50
Testing and validating	50
Exercises	52
2. End-to-end Machine Learning project.....	55
Working with real data	55
Look at the big picture	57
Frame the problem	58
Select a performance measure	59

Check the assumptions	62
Get the data	63
Create the workspace	63
Download the data	66
Take a quick look at the data structure	67
Create a test set	71
Discover and visualize the data to gain insights	75
Visualizing geographical data	76
Looking for correlations	78
Experimenting with attribute combinations	81
Prepare the data for Machine Learning algorithms	82
Data cleaning	83
Handling text and categorical attributes	85
Custom transformers	87
Feature scaling	88
Transformation pipelines	89
Select and train a model	91
Training and evaluating on the training set	91
Better evaluation using cross validation	92
Fine-tune your model	94
Grid search	94
Randomized search	97
Ensemble methods	97
Analyze the best models and their errors	97
Evaluate your system on the test set	98
Launch, monitor and maintain your system	99
Try it out!	100
Exercises	100
3. Classification.....	101
MNIST	101
Training a binary classifier	104
Performance measures	105
Measuring accuracy using cross-validation	105
Confusion matrix	106
Precision and recall	108
Precision/Recall tradeoff	109
The ROC curve	113
Multiclass classification	116
Error analysis	119
Multilabel classification	123
Multioutput classification	124

Exercises	126
4. Training linear models.....	129
Linear Regression	130
The Normal Equation	131
Computational complexity	134
Gradient Descent	134
Batch Gradient Descent	138
Stochastic Gradient Descent	141
Mini-batch Gradient Descent	143
Polynomial regression	145
Learning curves	147
Regularized linear models	151
Ridge regression	152
Lasso regression	154
Elastic Net	157
Early stopping	157
Logistic Regression	159
Estimating probabilities	159
Training and cost function	160
Decision boundaries	161
Softmax regression	163
Exercises	167
5. Support Vector Machines.....	169
Linear SVM classification	169
Soft margin classification	170
Non-linear SVM classification	173
Polynomial kernel	174
Adding similarity features	175
Gaussian RBF kernel	176
Computational complexity	178
SVM regression	178
Under the hood	180
Decision function and predictions	181
Training objective	182
Quadratic Programming	183
The dual problem	185
Kernelized SVM	186
Online SVMs	188
Exercises	190

6. Decision Trees.....	191
Training and visualizing a Decision Tree	191
Making predictions	193
Estimating class probabilities	195
The CART training algorithm	195
Computational complexity	196
Gini impurity or entropy?	196
Regularization hyperparameters	197
Regression	199
Instability	201
Exercises	202
7. Ensemble Learning and Random Forests.....	205
Voting Classifiers	206
Bagging and Pasting	208
Bagging and Pasting in Scikit-Learn	210
Out-of-bag evaluation	211
Random Patches and Random Subspaces	212
Random Forests	213
Extra-Trees	214
Feature importance	214
Boosting	215
AdaBoost	215
Gradient Boosting	219
Stacking	224
Exercises	227
8. Dimensionality Reduction.....	229
The curse of dimensionality	230
Main approaches for dimensionality reduction	231
Projection	231
Manifold learning	235
PCA	236
Preserving the variance	236
Principal Components	237
Projecting down to d dimensions	238
Using Scikit-Learn	239
Explained variance ratio	239
Choosing the right number of dimensions	240
PCA for compression	240
Incremental PCA	241
RandomizedPCA	242

Kernel PCA	242
Selecting a kernel and tuning hyperparameters	243
LLE	246
Other dimensionality reduction techniques	248
Exercises	249

Part II. Neural Networks and Deep Learning

9. Up and running with TensorFlow.....	253
Installation	256
Creating your first graph and running it in a session	256
Managing graphs	258
Lifecycle of a node value	259
Linear Regression with TensorFlow	260
Implementing Gradient Descent	261
Manually computing the gradients	261
Using autodiff	262
Using an optimizer	263
Feeding data to the training algorithm	263
Saving and restoring models	265
Visualizing the graph and training curves using TensorBoard	266
Name scopes	269
Modularity	270
Sharing variables	273
Exercises	276
10. Introduction to Artificial Neural Networks.....	279
From biological to artificial neurons	280
Biological neurons	281
Logical computations with neurons	283
The Perceptron	284
Multi-Layer Perceptron and Backpropagation	288
Training an FNN with TensorFlow's high level API	291
Training a DNN using plain TensorFlow	292
Construction phase	292
Execution phase	296
Using the neural network	297
Fine-tuning neural network hyperparameters	298
Number of hidden layers	298
Number of neurons per hidden layer	299
Activation functions	300

Other popular ANN architectures	300
Hopfield Nets	300
Boltzmann Machines	302
Restricted Boltzmann Machines	304
Deep Belief Nets	305
Self-Organizing Maps	307
Exercises	309
11. Deep Learning.....	311
Vanishing/exploding gradients problems	311
Xavier and He initialization	313
Non-saturating activation functions	314
Batch Normalization	318
Gradient clipping	322
Reusing pretrained layers	323
Reusing a TensorFlow model	324
Reusing models from other frameworks	325
Freezing the lower layers	326
Caching the frozen layers	326
Tweaking, dropping or replacing the upper layers	327
Model zoos	327
Unsupervised pretraining	328
Pretraining on an auxiliary task	329
Faster optimizers	330
Momentum optimization	330
Nesterov Momentum optimization	332
AdaGrad	333
RMSProp	335
Adam optimization	335
Learning rate scheduling	337
Avoiding overfitting through regularization	339
Early Stopping	340
ℓ_1 and ℓ_2 regularization	341
Dropout	342
Max-norm regularization	345
Data augmentation	347
Practical guidelines	348
Exercises	349
12. Distributing TensorFlow across devices and servers.....	353
Multiple devices on a single machine	354
Installation	354

Managing the GPU RAM	357
Placing operations on devices	359
Parallel execution	362
Control dependencies	363
Multiple devices across multiple servers	364
Opening a session	366
The master and worker services	366
Pinning operations across tasks	367
Sharding variables across multiple parameter servers	368
Sharing state across sessions using resource containers	369
Asynchronous communication using TensorFlow queues	370
Loading data directly from the graph	376
Parallelizing neural networks on a TensorFlow cluster	383
One neural network per device	383
In-graph <i>vs</i> between-graph replication	384
Model parallelism	387
Data parallelism	389
Exercises	395
A. Exercise solutions.....	397
B. Machine Learning project checklist.....	407
C. Autodiff.....	413
Index.....	421

Preface

The Machine Learning tsunami

In 2006, Geoffrey Hinton *et al.* published a paper¹ showing how to train a deep neural network capable of recognizing hand-written digits with state of the art precision (>98%). They branded this technique “Deep Learning”. Training a deep neural net was widely considered impossible at the time² and most researchers had abandoned the idea since the 1990s. This paper revived the interest of the scientific community and before long many new papers demonstrated that Deep Learning was not only possible, but capable of mind-blowing achievements that no other Machine Learning technique could hope to match (with the help of tremendous computing power and great amounts of data). This enthusiasm soon extended to many other areas of Machine Learning.

Fast-forward 10 years and Machine Learning has conquered the industry: it is now at the heart of much of the magic in today’s high-tech products, ranking your web search results, powering your smartphone’s speech recognition, recommending videos, beating the world champion at the game of Go, and before you know it it will be driving your car.

Machine Learning in your projects

So naturally you are excited about Machine Learning and you would love to join the party!

Perhaps you would like to give your home-made robot a brain of its own? Make it recognize faces? Or learn to walk around?

¹ Available on Hinton’s home page at <http://www.cs.toronto.edu/~hinton/>.

² Despite the fact that Yann Lecun’s deep convolutional neural networks worked well for image recognition since the 1990s, but they were not as general purpose.

Or maybe your company has tons of data (user logs, financial data, production data, machine sensor data, hotline stats, HR reports, etc.) and it is more than likely that some hidden gems can be unearthed if you just knew where to look:

- segment customers and find the best marketing strategy for each group,
- recommend products for each client based on what similar clients bought,
- detect which transactions are likely to be fraudulent,
- predict next year's revenue,
- etc.³

Whatever the reason, you have decided to learn Machine Learning and implement it in your projects. Great idea!

Objective and approach

This book assumes that you know close to nothing about Machine Learning. Its goal is to give you the concepts, the intuitions and the tools you need to actually implement programs capable of **learning from data**.

We will cover a large number of techniques, from the simplest and most commonly used (such as Linear Regression) to some of the Deep Learning techniques that regularly win competitions.

Rather than implementing our own toy versions of each algorithm, we will be using actual production-ready python frameworks:

- **Scikit-Learn** is very easy to use yet it implements many Machine Learning algorithms efficiently, so it makes for a great entry point to learn Machine Learning.
- **TensorFlow** is a more complex library for distributed numerical computation using data flow graphs. It makes it possible to train and run very large neural networks efficiently by distributing the computations across potentially thousands of multi-GPU servers. TensorFlow was created at Google and it supports many of their large scale Machine Learning applications. It was open-sourced in November 2015.

The book favors a hands-on approach, growing an intuitive understanding of Machine Learning through concrete working examples and just a little bit of theory. While you can read this book without picking up your laptop, we highly recommend you experiment with the code examples available online as Jupyter notebooks at <https://github.com/ageron/handson-ml>.

³ Find many more examples at <https://www.kaggle.com/wiki/DataScienceUseCases>.

Prerequisites

This book assumes that you have some python programming experience and that you are familiar with python's main scientific libraries, in particular **NumPy**, **Pandas** and **Matplotlib**.

Also, if you care about what's under the hood you should have a reasonable understanding of college-level math as well (calculus, linear algebra, probabilities and statistics).

If you don't know python yet, <http://learnpython.org/> is a great place to start. The official tutorial on python.org is also quite good.

If you have never used Jupyter, [Chapter 2](#) will guide you through installation and the basics: it is a great tool to have in your toolbox.

If you are not familiar with python's scientific libraries, the provided Jupyter notebooks include a few tutorials. There is also a quick math tutorial for Linear Algebra.

Roadmap

This book is organized in three parts:

1. The Fundamentals of Machine Learning

- What is Machine Learning? What problems does it try to solve? What are the main categories of Machine Learning systems and the fundamental concepts?
- The main steps in a typical Machine Learning project.
- Learning by fitting a model to data.
- Optimizing a cost function.
- Handling, cleaning and preparing data.
- Selecting and engineering features.
- Selecting a model and tuning hyper-parameters using cross-validation.
- The main challenges of Machine Learning, in particular underfitting and overfitting (the Bias/Variance tradeoff).
- Reducing the dimensionality of the training data to fight the curse of dimensionality.
- And of course we will go through the most common learning algorithms: Linear and polynomial regression, Logistic regression, k-Nearest Neighbors, Support Vector Machines, Decision Trees, Random Forests and Ensemble methods.

2. Neural Networks and Deep Learning

- What are neural nets? What are they good for?
- Building and training neural nets using TensorFlow.
- The most important neural net architectures: FeedForward Neural Nets, Convolutional Nets, Recurrent Nets, Long Short Term Memory Nets and Auto-Encoders.
- Techniques for training deep neural nets.
- Scaling neural networks for huge datasets.

3. Other types of learning

- Clustering and Hierarchical clustering.
- Reinforcement Learning.
- Anomaly Detection.
- Recommender Systems.

The first part is based mostly on Scikit-Learn, the second part is based mostly on TensorFlow, and the third part relies on both.



Don't jump into deep waters too hastily: while Deep Learning is no doubt one of the most exciting areas in Machine Learning, you should master the fundamentals first. Moreover, most problems can be solved quite well using simpler techniques such as Random Forests and Ensemble methods (part one). Deep Learning is best suited for complex problems such as image recognition, speech recognition or natural language processing, provided you have enough data, computing power and patience.

Other resources

Many resources are available to learn about Machine Learning (ML). Andrew Ng's ML course on Coursera⁴ and Geoffrey Hinton's course on Neural Networks and Deep Learning⁵ are amazing, although they both require a significant time investment (think months).

There are also many interesting websites about Machine Learning, including of course Scikit-Learn's exceptional [User Guide](#). You may enjoy as well <https://www.data>

⁴ <https://www.coursera.org/learn/machine-learning/>

⁵ <https://www.coursera.org/course/neuralnets>

[quest.io/](https://www.questranslate.com) which provides very nice interactive tutorials. And ML blogs such as those listed in this quora answer: <http://goo.gl/GwtU3A>. <http://deeplearning.net/> has a good list of resources to learn more.

Of course there are also many other introductory books about Machine Learning, in particular:

- “Data Science from Scratch”, by J. Grus. This book presents the fundamentals of Machine Learning, and implements some of the main algorithms in pure python (from scratch, as the name suggests).
- “Machine Learning, an algorithmic perspective”, by S. Marsland. A great introduction to Machine Learning, covering a wide range of topics in depth, with code examples in python (also from scratch, but using NumPy).
- “Python Machine Learning”, by S. Raschka. Also a great introduction to Machine Learning, leveraging python open source libraries (Pylearn 2 and Theano).
- “Learning From Data”, by Y. Abu-Mostafa, M. Magdon-Ismail, and H. Lin. A rather theoretical approach to ML, providing deep insights, in particular on the Bias/Variance tradeoff (see [Chapter 4](#)).
- “Artificial Intelligence, a modern approach”, by S. Russel and P. Norvig. This is a great (and huge) book covering an incredible amount of topics, including Machine Learning. It helps put ML into perspective.

Finally, a great way to learn is to join ML competition websites such as [Kaggle.com](https://www.kaggle.com): this will allow you to practice your skills on real world problems, with help and insights from some of the best ML professionals out there.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/ageron/handson-ml>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of [plans and pricing](#) for [enterprise](#), [government](#), [education](#), and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds [more](#). For more information about Safari Books Online, please visit us [online](#).

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://www.oreilly.com/catalog/0636920052289>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

I would like to thank my Google colleagues, in particular the YouTube video classification team, for teaching me so much about Machine Learning. I could never have

started this project without them. Special thanks to my personal ML gurus: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington and everyone at YouTube Paris.

Many thanks as well to O'Reilly's amazing staff, in particular Nicole Taché who gave me insightful feedback, always cheerful, encouraging and helpful. Thanks as well to Marie Beaugureau, Ben Lorica, Mike Loukides and Laurel Ruma for believing in this project and helping me define its scope. Thanks to Matt Hacker and all of the Atlas team for answering all my technical questions regarding formatting, asciidoc and latex.

I wish to thank my father in law, Michel Tessier, former mathematics teacher and now a great translator of Anton Tchekov, for helping me iron out some of the mathematics and notations in this book and reviewing the Linear Algebra notebook.

Lastly, huge thanks to my beloved wife Emmanuelle, who encouraged me throughout the writing of this book, and to my three wonderful kids, Alexandre, Rémi and Gabrielle, for being such a bright, loving and joyful trio.

PART I

The Fundamentals of Machine Learning

The Machine Learning landscape

When most people hear “Machine Learning”, they picture a robot: a dependable butler or a deadly Terminator depending on who you ask. But Machine Learning is not just a futuristic fantasy, it’s already here. In fact, it has been around for decades in some specialized applications, such as *Optical Character Recognition* (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: it was the *spam filter*. Not exactly a self-aware Skynet, but it does technically qualify as Machine Learning (it has actually learned so well that you seldom need to flag an email as spam anymore). It was followed by hundreds of ML applications which now quietly power hundreds of products and features that you use regularly, from better recommendations to voice search.

Where does Machine Learning start and where does it end? What exactly does it mean for a machine to *learn* something? If I download a copy of the Wikipedia, has my computer really “learned” something? Is it suddenly smarter? In this chapter we will start by clarifying what Machine Learning is and why you may want to use it.

Then before we set out to explore the Machine Learning continent, we will take a look at the map and learn about the main regions and the most notable landmarks: supervised *vs* unsupervised learning, online *vs* batch learning, instance based *vs* model based learning. Then we will look at the workflow of a typical ML project, discuss the main challenges you may face and how to evaluate and fine-tune a Machine Learning system.

This chapter introduces a lot of fundamental concepts (and jargon) that every data scientist should know by heart. It will be a high level overview (the only chapter without much code), all rather simple but you should make sure everything is crystal clear to you. So grab a coffee and let’s get started!



If you already know all the Machine Learning basics, you may want to skip directly to [Chapter 2](#). If you are not sure, try to answer all the questions listed at the end of the chapter.

What is Machine Learning?

Machine Learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

—Arthur Samuel, 1959

And a more engineering-oriented one:

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

—Tom Mitchell, 1997

For example, your spam filter is a Machine Learning program that can learn to flag spam given examples of spam emails (eg. flagged by users) and examples of regular (non-spam, also called “ham”) emails. The examples that the system uses to learn are called the *training set*. Each training example is called a training *instance* (or *sample*). In this case, the task T is to flag spam for new emails, the experience E is the training data, and the performance measure P needs to be defined, for example you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy* and it is often used in classification tasks.

If you just download a copy of the Wikipedia, your computer has a lot more data but it is not suddenly better at any task. Thus it is not Machine Learning.

Why use Machine Learning?

Consider how you would write a spam filter using traditional programming techniques ([Figure 1-1](#)):

1. first you would look at what spam typically looks like. You might notice that some words (such as “4U”, “credit card”, “Free”, “Amazing”...) tend to come up a lot in the subject. Perhaps you would also notice a few other patterns in the sender’s name, the email’s body, etc.

2. then you would write a detection algorithm for each of the patterns that you noticed above and your program would flag emails as spam if a number of these patterns are detected.
3. you would test your program, and repeat steps 1 and 2 until it is good enough.

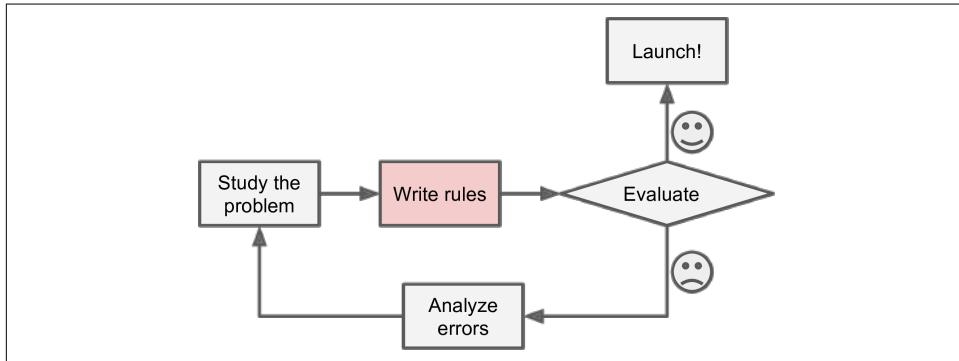


Figure 1-1. The traditional approach

Since the problem is not trivial, your program will likely become a long list of complex rules, pretty hard to maintain.

In contrast, a spam filter based on Machine Learning techniques automatically learns which words and phrases are good predictors of spam by detecting unusually frequent patterns of words in the spam examples compared to the ham examples. The program is much shorter, easier to maintain and most likely more accurate.

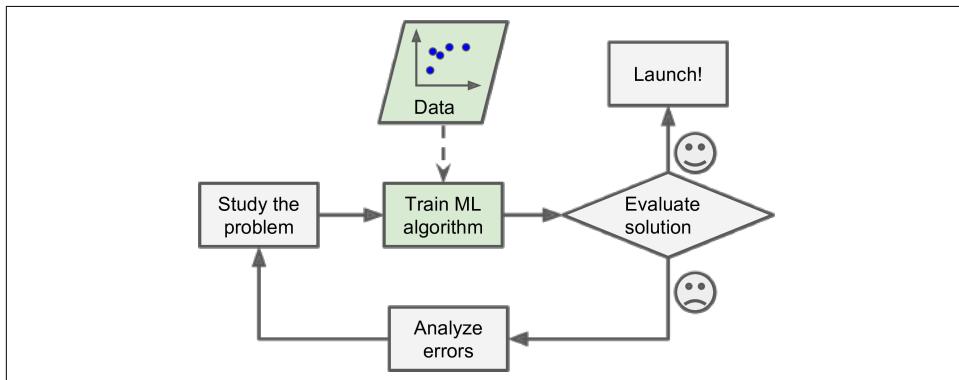


Figure 1-2. Machine Learning approach

Moreover, if spammers notice that all their emails containing the word “4U” are blocked, they might start writing “For U” instead. A spam filter using traditional programming techniques would need to be updated to flag “For U” emails. If spammers

keep working around your spam filter, you will need to keep writing new rules forever.

In contrast, a spam filter based on Machine Learning techniques automatically notices that “For U” has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention.

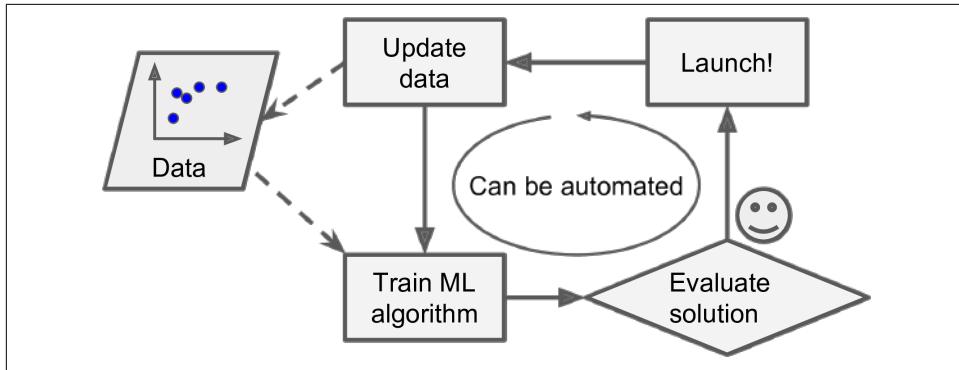


Figure 1-3. Automatically adapting to change

Another area where Machine Learning shines is for complex problems for which there is no known algorithm, or they are too complex. For example, consider speech recognition: say you want to start simple and write a program capable of distinguishing the words “One” and “Two”. You might notice that the word “Two” starts with a high-pitch sound (“T”), so you could hard-code an algorithm that measures high-pitch sound intensity and use that to distinguish ones and twos. Obviously this technique will not scale to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages. The best solution (at least today) is to write an algorithm that learns by itself, given many example recordings for each word.

Finally, Machine Learning can help humans learn: Machine Learning algorithms can be inspected to see what they have learned (although for some algorithms this can be tricky). For instance, once the spam filter has been trained on enough spam it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam. Sometimes this will reveal unsuspected correlations or new trends, and thereby lead to a better understanding of the problem.

Applying Machine Learning techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called *data mining*.

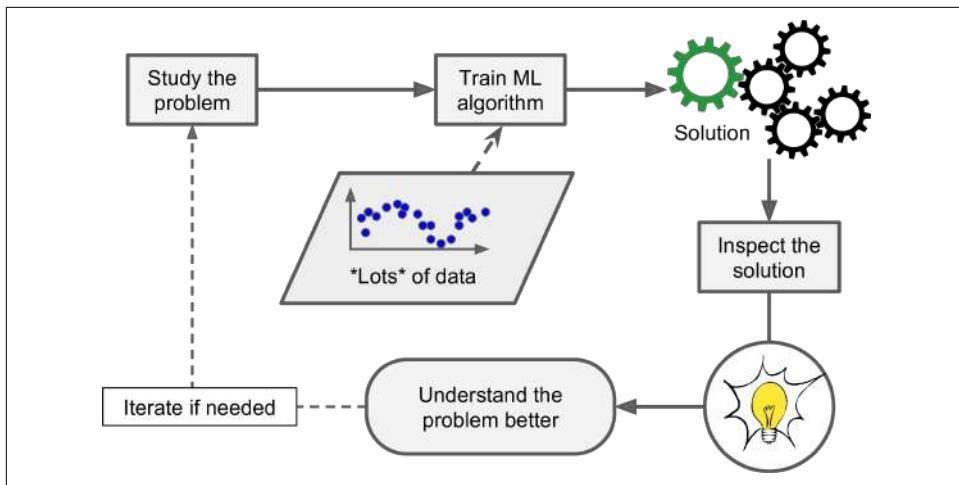


Figure 1-4. Machine Learning can help humans learn

To summarize, Machine Learning is great for:

- problems for which existing solutions require a lot of hand-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better.
- complex problems for which there is no good solution at all using a traditional approach: the best Machine Learning techniques can find a solution.
- fluctuating environments: a Machine Learning system can adapt to new data.
- and getting insights about complex problems and large amounts of data.

Types of Machine Learning systems

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories based on:

- whether or not they are trained with human supervision (supervised, unsupervised, semi-supervised and reinforcement learning),
- whether or not they can learn incrementally on the fly (online *vs* batch learning),
- whether they work by simply comparing new data points to known data points, or instead they detect patterns in the training data and build a predictive model, much like scientists do (instance based *vs* model based learning).

These criteria are not exclusive, you can combine them in any way you like. For example, a state-of-the-art spam filter may learn on the fly using a deep neural net-

work model trained using examples of spam and ham: this makes it an online, model based, supervised learning system.

Let's look at each of these criteria a bit more closely.

Supervised/Unsupervised learning

Machine Learning systems can be classified according to the amount and type of supervision they get during training. There are four major categories: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning.

Supervised learning

In *supervised learning*, the training data you feed to the algorithm includes the desired solutions, called *labels*.

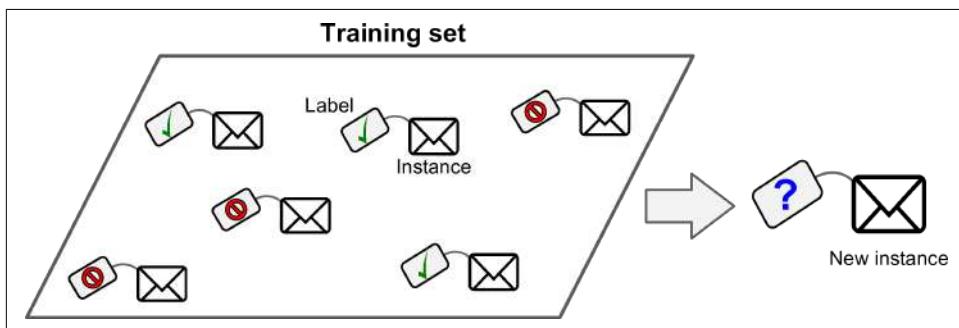


Figure 1-5. A labeled training set for supervised learning (eg. spam classification)

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value such as the price of a car given a set of *features* (eg. mileage, age, brand, etc.) called *predictors*. This sort of task is called *regression*¹. To train the system, you need to give it many examples of cars, including both their predictors and their labels (ie. their prices).

¹ Fun fact: this odd-sounding name is a statistics term introduced by Francis Galton while he was studying the fact that tall people tend to have smaller children than them. Since children were smaller, he called this *regression to the mean*. This name was then applied to the methods he used to analyze correlations between variables.



In Machine Learning an *attribute* is a data type (eg. “Mileage”) while a *feature* has several meanings depending on the context, but generally it means an attribute plus its value (eg. “Mileage = 15,000”). Many people use the words *attribute* and *feature* interchangeably, though.

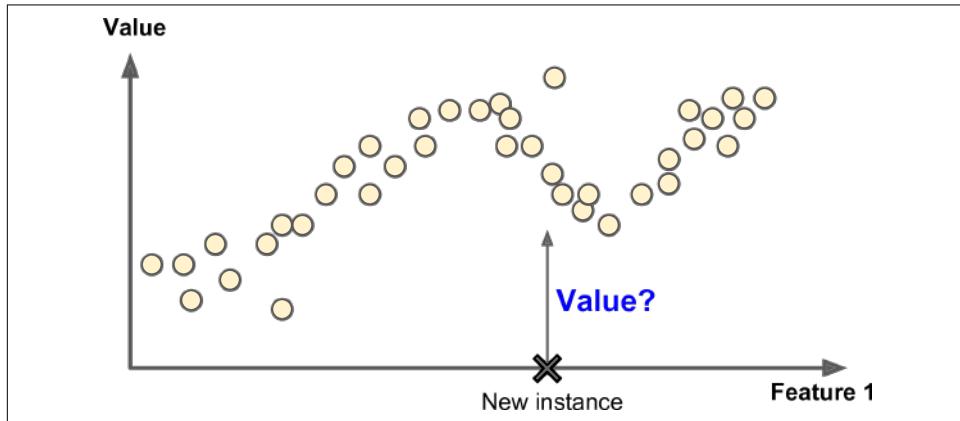


Figure 1-6. Regression

Note that some regression algorithms can be used for classification as well, and *vice versa*. For example, *Logistic Regression* is commonly used for classification, as it can output a value that corresponds to the probability of belonging to a given class (eg. 20% chance of being spam).

Here are some of the most important supervised learning algorithms (covered in this book):

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural Networks²

² Some Neural Network architectures can be unsupervised, such as auto-encoders or Restricted Boltzmann Machines. They can also be semi-supervised, such as in Deep Belief Networks and whenever using unsupervised pretraining.

Unsupervised learning

In *unsupervised learning*, as you might guess, the training data is unlabeled. The system tries to learn without a teacher.

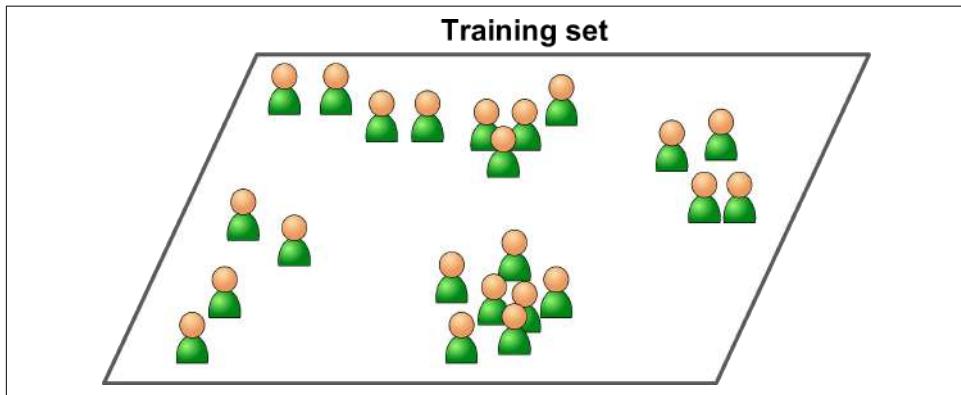


Figure 1-7. An unlabeled training set for unsupervised learning

Here are some of the most important unsupervised learning algorithms (we will cover dimensionality reduction in [Chapter 8](#)):

- Clustering
 - k-Means
 - Hierarchical Cluster Analysis (HCA)
 - Expectation Maximization
- Visualization and dimensionality reduction
 - Principal Component Analysis (PCA)
 - Kernel PCA
 - Locally-Linear Embedding (LLE)
 - t-distributed Stochastic Neighbor Embedding (t-SNE)
- Association rule learning
 - Apriori
 - Eclat

For example, say you have a lot of data about your blog's visitors, you may want to run a *clustering* algorithm to try to detect groups of similar visitors. At no point do you tell the algorithm which group a visitor belongs to: it finds them without your help. For example, it might notice that 40% of your visitors are males who love comic books and generally read your blog in the evening, while 20% are young science-fi

lovers who visit during the week-ends, and so on. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.

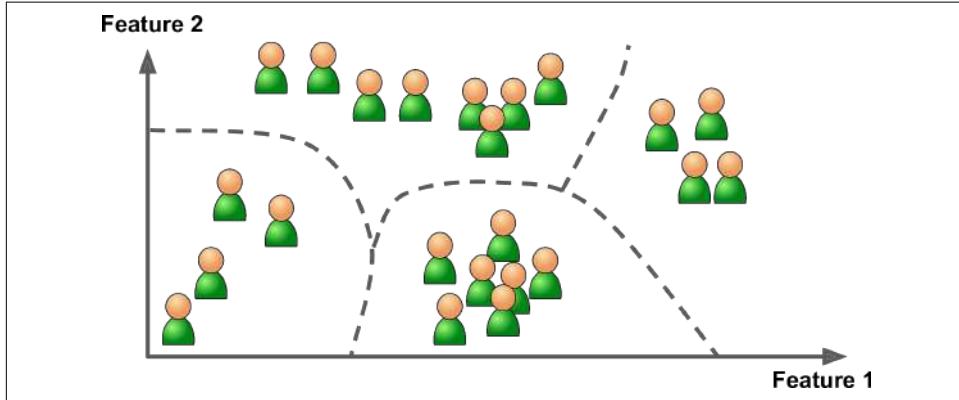


Figure 1-8. Clustering

Visualization algorithms are also good examples of unsupervised learning algorithms: you feed them a lot of complex and unlabeled data, and they output a 2D or 3D representation of your data that can easily be plotted. These algorithms try to preserve as much structure as they can (eg. trying to keep separate clusters in the input space from overlapping in the visualization), so you can understand how the data is organized and perhaps identify unsuspected patterns.

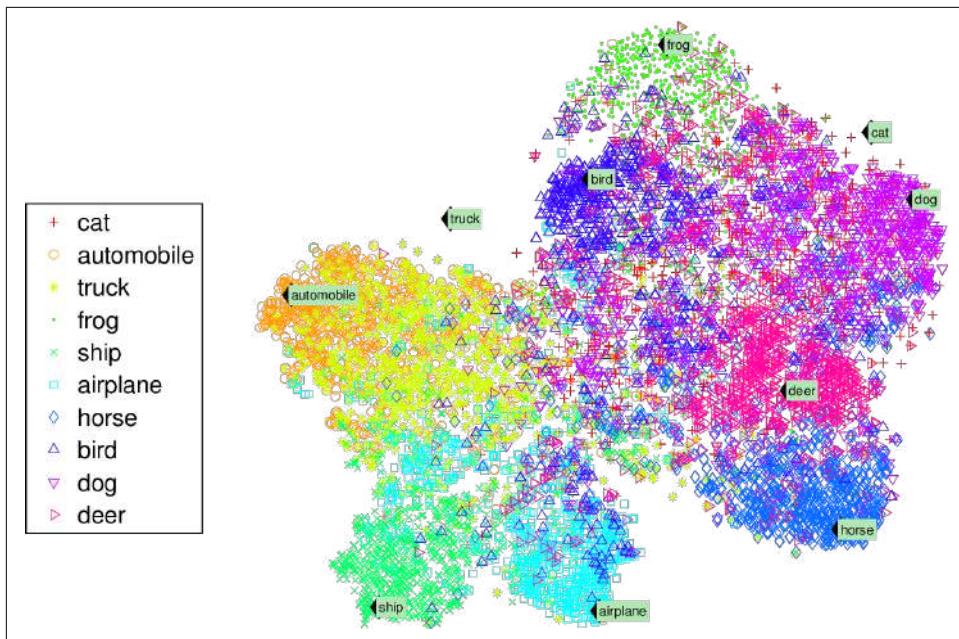


Figure 1-9. Example of a t-SNE visualization highlighting semantic clusters³

A related task is *dimensionality reduction*: the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one: for example, a car's mileage may be very correlated with its age so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.



It is often a good idea to try to reduce the dimension of your training data using a dimensionality reduction algorithm before you feed it to another Machine Learning algorithm (such as a supervised learning algorithm): it will run much faster, the data will take up less disk and memory space, and in some cases it may also perform better.

Yet another important unsupervised task is *anomaly detection*, for example detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is trained with normal instances, and when it sees a new

³ Notice how animals are rather well separated from vehicles, how horses are close to deers but far from birds, etc. Figure reproduced with permission from Socher, Ganjoo, Manning, Ng (2013). Original title: “T-SNE visualization of the semantic word space”.

instance it can tell whether it looks like a normal instance or whether it is likely to be an anomaly (see [Figure 1-10](#)).

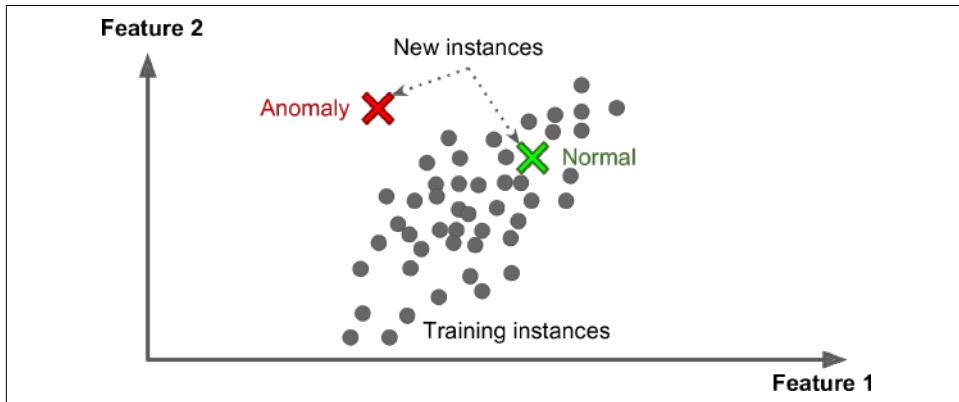


Figure 1-10. Anomaly detection

Finally, another common unsupervised task is *association rule learning*: the goal is to dig into large amounts of data and discover interesting relations between attributes. For example, suppose you own a supermarket, then running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. You may want to place these items close to each other.

Semi-supervised learning

Some algorithms can deal with partially labeled training data, usually a lot of unlabeled data and a little bit of labeled data. This is called *semi-supervised learning*.

Some photo hosting services such as Google Photos are good examples of this: once you upload all your family photos to the service, it automatically recognizes that the same person A shows up on photos 1, 5 and 11, while another person B shows up on photos 2, 5 and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are: just one label per person⁴ and it is able to name everyone on every photo, which is useful for searching photos.

⁴ That's when the system works perfectly. In practice it often creates a few clusters per person, and sometimes mixes up two people who look alike, so you need to provide a few labels per person and manually clean up some clusters.

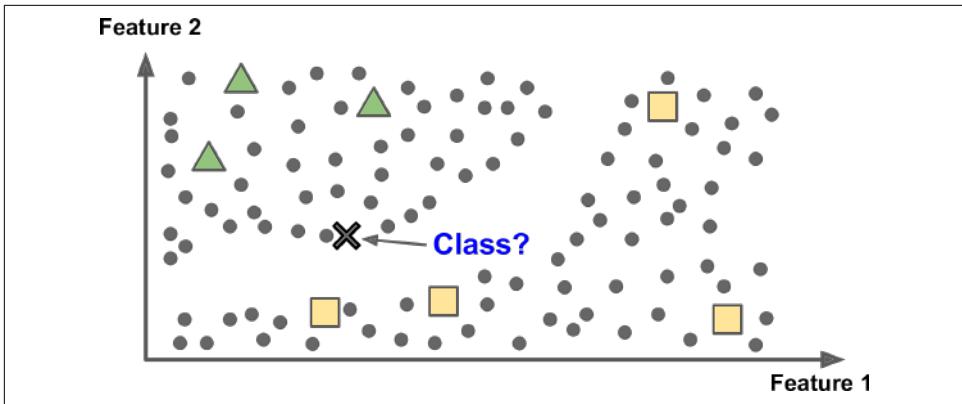


Figure 1-11. Semi-supervised learning

Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms, for example *Deep Belief Networks* (DBNs) are based on unsupervised components called *Restricted Boltzmann Machines* (RBMs) stacked on top of one another: RBMs are trained sequentially in an unsupervised manner, then the whole system is fine-tuned using supervised learning techniques.

Reinforcement learning

Reinforcement learning is a very different beast: the learning system, called an *agent* in this context, can observe the environment, select and perform actions, and get *rewards* in return (or *penalties* in the form of negative rewards, as in Figure 1-12). It must then learn by itself what is the best strategy, called a *policy*, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

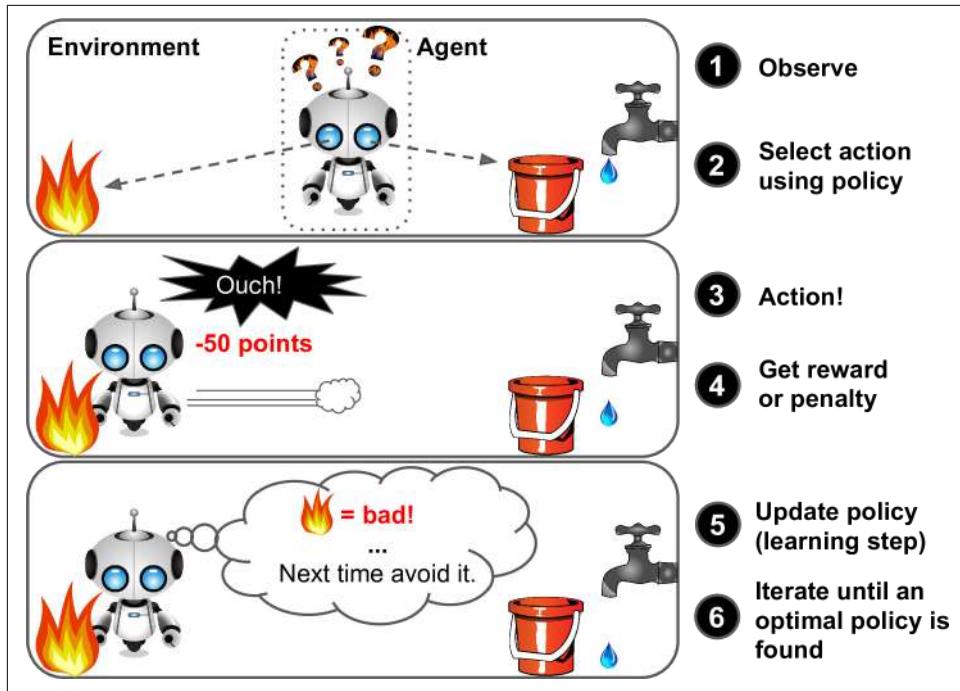


Figure 1-12. Reinforcement learning

For example, many robots implement reinforcement learning algorithms to learn how to walk. DeepMind's AlphaGo program is also a good example of reinforcement learning: it made the headlines in March 2016 when it beat the world champion Lee Sedol at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion, AlphaGo was just applying the policy it had learned.

Batch and Online learning

Another criterion used to classify Machine Learning systems is whether or not it can learn incrementally, from a stream of incoming data.

Batch Learning

In *batch learning*, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline: first the system is trained, then it is launched into production and it runs without learning anymore, it just applies what it has learned. This is called *offline learning*.

If you want a batch learning system to know about new data (such as new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.

Fortunately the whole process of training, evaluating and launching a Machine Learning system can be automated fairly easily (as shown on [Figure 1-3](#)), so even a batch learning system can adapt to change: simply update the data and train a new version of the system from scratch as often as needed.

This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (eg. to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O...). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm.

Finally, if your system needs to be able to learn autonomously and it has limited resources (eg. a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a show stopper.

Fortunately, a better option in all these cases is to use algorithms that are capable of learning incrementally.

Online Learning

In *online learning*, the system is trained incrementally by feeding it data instances sequentially, either individually or by small groups called *mini-batches*. Each learning step is fast and cheap so the system can learn about new data on the fly, as it arrives (see [Figure 1-13](#)).

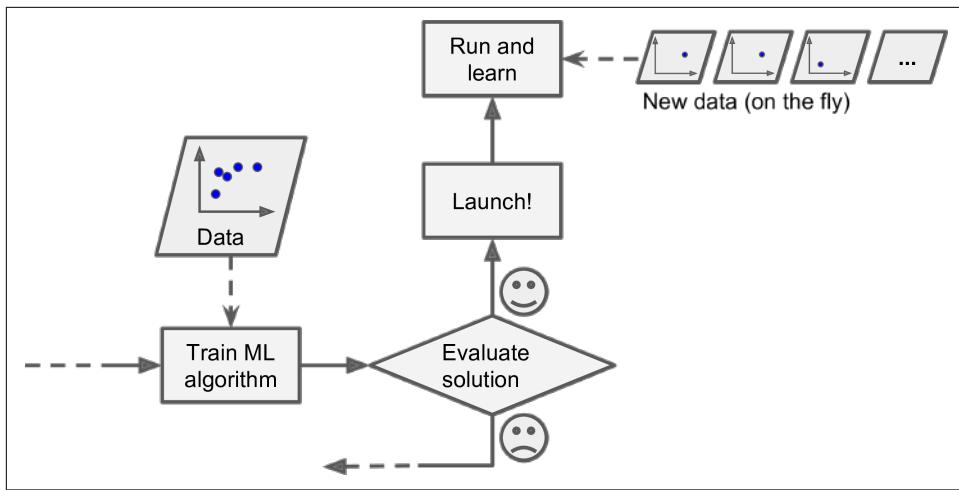


Figure 1-13. Online learning

Online learning is great for systems that receive data as a continuous flow (eg. stock prices) and need to adapt to change rapidly or autonomously. They are also well suited if you have limited computing resource: once an online learning system has learned about new data instances, it does not need them anymore so you can discard them (unless you want to be able to rollback to a previous state and “replay” the data). This can save a huge amount of space.

Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine’s main memory (this is called *out-of-core* learning). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data (see Figure 1-14).



This whole process is usually done offline (ie. not on the live system), so *online learning* can be a confusing name. Think of it as *incremental learning*.

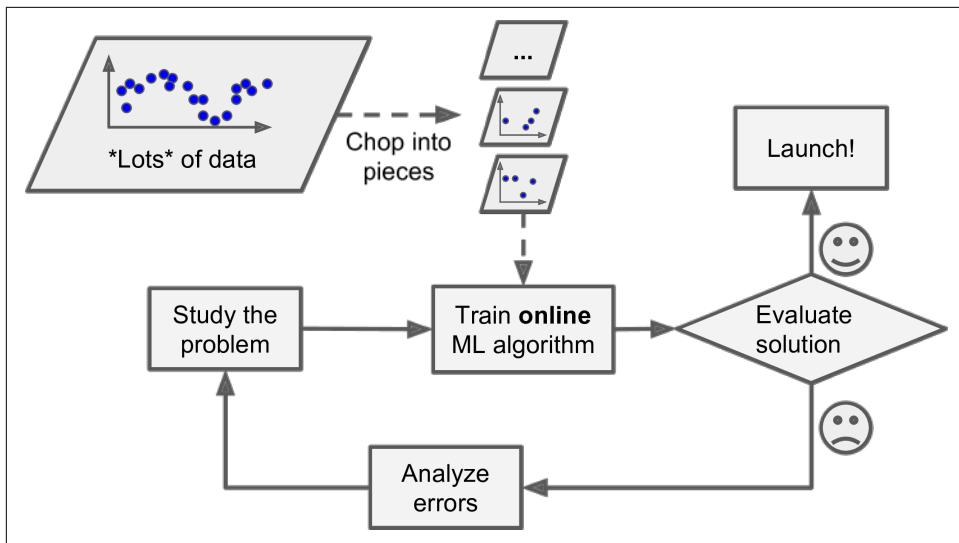


Figure 1-14. Using online learning to handle huge datasets

One important parameter of online learning systems is how fast they should adapt to changing data: this is called the *learning rate*. If you set a high learning rate, then your system will rapidly adapt to new data but it will also tend to quickly forget the old data (you don't want a spam filter to only flag the latest kinds of spam it was shown). Conversely, if you set a small learning rate, the system will have more inertia: it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of non-representative data points.

A big challenge with online learning is that if bad data is fed to the system, it will gradually perform worse. If we are talking about a live system, your clients will notice. For example, bad data could come from a malfunctioning sensor on a robot. Another example is someone spamming a search engine to try to rank high in search results. To reduce this risk you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data (eg. using an anomaly detection algorithm).

Instance based vs Model based learning

One more way to categorize Machine Learning systems is on how they *generalize*. Most Machine Learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to generalize to examples it has never seen before. Having a good performance measure on the training data is good, but insufficient: the true goal is to perform well on new instances.

There are two main approaches to generalization: instance based learning, and model based learning.

Instance based learning

Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are exactly identical to emails that have already been flagged by users. Not the worst solution but certainly not the best.

Instead of just flagging emails that are identical to known spam emails, you could program it to also flag emails that are very similar to known spam emails. This requires a *measure of similarity* between two emails. A (very basic) similarity measure between two emails could be to count the number of words that they have in common. The system would flag an email as spam if it has many words in common with a known spam email.

This is called *instance based learning*: the system learns the examples by heart, then it generalizes to new cases using a similarity measure.



Figure 1-15. Instance based learning

Model based learning

Another way to generalize from a set of examples is to build a model of these examples, then use the model to make *predictions*. This is called *model based learning*.

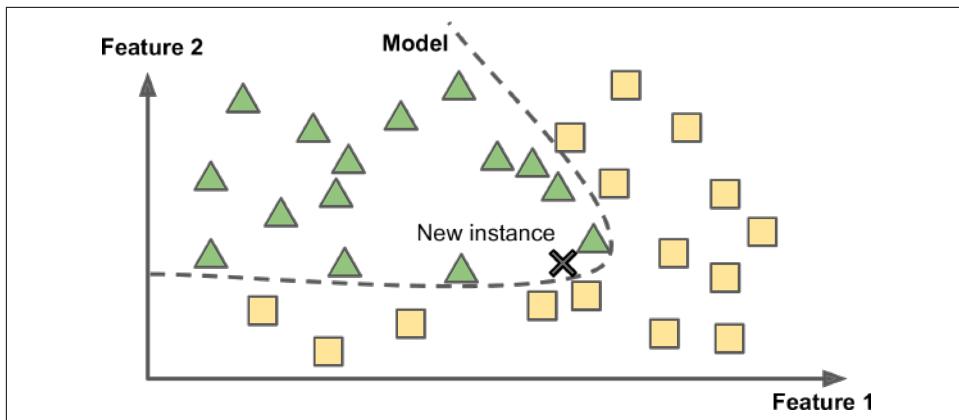


Figure 1-16. Model based learning

For example, suppose you want to know if money makes people happy, so you download the *Better Life Index* data from the OECD's website⁵ as well as stats about GDP per capita from the IMF's website⁶, you join the tables and sort by GDP per capita. Table 1-1 shows an excerpt of what you get:

Table 1-1. Does money make people happier?

Country	GDP per capita (\$US)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2

Let's plot the data for a few random countries (Figure 1-17).

⁵ <https://goo.gl/0Eht9W>

⁶ <http://goo.gl/j1MSKe>

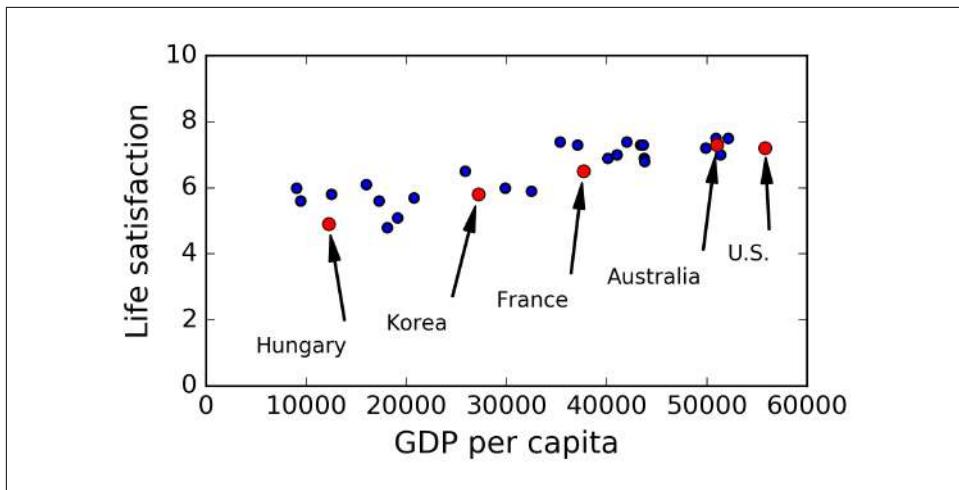


Figure 1-17. Do you see a trend here?

Mmh, there does seem to be a trend here! Although the data is *noisy* (ie. partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called *model selection*: you selected a *linear model* of life satisfaction with just one attribute, GDP per capita.

Equation 1-1. A simple linear model

$$\text{life_satisfaction} = \theta_0 + \theta_1 \times \text{GDP_per_capita}$$

This model has 2 *model parameters*, θ_0 and θ_1 .⁷ By tweaking these parameters, you can make your model represent any linear function, as shown on Figure 1-18.

⁷ By convention, the Greek letter θ (theta) is frequently used to represent model parameters.

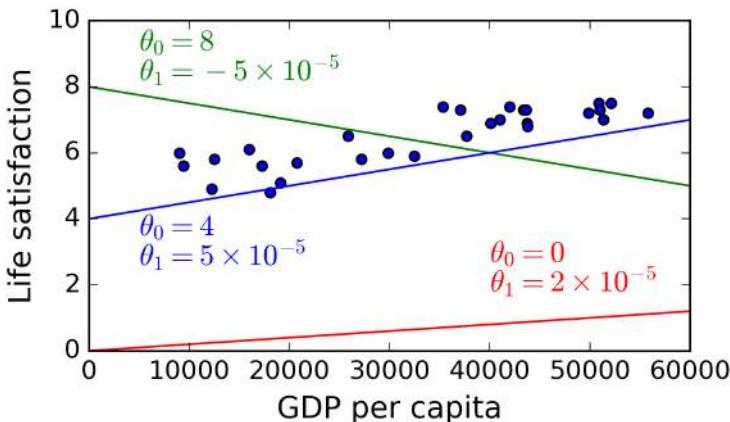


Figure 1-18. A few possible linear models

Before you can use your model, you need to define the parameter values θ_0 and θ_1 . How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a *utility function* (or *fitness function*) that measures how *good* your model is, or you can define a *cost function* that measures how *bad* it is. For linear regression problems, people typically use a cost function that measures the distance between the linear model's predictions and the training examples: the objective is to minimize this distance.

This is where the linear regression algorithm comes in: you feed it your training examples and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case the algorithm finds that the optimal parameter values are $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$.

Now the model fits the training data as closely as possible (for a linear model), as you can see in Figure 1-19

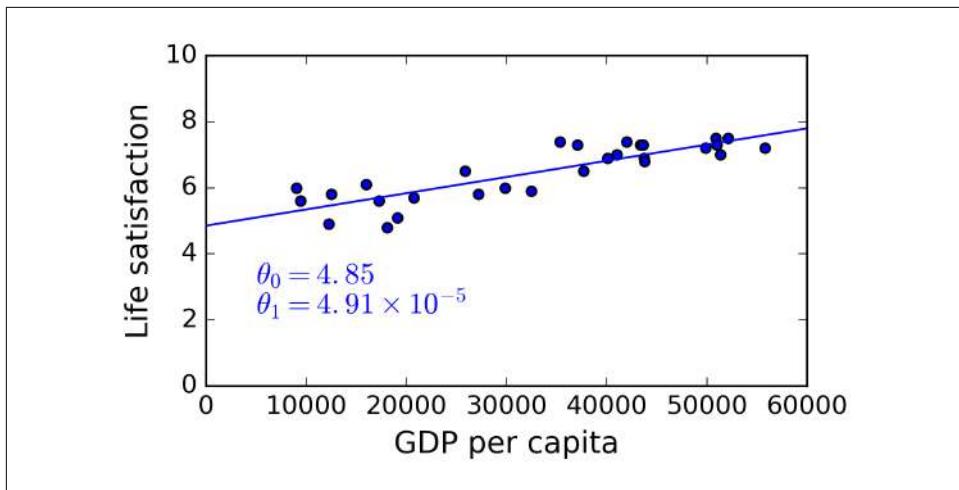


Figure 1-19. The linear model that fits the training data best

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, the OECD data does not have the answer, but fortunately you can use your model to make a good prediction: you look up Cyprus' GDP per capita, you find \$22,587, and then you apply your model and find that life satisfaction is likely to be somewhere around $4.85 + 22,587 \times 4.91 \times 10^{-5} = 5.96$.

To whet your appetite, [Example 1-1](#) shows the python code that loads the data, prepares it⁸, creates a scatterplot for visualization, then trains a linear model and makes a prediction⁹:

Example 1-1. Training and running a linear model using Scikit-Learn

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv", thousands=',', delimiter='\t',
                             encoding='latin1', na_values="n/a")
```

⁸ The code assumes that the `prepare_country_stats()` is already defined: it merges the GDP and life satisfaction data into a single Pandas dataframe.

⁹ It's ok if you don't understand all the code yet, we will present Scikit-Learn in the next chapters.

```

# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select a linear model
lin_reg_model = sklearn.linear_model.LinearRegression()

# Train the model
lin_reg_model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]] # Cyprus' GDP per capita
print(lin_reg_model.predict(X_new)) # outputs [[ 5.96242338]]

```



If you had used an instance based learning algorithm instead, you would have found that Slovenia has the closest GDP per capita to that of Cyprus (\$20,732), and since the OECD data tells us that their life satisfaction is 5.7, you would have predicted a life satisfaction of 5.7 for Cyprus. If you zoom out a bit and look at the two next closest countries, you will find Portugal and Spain with life satisfactions of 5.1 and 6.5 respectively. Averaging these three values you get 5.77. Pretty close to your model based prediction. This simple algorithm is called *k-Nearest Neighbors* regression (in this example, k=3).

Replacing the linear regression model by k-Nearest neighbors regression in the code above is as simple as replacing this line:

```
clf = sklearn.linear_model.LinearRegression()
```

with this one:

```
clf = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

If all went well, your model will make good predictions. If not, you may need to use more attributes (eg. employment rate, health, air pollution, etc.), get more or better quality training data, or perhaps select a more powerful model (eg. a polynomial regression model).

In summary:

- you studied the data,
- you selected a model,
- you trained it on the training data (ie. the learning algorithm searched for the model parameter values that minimize a cost function),

- then you applied the model to make predictions on new cases (this is called *inference*), hoping that this model will generalize well.

This is what a typical Machine Learning project looks like. In [Chapter 2](#) you will experience this first-hand by going through an end-to-end project.

We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what are some of the most common categories of ML systems, and what a typical project workflow looks like. Now let's look at what can go wrong in learning and prevent you from making accurate predictions.

Main challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the two things that can go wrong are “bad algorithm” and “bad data”. Let's start with examples of bad data.

Insufficient quantity of training data

All it takes for a toddler to learn what an apple is is for you to point to an apple and say “apple” (possibly repeat the procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.

Machine Learning is not quite there yet: it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

The unreasonable effectiveness of data

In a famous paper by Microsoft Researchers Banko and Brill, published in 2001¹⁰, the authors showed that very different Machine Learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation¹¹ once they were given enough data (as you can see on [Figure 1-20](#)).

¹⁰ <http://goo.gl/R5enIE>

¹¹ For example, knowing whether to write “to”, “two” or “too” depending on the context.

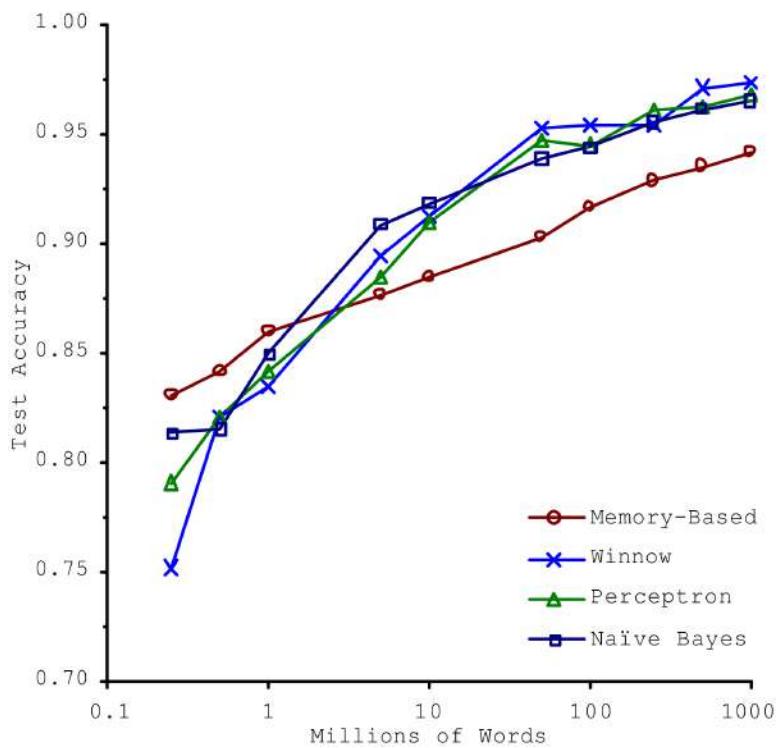


Figure 1-20. The importance of data vs algorithms¹²

As the authors put it: “[...]these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development.”

The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig *et al.* in a paper titled “The unreasonable effectiveness of data”¹³ published in 2009. It should be noted however that small and medium sized datasets are still very common, and it is not always easy or cheap to get extra training data, so don’t abandon algorithms just yet.

¹² Figure reproduced with permission from Banko & Brill (2001). Original title: “Learning Curves for Confusion Set Disambiguation”.

¹³ <http://goo.gl/q6LaZ8>

Non representative training data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance based learning or model based learning.

For example, the set of countries we used for training the linear model above was not perfectly representative: a few countries were missing. Here is what the data looks like when you add the missing countries:

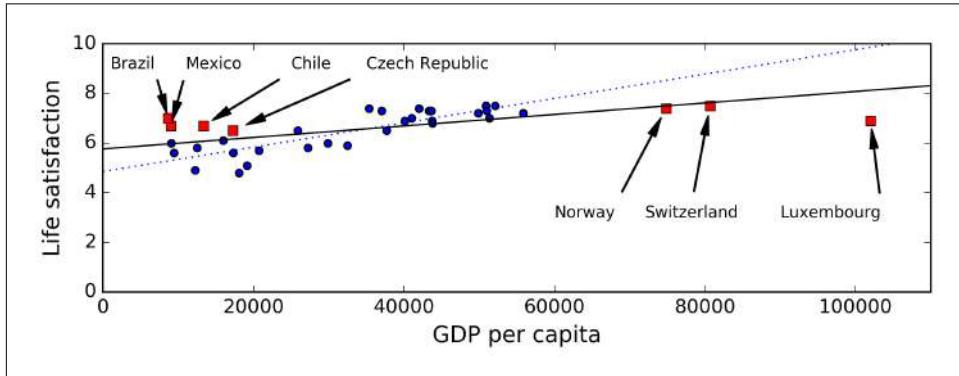


Figure 1-21. A more representative training sample

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact they seem unhappier), and conversely some poor countries seem happier than many rich countries.

By using a non representative training set we trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have *sampling noise* (ie. non-representative data as a result of chance), but even very large samples can be non-representative if the sampling method is flawed. This is called *sampling bias*.

A famous example of sampling bias

Perhaps the most famous example of sampling bias happened during the U.S. presidential election in 1936 which pitted Landon against Roosevelt: the Literary Digest conducted a very large poll, sending mail to about 10 million people. It got 2.4 million

answers, and predicted with high confidence that Landon would get 57% of the votes. Instead, Roosevelt won with 62% of the votes. The flaw was in the Literary Digest's sampling method:

- First, to obtain the addresses to send the polls to, they used telephone directories, lists of magazine subscribers, club membership lists, etc. All of these lists tend to favor wealthier people, who are more likely to vote Republican (hence Landon).
- Second, less than 25% of the people who received the poll answered. Again, this introduces a sampling bias, by ruling out people who don't care much about politics, people who don't like the Literary Digest, etc. This is a special type of sampling bias called *non-response bias*.

Here is another example: say you want to build a system to recognize funk music videos. One way to build your training set is to search "funk music" on YouTube and use the resulting videos. But this assumes that YouTube's search engine returns a set of videos that are representative of all the funk music videos on YouTube. In reality, the search results are likely to be biased towards popular artists (and if you live in Brazil you will get a lot of "funk carioca" videos which sound nothing like James Brown). On the other hand, how else can you get a large training set?

Poor quality data

Obviously, if your training data is full of errors, outliers and noise (eg. due to poor quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is most data scientists spend a significant part of their time doing just that. For example:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (eg. 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, or ignore these instances, or fill in the missing values (eg. with the median age), or train one model with the feature and one model without it, etc.

Irrelevant features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on: this is called *feature engineering*. This involves:

- *Feature selection*: the process of selecting the most useful features to train on among existing features.
- *Feature extraction*: combining existing features to produce a more useful one (as we saw earlier, dimensionality reduction algorithms can help).
- And of course creating new features by gathering new data.

Now that we have looked at many examples of bad data, let's look at a couple examples of bad algorithms.

Overfitting the training data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Over-generalizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called *overfitting*: it means that the model performs well on the training data but it does not generalize well.

[Figure 1-22](#) shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

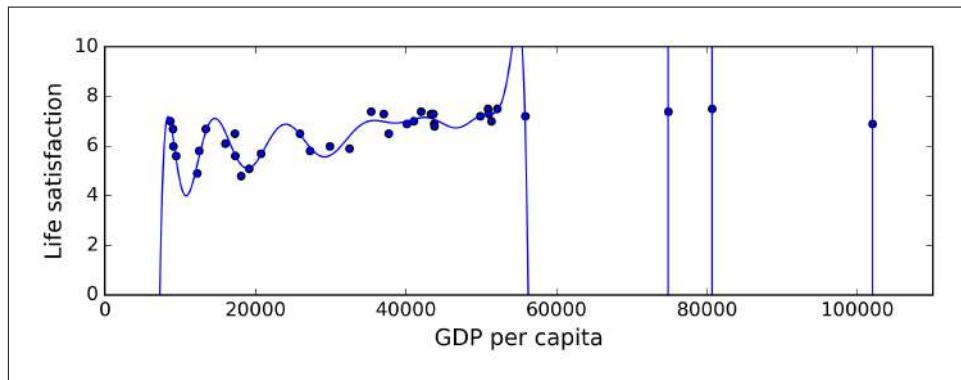


Figure 1-22. Overfitting the training data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small (which introduces sampling noise), then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances. For example, say you feed your life satisfaction model many more attributes, including uninformative ones such as the country's name, then a complex model may detect patterns like the fact that all countries in the training data with a "W" in their name have a life satisfaction greater than 7: New Zealand (7.3), Norway (7.4), Sweden (7.2) and Switzerland (7.5). How confident are

you that the W-satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.



Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. The possible solutions are:

- to simplify the model by selecting one with less parameters (eg. a linear model rather than a high degree polynomial model), by reducing the number of attributes in the training data or by constraining the model,
- to gather more training data,
- to reduce the noise in the training data (eg. fix data errors and remove outliers).

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*. For example, the linear model we defined earlier has two parameters θ_0 and θ_1 . This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height (θ_0) and the slope (θ_1) of the line. If we forced $\theta_1 = 0$, the algorithm would have only one degree of freedom and it would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a simpler model than with two degrees of freedom, but more complex than with just one. You want to find the right balance between fitting the data perfectly and keeping the model simple enough to ensure that it will generalize well.

The [Figure 1-23](#) shows three models: the dotted line represents the original model that was trained with a few countries missing, the dashed line is our second model trained with all countries, and the solid line is a linear model trained with the same data as the first model but with a regularization constraint. You can see that regularization forced the model to have a smaller slope, which fits a bit less the training data that the model was trained on, but actually allows it to generalize better to new examples.

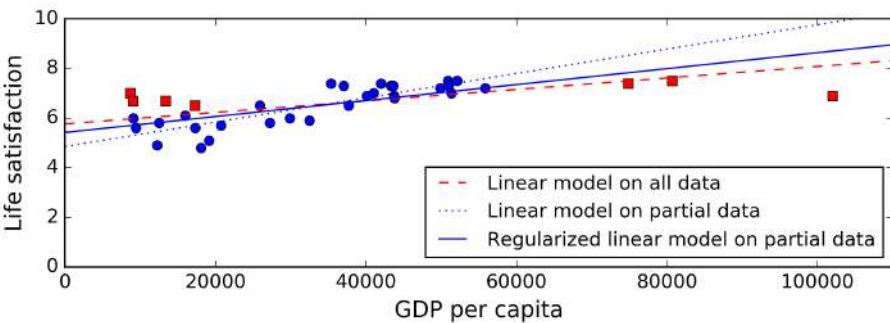


Figure 1-23. Regularization reduces the risk of overfitting

The amount of regularization to apply during learning can be controlled by a *hyperparameter*. A hyperparameter is a parameter of a learning algorithm (not of the model): as such, it is not affected by the learning algorithm itself, it must be set prior to training and remains constant during training. If you set the regularization hyperparameter to a very large value, you will get an almost flat model (a slope close to zero): the learning algorithm will almost certainly not overfit the training data, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a Machine Learning system (you will see a detailed example in the next chapter).

Underfitting the training data

As you might guess, *underfitting* is the opposite of overfitting; it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit: reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

The main options to fix this problem are:

- selecting a more powerful model, with more parameters,
- feeding better features to the learning algorithm (feature engineering),
- reducing the constraints on the model (eg. reducing the regularization hyperparameter).

Stepping back

By now you already know a lot about Machine Learning. However, we went through so many concepts that you may be feeling a little lost so let's step back and look at the big picture:

- Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules.
- There are many different types of ML systems, supervised or not, batch or online, instance based or model based, and so on.
- In a ML project you gather data in a training set, you feed the training set to a learning algorithm, and if it is model based it tunes some parameters to fit the model to the training set (ie. to make good predictions on the training set itself), then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance based, it just learns the examples by heart and uses a similarity measure to generalize to new instances.
- The system will not perform well if your training set is too small, or if the data is not representative, noisy or polluted with irrelevant features (garbage in, garbage out). Lastly, your model needs to be neither too simple (it will underfit) nor too complex (it will overfit).

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases: you want to evaluate it, and fine-tune it if necessary. Let's see how.

Testing and validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well but if your model is horribly bad, your users will complain. Not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimation of this error: this tells you how well your model will perform on instances it has never seen before.

If the training error is low (ie. your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.



It is common to use 80% of the data for training and *hold out* 20% for testing.

So evaluating a model is simple enough: just use a test set. Now suppose you are hesitating between two models (say a linear model and a polynomial model): how can you decide? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better but you want to apply some regularization to avoid overfitting. The question is: how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error, say just 5% error.

So you launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model *for that set*. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is to have a second holdout set called the *validation set*. You train multiple models with various hyperparameters using the training set, you select the model and hyperparameters that perform best on the validation set, and when you are happy about your model you run a single final test against the test set to get an estimate of the generalization error.

To avoid “wasting” too much training data in validation sets, a common technique is to use *cross-validation*: the training set is split into complementary subsets, each model is trained against a different combination of these subsets and it is validated against the remaining parts. Once the model type and hyperparameters have been selected, a final model is trained using these hyperparameters on the full training set, and the generalized error is measured on the test set.

No Free Lunch theorem

A model is a simplified version of the observations. The simplifications are meant to discard the superfluous details that are unlikely to generalize to new instances. However, to decide what data to discard and what data to keep, you must make *assumptions*. For example a linear model makes the assumption that the data is fundamentally linear and that the distance between the instances and the straight line is just noise, which can safely be ignored.

In a famous 1996 paper¹⁴, David Wolpert demonstrated that if you make absolutely no assumption about the data, then there is no reason to prefer a model over any other. This is called the *No Free Lunch theorem* (NFL). For some datasets the best model is a linear model, while for other datasets it is a neural network. There is no model that is *a priori* guaranteed to work better (hence the name of the theorem). The only way to know for sure which model is best is to evaluate them all. Since this is not possible, in practice you make some reasonable assumptions about the data and you evaluate only a few reasonable models. For example, for simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

Exercises

In this chapter we have covered some of the most important concepts in Machine Learning. In the next chapters we will dive deeper and write more code, but before we do, you should make sure you know how to answer the following questions:

1. How would you define Machine Learning?
2. Can you name 4 types of problems where it shines?
3. What is a labeled training set?
4. What are the two most common supervised tasks?
5. Can you name 4 common unsupervised tasks?
6. What type of Machine Learning algorithm would you use to allow a robot to walk in various unknown terrains?
7. What type of algorithm would you use to segment your customers into multiple groups?
8. Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?
9. What is an online learning system?
10. What is out-of-core learning?
11. What type of learning algorithm relies on a similarity measure to make predictions?
12. What is the difference between a model parameter and a learning algorithm's hyperparameter?

¹⁴ "The Lack of A Priori Distinctions Between Learning Algorithms", D. Wolpert (1996): <http://goo.gl/3zaHIZ>

13. What do model based learning algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?
14. Can you name 4 of the main challenges in Machine Learning?
15. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name 3 possible solutions?
16. What is a test set and why would you want to use it?
17. What is the purpose of a validation set?
18. What can go wrong if you tune hyperparameters using the test set?
19. What is cross-validation and why would you prefer it to a validation set?

Solutions to these exercises are available in [Appendix A](#).

End-to-end Machine Learning project

In this chapter, you will go through an example project end to end, pretending to be a recently hired data scientist in a real estate company¹. Here are the main steps you will go through:

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor and maintain your system.

Working with real data

When learning about Machine Learning it is best to actually experiment with real world data, not just artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

- Popular open data repositories:

¹ The example project is completely fictitious, the goal is just to illustrate the main steps of a Machine Learning project, not to learn anything about the real-estate business.

- UC Irvine Machine Learning Repository
- Kaggle datasets
- Amazon's AWS datasets
- Meta portals (they list open data repositories):
 - <http://dataportals.org/>
 - <http://opendatamonitor.eu/>
 - <http://quandl.com/>
- Other pages listing many popular open data repositories:
 - Quora.com question (<http://goo.gl/zDR78y>)
 - Datasets subreddit

In this chapter we chose the California Housing Prices dataset from the StatLib repository² (see Figure 2-1). This dataset was based on data from the 1990 California census. It is not exactly recent (you could still afford a nice house in the Bay Area at the time), but it has many qualities for learning so we will pretend it is recent data. We also added a categorical attribute and removed a few features for teaching purposes.

² The original dataset appeared in Pace, Kelley and Barry (1997), “Sparse Spatial Autoregressions”, Statistics and Probability Letters.

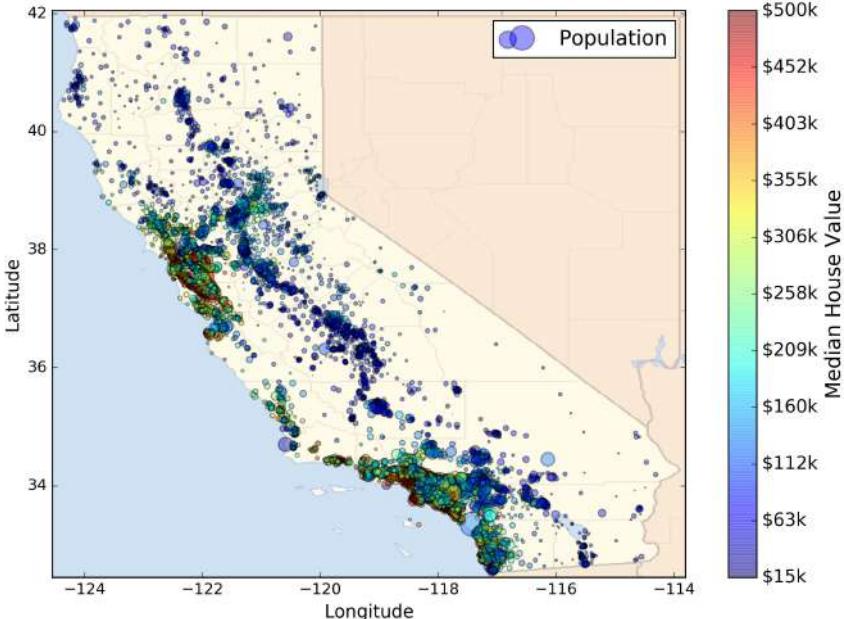


Figure 2-1. California housing prices

Look at the big picture

Welcome to Machine Learning Housing Corp.! The first task you are asked to perform is to build a model of housing prices in California using the California census data. This data has metrics such as the population, median income, median housing price, and so on, for each block group in California. Block groups are the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will just call them “districts” for short.

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.



Since you are a well organized data scientist, the first thing you do is to pull out your Machine Learning project checklist. You can start with the one in [Appendix B](#), it should work reasonably well for most Machine Learning projects but make sure to adapt it to your needs. In this chapter we will go through many checklist items, but we will also skip a few, either because they are self-explanatory or because they will be discussed in later chapters.

Frame the problem

The first question to ask your boss is what exactly is the business objective: building a model is probably not the end-goal. How does the company expect to use and benefit from this model? This is important because it will determine how you frame the problem, what algorithms you will select, what performance measure you will use to evaluate your model, and how much effort you should spend tweaking it.

Your boss answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system (see [Figure 2-2](#)), along with many other *signals*³. This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical as it directly affects revenue.

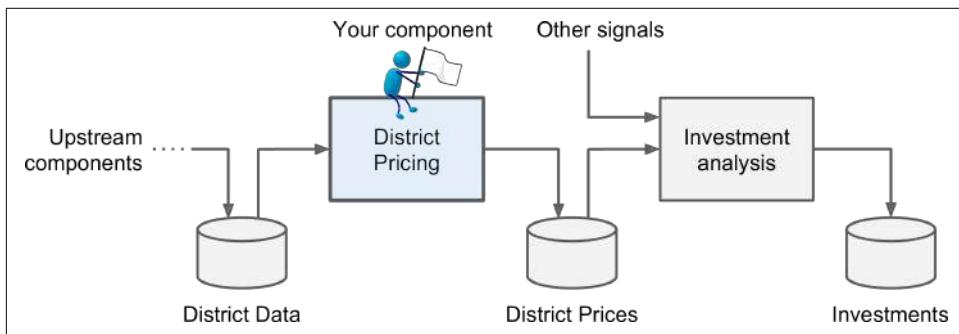


Figure 2-2. A Machine Learning pipeline for real-estate investments

Pipelines

A sequence of data processing *components* is called a data *pipeline*. These are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically run asynchronously: each component pulls in a large amount of data, processes it and spits out the result in another data store, then some time later the next component in the pipeline pulls this data and spits out its own output, and so on. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system quite simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.

³ A piece of information fed to a Machine Learning system is often called a *signal* in reference to Shannon's information theory: you want a high signal/noise ratio.

On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system's performance drops.

Next question to ask is what the current solution looks like (if any). It will often give you a reference performance, as well as insights on how to solve the problem. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district (excluding median housing prices), and they use complex rules to come up with an estimate. This is costly, time consuming and their estimates are not great: they typically make about 15% error.

Ok, with all this information you are now ready to start designing your system. First, you need to frame the problem: is it supervised, unsupervised or reinforcement learning? Is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques? Before you read on, pause and try to answer these questions for yourself.

Have you found the answers? Let's see: it is clearly a typical supervised learning task since you are given *labelled* training examples (each instance comes with the expected output, ie. the district's median housing price). Moreover, it is also a typical regression task, since you are asked to predict a value. More specifically this is a *multivariate regression* problem since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.). In the first chapter, you predicted life satisfaction based on just one feature, the GDP per capita, so it was a *univariate regression* problem. Finally, there is no continuous flow of data coming in the system, and no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory so plain batch learning should do just fine.



If the data was huge, you could either split your batch learning work across multiple servers (using the *map-reduce* technique, as we will see later), or you could use an online learning technique instead.

Select a performance measure

Next step is to select a performance measure: a typical performance measure for regression problems is the Root Mean Square Error (RMSE). It measures the *standard deviation*⁴ of the errors the system makes in its predictions. For example an RMSE

⁴ The standard deviation, generally noted σ (the Greek letter sigma), is the square root of the *variance*, which is the average of the squared deviation from the mean.

equal to 50,000 means that about 68% of the system's predictions fall within \$50,000 of the actual value, and about 95% of the predictions fall within \$100,000 of the actual value⁵. [Equation 2-1](#) shows the mathematical formula to compute the RMSE.

Equation 2-1. Root Mean Square Error (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

Notations

This equation introduces several very common Machine Learning notations that we will use throughout this book:

- m is the number of instances in the dataset you are measuring the RMSE on.
 - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2,000$.
- $\mathbf{x}^{(i)}$ is a vector of all the feature values (excluding the label) of the i^{th} instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance).
 - For example, if the first district in the dataset is located at longitude -118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of \$38,372, and the median house value is \$156,400 (ignoring the other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

and

$$y^{(1)} = 156,400$$

⁵ When a feature has a bell-shaped *normal distribution* (also called a *Gaussian distribution*), which is very frequent, the “68-95-99.7” rule applies: about 68% of the values fall within 1σ of the mean, 95% within 2σ , and 99.7% within 3σ .

- \mathbf{X} is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance and the i^{th} row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$.⁶
 - For example, if the first district is as described above, then the matrix \mathbf{X} looks like this:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ for that instance (\hat{y} is pronounced "y-hat").
 - For example, if your system predicts that the median housing price in the first district is \$158,400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158,400$. The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2,000$.
- $\text{RMSE}(\mathbf{X}, h)$ is the cost function measured on the set of examples using your hypothesis h .

We use lower case italic font for scalar values (such as m or $y^{(i)}$) and function names (such as h), lower case bold font for vectors (such as $\mathbf{x}^{(i)}$), and upper case bold font for matrices (such as \mathbf{X}).

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts, you may consider using the *Mean Absolute Error* (also called the Average Absolute Deviation):

Equation 2-2. Mean Absolute Error

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

⁶ Recall that the transpose operator flips a column vector into a row vector (and *vice versa*).

Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values. Various distance measures, or *norms*, are possible:

- Computing the root of a sum of squares (RMSE) corresponds to the *Euclidian norm*: it is the notion of distance you are familiar with. It is also called the ℓ_2 norm, noted $\|\cdot\|_2$ (or just $\|\cdot\|$).
- Computing the sum of absolutes (MAE) corresponds to the ℓ_1 norm, noted $\|\cdot\|_1$. It is sometimes called the *Manhattan norm* because it measures the distance between two points in a city if you can only travel along orthogonal city blocks.
- More generally, the ℓ_k norm of a vector \mathbf{v} containing n elements is defined as
$$\|\mathbf{v}\|_k = \left(|v_0|^k + |v_1|^k + \cdots + |v_n|^k \right)^{\frac{1}{k}}.$$
 ℓ_0 just gives the cardinality of the vector (ie. the number of elements), and ℓ_∞ gives the maximum absolute value in the vector.
- The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

Check the assumptions

Lastly, it is good practice to list and verify the assumptions that were made so far (by you or others): this can catch serious issues early on. For example, the district prices that your system outputs are going to be fed into a downstream Machine Learning system, and we assume that these prices are going to be used as such. But what if the downstream system actually converts the prices into categories (eg. “cheap”, “medium” or “expensive”) and then uses those categories instead of the prices themselves? In this case, getting the price perfectly right is not important at all, your system just needs to get the category right: if so, then the problem should have been framed as a classification task, not a regression task. You don’t want to find this out after working on a regression system for months.

Fortunately, after talking with the team in charge of the downstream system, you are confident that they do indeed need the actual prices, not just categories. Great! You are all set, the lights are green, you can start coding now!

Get the data

It's time to get your hands dirty. Don't hesitate to pick up your laptop and walk through the code examples below in a Jupyter notebook. The full Jupyter notebook is available at <https://github.com/ageron/handson-ml>.

Create the workspace

First you will need to have Python installed. It is probably already installed on your system. If not you can get it at <https://www.python.org/>⁷.

Next you need to create a workspace directory for your Machine Learning code and datasets. Open a terminal and type the following commands (after the \$ prompts):

```
$ export ML_PATH="$HOME/ml"      # You can change the path if you prefer  
$ mkdir -p $ML_PATH
```

You will need a number of python modules: Jupyter, NumPy, Pandas, Matplotlib and Scikit-Learn. If you already have Jupyter running with all these modules installed, you can safely skip to “[Download the data](#)” on page 66. If you don't have them yet, there are many ways to install them (and their dependencies): you can use your system's packaging system (eg. apt-get on Ubuntu, or MacPorts or HomeBrew on MacOSX), or install a Scientific Python distribution such as Anaconda and use its packaging system, or you can just use Python's own packaging system which is included by default with the Python binary installers (since Python 2.7.9): pip.⁸ You can check to see if pip is installed by typing the following command:

```
$ pip3 --version  
pip 8.1.2 from [...]/lib/python3.5/site-packages (python 3.5)
```

You should make sure you have a recent version of pip installed, at the very least >1.4 to support binary module installation (a.k.a. wheels). To upgrade the pip module, type:⁹

```
$ pip3 install --upgrade pip  
Collecting pip
```

⁷ The latest version of Python 3 is recommended. Python 2.7+ should work fine too, but it is deprecated.

⁸ We will show the installation steps using pip in a bash shell on a Linux or MacOSX system. You may need to adapt these commands to your own system. On Windows, we recommend installing Anaconda instead.

⁹ You may need to administrator rights to run this command, for example by prefixing it with sudo.

```
[...]
Successfully installed pip-8.1.2
```

Creating an isolated environment

If you would like to work in an isolated environment (which is strongly recommended so you can work on different projects without having conflicting library versions), you should install virtualenv by running the following pip command:

```
$ pip3 install --user virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

Now you can create an isolated python environment by typing:

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix '[...]'
New python executable in [...]/ml/env/bin/python3.5
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

Now every time you want to activate this environment, just open a terminal and type:

```
$ cd $ML_PATH
$ source env/bin/activate
```

While the environment is active, any package you install using pip will be installed in this isolated environment, and python will only have access to these packages (if you also want access to the system's site packages, you should create the environment using virtualenv's `--system-site-packages` option). Check out virtualenv's documentation for more information.

Now you can install all the required modules and their dependencies using this simple pip command:

```
$ pip3 install jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
[...]
```

To check your installation, try to import every module like this:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

There should be no output and no error. Now you can fire up Jupyter by typing:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
```

```
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/  
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all  
kernels (twice to skip confirmation).
```

A Jupyter server is now running in your terminal, listening to port 8888, and you can visit this server by opening your Web browser to <http://localhost:8888/> (this usually happens automatically when the server starts). You should see your empty workspace directory (containing only the `env` directory if you followed the `virtualenv` instructions above).

Now create a new Python notebook by clicking on the “New” button and selecting the appropriate Python version (see [Figure 2-3](#)).¹⁰

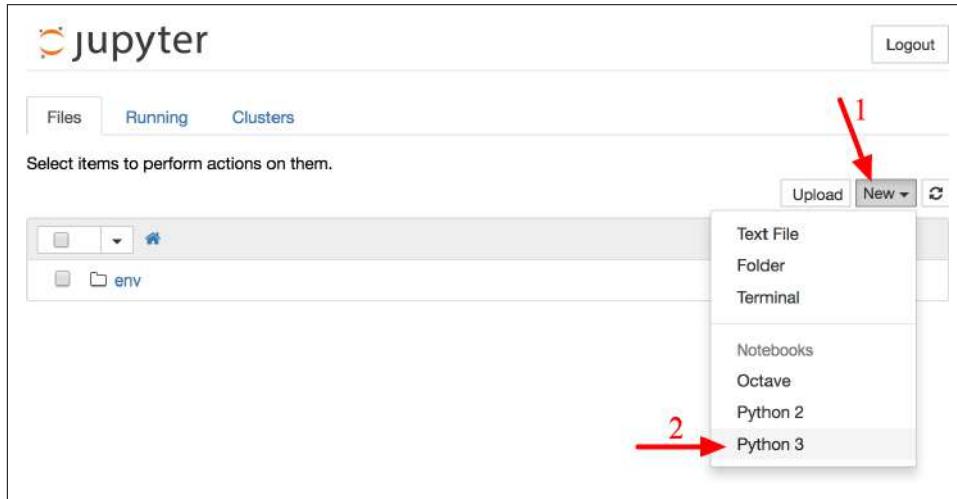


Figure 2-3. Your workspace in Jupyter

This does three things: first it creates a new notebook file called `Untitled.ipynb` in your workspace, second it starts a Jupyter python kernel to run this notebook, and third it opens this notebook in a new tab. You should start by renaming this notebook to “Housing” (this will automatically rename the file to `Housing.ipynb`) by clicking on “Untitled” and typing the new name.

A notebook contains a list of cells. Each cell can contain executable code or formatted text. Right now the notebook contains only one empty code cell, labeled “In [1]:”. Try typing `print("Hello world!")` in the cell, and click on the play button (see [Figure 2-4](#)) or type Shift-Enter. This sends the current cell to this notebook’s python kernel, which runs it and returns the output. The result is displayed below the cell,

¹⁰ Note that Jupyter can handle multiple versions of Python, and even many other languages such as R or Octave.

and since we reached the end of the notebook, a new cell is automatically created. Go through the User Interface Tour from Jupyter's Help menu to learn the basics.

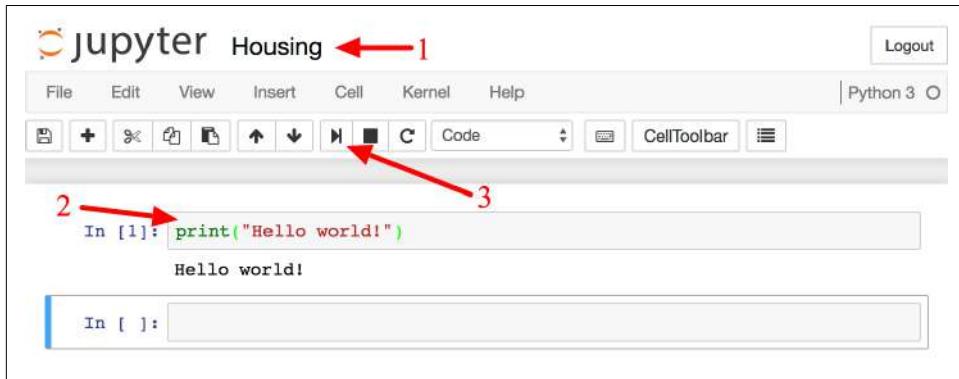


Figure 2-4. Hello world python notebook

Download the data

In typical environments your data would be available in a relational database (or some other common datastore), and spread across multiple tables/documents/files. You would first need to get your credentials and access authorizations¹¹, and familiarize yourself with the data schema. In this project however things will be much simpler: you will just download a single compressed file `housing.tgz` containing a CSV file `housing.csv` with all the data.

You could use your web browser to download it, and run `tar xzf housing.tgz` to decompress the file and extract the CSV file, but it is preferable to create a small function to do that. It is useful in particular if data changes regularly, as it allows you to write a small script that you can run whenever you need to fetch the latest data (or you can setup a scheduled job to do that automatically at regular intervals). Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

Here is the function to fetch the data¹²:

```
import os
import tarfile
import urllib.request
```

¹¹ You might also need to check legal constraints, such as private fields that should never be copied to unsafe datastores.

¹² In a real project you would save this code in a python file, but for now you can just write it in your Jupyter notebook

```

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()

```

Now when you call `fetch_housing_data()`, it creates a `datasets/housing` directory in your workspace, it downloads the `housing.tgz` file and extracts the `housing.csv` from it in this directory.

Now let's load the data using Pandas. Once again you should write a small function to load the data:

```

import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)

```

This function returns a Pandas DataFrame object containing all the data.

Take a quick look at the data structure

Let's take a look at the top 5 rows using the DataFrame's `head()` method (see Figure 2-5).

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Figure 2-5. Top 5 rows in the dataset

Each row represents one district. There are 10 attributes (you can see the first 6 on the screenshot): `longitude`, `latitude`, `housing_median_age`, `total_rooms`,

`total_bedrooms`, `population`, `households`, `median_income`, `median_house_value`, and `ocean_proximity`.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, and each attribute's type and number of non-null values:

```
In [6]: housing.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude          20640 non-null float64
latitude           20640 non-null float64
housing_median_age 20640 non-null float64
total_rooms         20640 non-null float64
total_bedrooms      20433 non-null float64
population          20640 non-null float64
households          20640 non-null float64
median_income        20640 non-null float64
median_house_value   20640 non-null float64
ocean_proximity     20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 2-6. Housing info

There are 20,640 instances in the dataset, which means that it is fairly small by Machine Learning standards, but it's perfect to get started. Notice that the `total_bedrooms` attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature. We will need to take care of this later.

All attributes are numerical, except the `ocean_proximity` field: its type is `object`, so it could hold any kind of python object, but since you loaded this data from a CSV file you know that it must be a text attribute. When you looked at the top 5 rows, you noticed that the values in that column were repetitive, which means that it is probably a categorical attribute. You can find out what categories exist and how many districts belong to each category by using the `value_counts()` method:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN    9136
INLAND       6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND         5
Name: ocean_proximity, dtype: int64
```

Let's look at the other fields. The `describe()` method shows a summary of the numerical attributes:

In [8]:	housing.describe()					
Out[8]:						
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.0000	
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	
std	2.003532	2.135952	12.585558	2181.615252	421.385070	
min	-124.350000	32.540000	1.000000	2.000000	1.000000	
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	

Figure 2-7. Summary of each numerical attribute

The `count`, `mean`, `min` and `max` rows are self-explanatory. Note that the null values are ignored (so for example `count` of `total_bedrooms` is 20,433, not 20,640). The `std` row shows the standard deviation (which measures how dispersed the values are). The 25%, 50% and 75% rows show the corresponding *percentiles*: a percentile indicates the value below which a given percentage of observations in a group of observations fall. For example, 25% of the districts have a `housing_median_age` lower than 18, while 50% are lower than 29 and 75% are lower than 37. These are often called the 25th percentile (or 1st *quartile*), the median, and the 75th percentile (or 3rd quartile).

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. A histogram shows the number of instances (on the vertical axis) have a given value range (on the horizontal axis). You can either do this one attribute at a time, or you can call the `hist()` method on the whole dataset, and it will plot a histogram for each numerical attribute (see Figure 2-8). For example, you can see that a bit over 800 districts have a `median_house_value` equal to about \$500,000.

```
%matplotlib inline # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

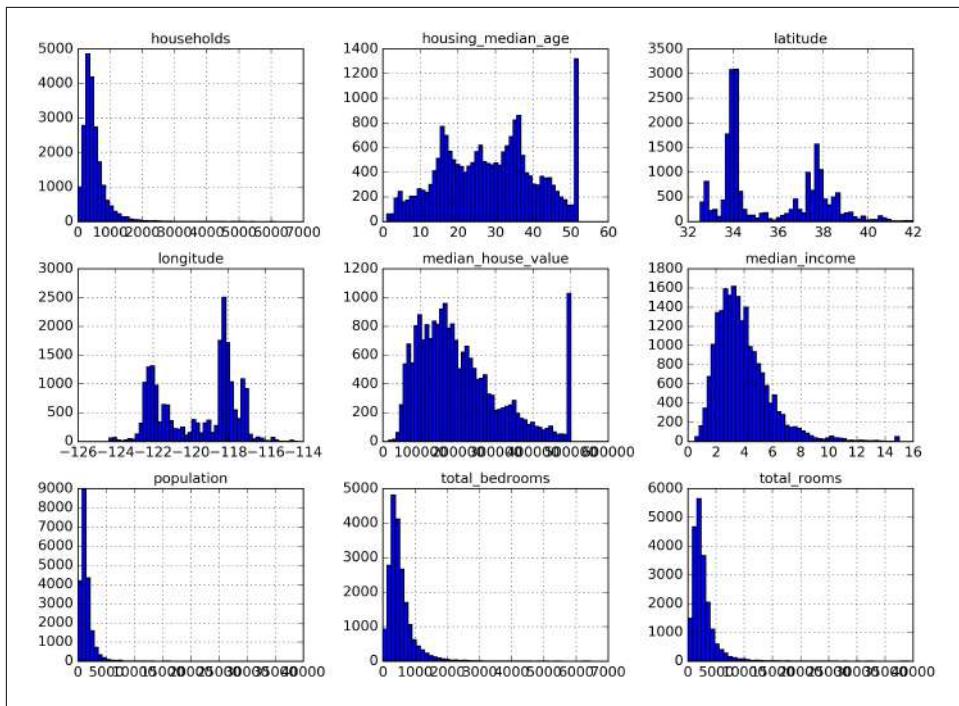


Figure 2-8. A histogram for each numerical attribute



The `hist()` method relies on Matplotlib, which in turn relies on a user-specified graphical backend to draw on your screen. So before you can plot anything, you need to specify which backend Matplotlib should use. The simplest option is to use Jupyter's magic command `%matplotlib inline`: this tells Jupyter to setup Matplotlib so it uses Jupyter's own backend. Plots are then rendered within the notebook itself. Note that calling `show()` is optional in a Jupyter notebook, as Jupyter will automatically display plots when a cell is executed.

You can notice a few things in these histograms:

- First, the median income attribute does not look like it is expressed in \$US. After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually 15.0001) for higher median incomes, and at 0.5 (actually 0.4999) for lower median incomes. Working with preprocessed attributes is common in Machine Learning, and it is not necessarily a problem, but you should try to understand how the data was computed.

- Notice that the housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit. You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond \$500,000, then you have mainly two options:
 1. collect proper labels for the districts whose labels were capped,
 2. remove those districts from the training set (and also from the test set. since your system should not be evaluated poorly if it predicts values beyond \$500,000).
- These attributes have very different scales. We will discuss this later in this chapter when we explore feature scaling.
- Finally, notice that many histograms are *tail heavy*: they extend much farther to the right of the median than to the left. This may make it a bit harder for some Machine Algorithms to detect patterns. We will try transforming these attributes later on to have more bell-shaped distributions.

Hopefully you now have a better understanding of the kind of data you are dealing with.



Wait! Before you look at the data any further, you need to create a test set, put it aside and never look at it.

Create a test set

It may sound strange to voluntarily set aside part of the data at this stage: after all, you have only taken a quick glance at the data, surely you should learn a whole lot more about it before you decide what algorithms to use? This is true, but your brain is an amazing pattern detection system, which means that it is highly prone to overfitting: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model. When you will estimate the generalization error using the test set, your estimate will be too optimistic and you will launch a system that will not perform as well as expected. This is called *data snooping* bias.

Creating a test set is theoretically quite simple: just pick some instances randomly, typically 20% of the dataset, set them aside and you are done:

```
import numpy as np
import numpy.random as rnd
```

```

def split_train_test(data, test_ratio):
    shuffled_indices = rnd.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]

```

You can then use this function like this:

```

>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test

```

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set! Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid.

One solution is to save the test set on the first run and then load it in subsequent runs. Another option is to set the random number generator's seed (eg. `rnd.seed(42)`¹³ before calling `rnd.permutation()`), so that it always generates the same shuffled indices.

But both these solutions will break next time you fetch an updated dataset. A common solution is to use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier). For example, you could compute a hash of each instance's identifier, keep only the last byte of the hash and put the instance in the test set if this value is lower or equal to 51 (~20% of 256). This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset: the new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set. Here is a possible implementation:

```

import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(identifier).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]

```

Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the id:

¹³ You will often see people set the random seed to 42. This number has no special property, other than to be The Answer to the Ultimate Question of Life, the Universe, and Everything.

```
housing_with_id = housing.reset_index() # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset, and no row ever gets deleted. If this is not possible, then you can try to use the most stable features to build a unique identifier. For example, a district's latitude and longitude are guaranteed to be stable for a few million years so you could combine them into an id like so¹⁴:

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is `train_test_split` which does pretty much the same thing as the function `split_train_test` defined earlier, with a couple additional features: first there is a `random_state` parameter that allows you to set the random generator seed as explained above, and second you can pass it multiple datasets with identical number of rows, and it will split them on the same indices (this is very useful for example if you have a separate DataFrame for labels):

```
from sklearn.cross_validation import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phonebooth. They try to ensure that these 1,000 people are representative of the whole population. For example, the U.S. population is composed of 51.3% female and 48.7% male, so a well-conducted survey in the U.S. would try to maintain this ratio in the sample: 513 female and 487 male. This is called *stratified sampling*: the population is divided into homogeneous subgroups called *strata*, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population. If they used purely random sampling, there would be about 12% chance of sampling a skewed test set with either less than 49% female or more than 54% female. Either way, the survey results would be significantly biased.

Suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create

¹⁴ The location information is actually quite coarse, and as a result many districts will have the exact same id, so they will end up in the same set (test or train). This introduces some unfortunate sampling bias.

an income category attribute. Let's look at the median income histogram more closely (see [Figure 2-9](#)):

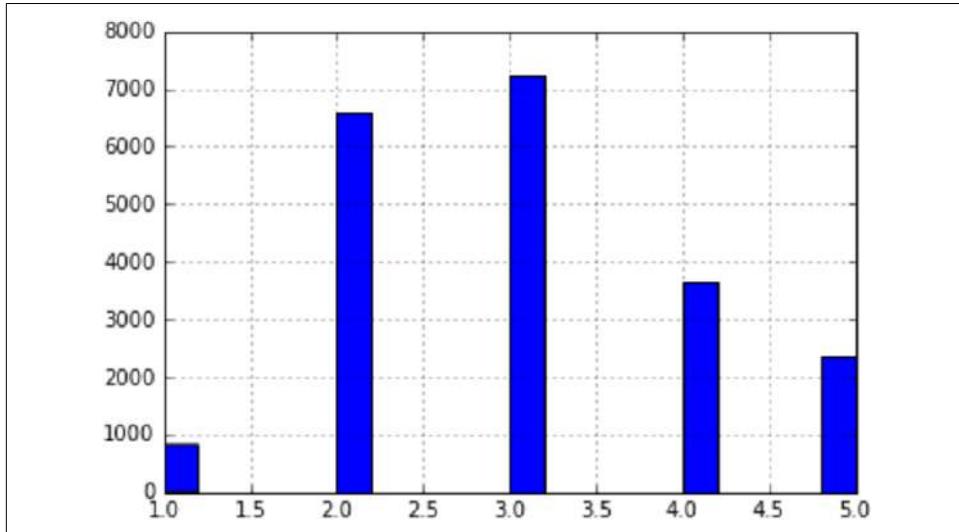


Figure 2-9. Histogram of income categories

Most median income values are clustered around 2-5 (tens of thousands of dollars), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of the stratum's importance may be biased. This means that you should not have too many strata, and each stratum should be large enough. The following code creates an income category attribute by dividing the median income by 1.5 (to limit the number of income categories), rounding up using `ceil` (to have discrete categories), then merging all the categories greater than 5 into category 5.

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn's `StratifiedShuffleSplit` class:

```
from sklearn.cross_validation import StratifiedShuffleSplit

split = StratifiedShuffleSplit(housing["income_cat"], test_size=0.2)
train_indices, test_indices = next(iter(split))
strat_train_set = housing.loc[train_indices]
strat_test_set = housing.loc[test_indices]
```

Let's see if this worked as expected. You can start by looking at the income category proportions in the full housing dataset:

```
>>> housing["income_cat"].value_counts() / len(housing)
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

With similar code you can measure the income category proportions in the test set. [Figure 2-10](#) compares the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling. As you can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is quite skewed.

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

Figure 2-10. Sampling bias comparison of stratified vs purely random sampling.

Now you should remove the `income_cat` attribute so the data is back to its original state:

```
for set in (strat_train_set, strat_test_set):
    set.drop(["income_cat"], axis=1, inplace=True)
```

We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project. Moreover, many of these ideas will be useful later when we discuss cross-validation. Now it's time to move on to the next stage: exploring the data.

Discover and visualize the data to gain insights

So far you have only taken a quick glance at the data to get a general understanding of the kind of data you are manipulating. Now the goal is to go a little bit more in depth.

First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small so you can

just work directly on the full set. Let's create a copy so you can play with it without harming the training set:

```
housing = strat_train_set.copy()
```

Visualizing geographical data

Since there is geographical information (latitude and longitude), it is a good idea to create a scatter plot of all districts to visualize the data:

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

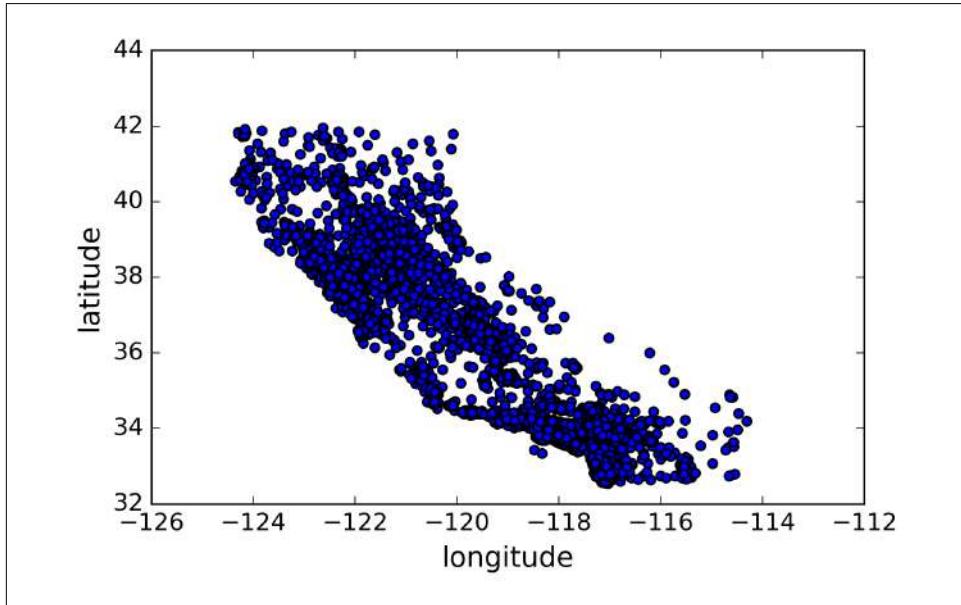


Figure 2-11. A geographical scatterplot of the data

This looks like California all right, but other than that it is hard to see any particular pattern. By setting the `alpha` option to 0.1, it is much easier to visualize the places where there is a high density of data points:

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

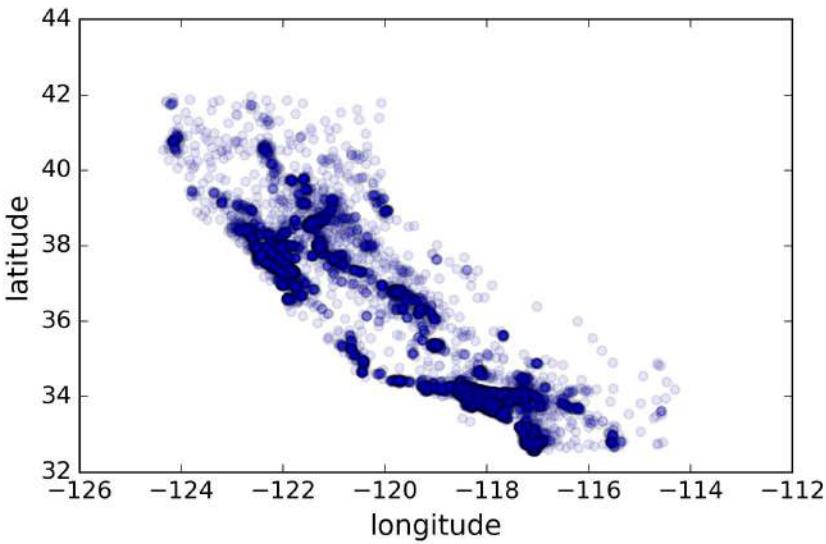


Figure 2-12. A better visualization highlighting high density areas

Now that's much better: you can clearly see the high density areas, namely the Bay area and around Los Angeles and San Diego, plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno.

More generally, our brains are very good at spotting patterns on pictures, but you may need to play around with visualization parameters to make the patterns stand out.

Now let's look at the housing prices. The radius of each circle will represent the district's population (using option `s`), and the color represents the price (option `c`). We will use a predefined color map (option `cmap`) called "jet", which ranges from blue (low values) to red (high prices).¹⁵

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
             s=housing["population"]/100, label="population",
             c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

¹⁵ If you are reading this in gray scale, grab a red pen and scribble over most of the coastline from the Bay Area down to San Diego (as you might expect). You can add a patch of yellow around Sacramento as well.

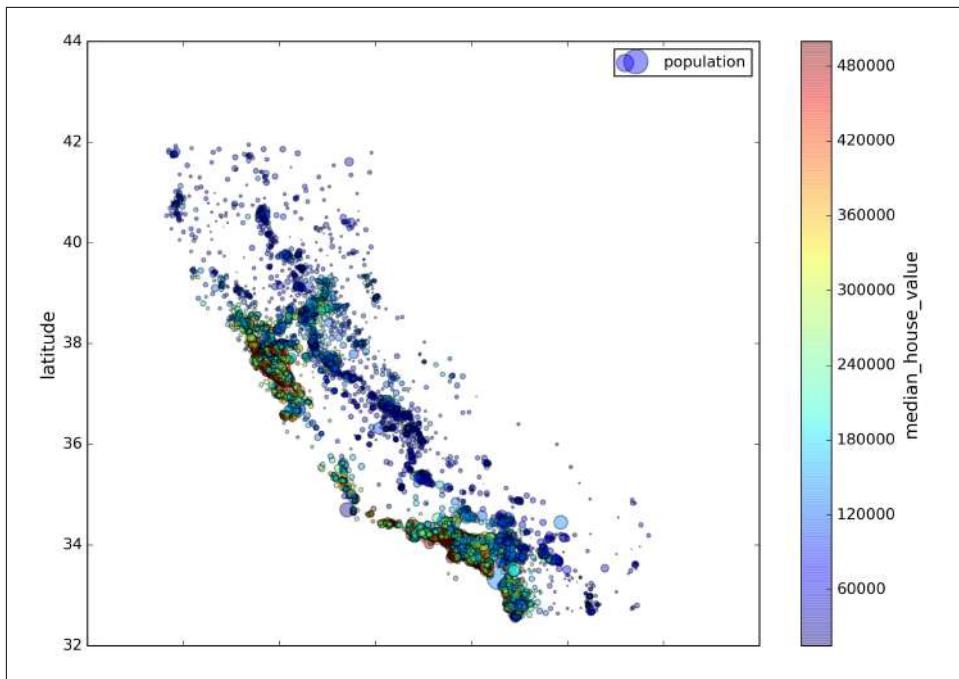


Figure 2-13. California housing prices

This image tells you that the housing prices are very much related to the location (eg. close to the ocean), and to the population density, as you probably knew already. It will probably be useful to use a clustering algorithm to detect the main clusters, and add new features that measure the proximity to the cluster centers. The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.

Looking for correlations

Since the dataset is not too large, you can easily compute the *standard correlation coefficient* (also called *Pearson's r*) between every pair of attributes using the `corr()` method:

```
corr_matrix = housing.corr()
```

Now let's look at how much each attribute correlates with the median house value:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income        0.687170
total_rooms          0.135231
housing_median_age   0.114220
households           0.064702
```

```

total_bedrooms      0.047865
population        -0.026699
longitude         -0.047279
latitude          -0.142826
Name: median_house_value, dtype: float64

```

The correlation coefficient ranges from -1 to 1. When it is close to 1, it means that there is a strong positive correlation: for example, the median house value tends to go up when the median income goes up. When the coefficient is close to -1, it means that there is a strong negative correlation: you can see a small negative correlation between the latitude and the median house value (ie. prices have a slight tendency to go down when you go North). Finally, coefficients close to zero mean that there is no linear correlation. [Figure 2-14](#) shows various plots along with the correlation coefficient between their horizontal and vertical axes.

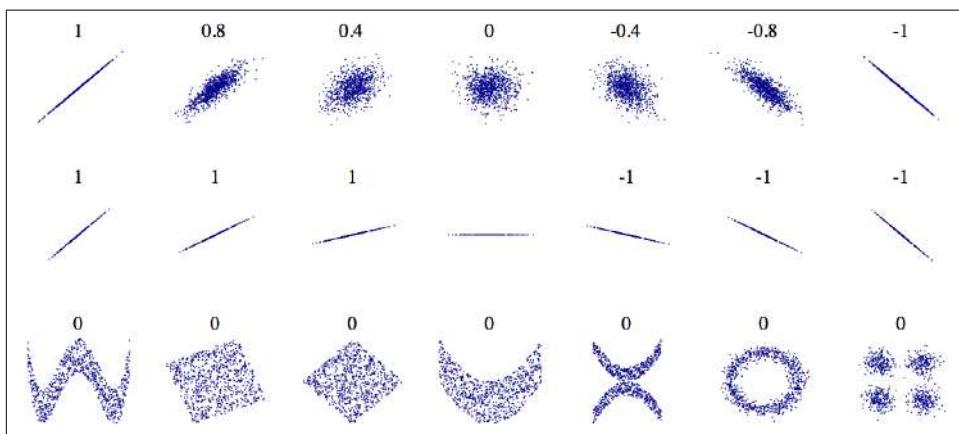


Figure 2-14. Standard Correlation Coefficient of various datasets¹⁶



The correlation coefficient only measures linear correlations (“if x goes up then y generally goes up/down”). It may completely miss out on non-linear relationships (eg. “if x is close to zero then y generally goes up”). Note how all the plots of the bottom row have a correlation coefficient equal to zero despite the fact that their axes are clearly not independent: these are examples of non-linear relationships. Also, the second row shows examples where the correlation coefficient is equal to 1 or -1: notice that this has nothing to do with the slope. For example, your height in inches has a correlation coefficient of 1 with your height in feet or in nanometers.

¹⁶ Source: Wikipedia. Public domain image.

Another way to check for correlation between attributes is to use Pandas' `scatter_matrix` function which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you would get an $11^2=121$ plots, which would not fit on a page, so let's just focus on a few promising attributes that seem most correlated with the median housing value:

```
from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms", "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

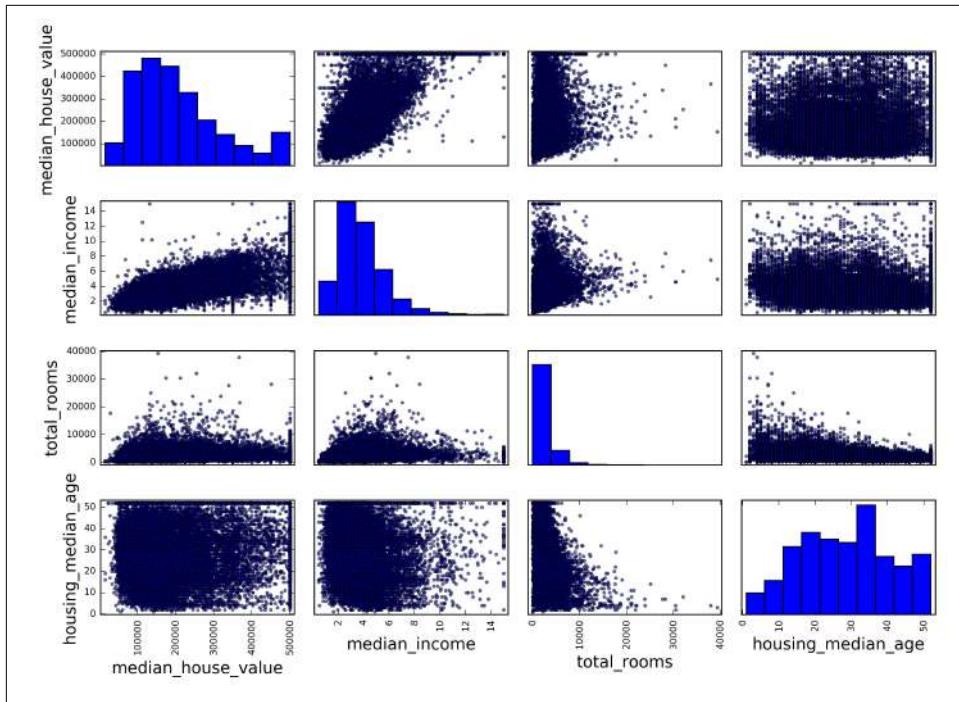


Figure 2-15. Scatter matrix

The main diagonal (top left to bottom right) would be full of straight lines if Pandas plotted each variable against itself: this would not be very useful. So instead Pandas displays a histogram of each attribute (other options are available, see Pandas' documentation for more details).

The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot:

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
alpha=0.1)
```

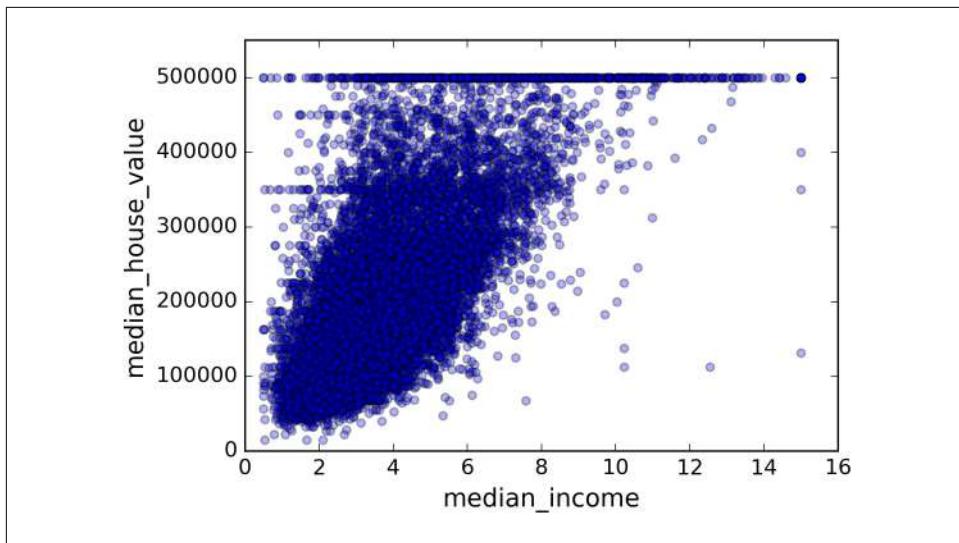


Figure 2-16. Median income vs median house value

This plot reveals a few things: first the correlation is indeed very strong, you can clearly see the upward trend and the points are not too dispersed. Second, the price cap that we noticed earlier is clearly visible as a horizontal line at \$500,000. But this plot reveals other less obvious straight lines: a horizontal line around \$450,000, another around \$350,000, perhaps one around \$280,000 and a few more below that. You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

Experimenting with attribute combinations

Hopefully the previous sections gave you an idea of a few ways you can explore the data and gain insights. You identified a few data quirks that you may want to clean up before feeding the data to a Machine Learning algorithm, you found interesting correlations between attributes, in particular with the target attribute. You also noticed that some attributes have a tail-heavy distribution, so you may want to transform them (eg. by computing their logarithm). Of course your mileage will vary considerably with each project, but the general ideas would be similar.

One last thing you may want to do before actually preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also

seems like an interesting attribute combination to look at. Let's create these new attributes:

```
housing["rooms_per_household"] = housing["total_rooms"] / housing["population"]
housing["bedrooms_per_room"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["population_per_household"] = housing["population"] / housing["households"]
```

And now let's look at the correlation matrix again:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income           0.687170
rooms_per_household    0.199343
total_rooms              0.135231
housing_median_age       0.114220
households               0.064702
total_bedrooms            0.047865
population_per_household -0.021984
population                -0.026699
longitude                 -0.047279
latitude                  -0.142826
bedrooms_per_room         -0.260070
Name: median_house_value, dtype: float64
```

Hey, not bad! The new `bedrooms_per_room` attribute is much more correlated with the median house value than the total number of rooms or bedrooms. Apparently houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district: obviously the larger the houses, the more expensive they are.

This round of exploration does not have to be absolutely thorough: the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype. But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

Prepare the data for Machine Learning algorithms

It's time to prepare the data for your Machine Learning algorithms. Instead of just doing this manually, you should write functions to do that, for several good reasons:

- this will allow you to reproduce these transformations easily on any dataset (eg. next time you get a fresh dataset),
- you will gradually build a library of transformation functions that you can reuse in future projects,
- you can use these functions in your live system to transform the new data before feeding it to your algorithms,

- and finally this will make it possible to easily try various transformations and see which combination of transformations works best.

But first let's revert to a clean training set (by copying `strat_train_set` once again), and let's separate the predictors and the labels since we don't necessarily want to apply the same transformations to the predictors and the target values (note that `drop()` creates a copy of the data, it does not affect `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set[["median_house_value"]].copy()
```

Data cleaning

Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them. You noticed earlier that the `total_bedrooms` attribute has some missing values, so let's fix this. You have three options:

1. you can get rid of the corresponding districts,
2. get rid of the whole attribute,
3. set the values to some value (zero, the mean, the median, etc.).

These can be done easily using DataFrames' `dropna()`, `drop()` and `fillna()` methods:

```
housing.dropna(subset=["total_bedrooms"])      # option 1
housing.drop("total_bedrooms", axis=1)          # option 2
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median)        # option 3
```

If you choose option 3, you should compute the median value on the training set, use it to fill the missing values in the training set, but also don't forget to save the median value that you have computed: you will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.

Scikit-Learn provides a handy class to take care of missing values: `Imputer`. Here is how to use it. First, you need to create an `Imputer` instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")
```

Since the median can only be computed on numerical attributes, we need to create a copy of the data without the text attribute `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(housing_num)
```

The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance variable. Only the `total_bedrooms` attribute had missing values, but cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes:

```
>>> imputer.statistics_
array([-118.51, 34.26, 29., 2119., 433., 1164., 408., 3.5414])
>>> housing_num.median().values
array([-118.51, 34.26, 29., 2119., 433., 1164., 408., 3.5414])
```

Now you can use this “trained” `imputer` to transform the training set by replacing missing values by the learned medians:

```
X = imputer.transform(housing_num)
```

The result is a plain Numpy array containing the transformed features. If you want to put it back into a Pandas DataFrame, it's simple:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

Scikit-Learn design

Scikit-Learn's API is remarkably well designed. The main design principles are¹⁷:

- **Consistency.** All objects share a consistent and simple interface:
 - *Estimators*. Any object that can estimate some parameters based on a dataset is called an *estimator* (eg. an imputer is an estimator). The estimation itself is performed by the `fit()` method, and it takes only a dataset as a parameter (or two for supervised learning algorithms: the second dataset contains the labels). Any other parameter needed to guide the estimation process is considered a hyperparameter (such as an imputer's `strategy`), and it must be set as an instance variable (generally *via* a constructor parameter).
 - *Transformers*. Some estimators (such as an imputer) can also transform a dataset: these are called *transformers*. Once again, the API is quite simple: the transformation is performed by the `transform()` method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for an imputer. All transformers also have a convenience method `fit_transform()` that is

¹⁷ For more details on the design principles, see Buitinck, Louppe, Blondel, Pedregosa, Mueller, *et al.* (2013), <http://goo.gl/wL10sI>

equivalent to calling `fit()` then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

- **Predictors.** Finally some estimators are capable of making predictions given a dataset: they are called *predictors*. For example the `LinearRegression` model in the previous chapter was a predictor: it predicted life satisfaction given a country's GDP per capita. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions given a test set (and the corresponding labels in the case of supervised learning algorithms).¹⁸
- **Inspection:** all the estimator's hyperparameters are accessible directly *via* public instance variables (eg. `imputer.strategy`), and all the estimator's learned parameters are also accessible via public instance variables with an underscore suffix (eg. `imputer.statistics_`).
- **Non-proliferation of classes.** Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of home-made classes. Hyperparameters are just regular python strings or numbers.
- **Composition.** Existing building blocks are reused as much as possible. For example it is easy to create a `Pipeline` estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.
- **Sensible defaults.** Scikit-Learn provides reasonable default values for most parameters, making it easy to create a baseline working system quickly.

Handling text and categorical attributes

Earlier we left out the categorical attribute `ocean_proximity` because it is a text attribute so we cannot compute its median. Most Machine Learning algorithms prefer to work with numbers anyway, so let's convert these text labels to numbers.

Scikit-Learn provides a transformer for this task called `LabelEncoder`:

```
>>> from sklearn.preprocessing import LabelEncoder  
>>> encoder = LabelEncoder()  
>>> housing_cat = housing["ocean_proximity"]  
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)  
>>> housing_cat_encoded  
array([1, 1, 4, ..., 1, 0, 3])
```

¹⁸ Some predictors also provide methods to measure the confidence of their predictions.

This is better: now we can use this numerical data in any ML algorithm. You can look at the mapping that this encoder has learned using the `classes_` attribute (“<1H OCEAN” is mapped to 0, “INLAND” is mapped to 1, etc.):

```
>>> print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. Obviously this is not the case (for example categories 0 and 4 are more similar than categories 0 and 1). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is “<1H OCEAN” (and 0 otherwise), another attribute equal to 1 when the category is “INLAND” (and 0 otherwise), etc. This is called *one-hot encoding*, because only one attribute will be equal to 1 (hot) while the others will be 0 (cold).

Scikit-Learn provides a `OneHotEncoder` encoder to convert integer categorical values into one-hot vectors. Let’s encode the categories as one-hot vectors (note that `fit_transform` expects a 2D array but `housing_cat_encoded` is a 1D array, so we need to reshape it):

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16513x5 sparse matrix of type '<class 'numpy.float64'>'>
      with 16513 stored elements in Compressed Sparse Row format>
```

Notice that the output is a SciPy *sparse matrix*, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories: after one-hot encoding we get a matrix with thousands of columns, and the matrix is full of zeros except for one 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the non-zero elements. You can use it mostly like a normal 2D array¹⁹, but if you really want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()
array([[ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

¹⁹ See SciPy’s documentation for more details.

We can apply both transformations (from text categories to integer categories, then from integer categories to one-hot vectors) in one shot using the `LabelBinarizer` class:

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> encoder = LabelBinarizer()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
array([[0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]])
```

Note that this returns a dense NumPy array by default. You can get a sparse matrix instead by passing `sparse_output=True` to the `LabelBinarizer` constructor.

Custom transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), all you need is to create a class and implement three methods: `fit()` (returning `self`), `transform()` and `fit_transform()`. You can get the last one for free by simply adding `TransformerMixin` as a base class. Also, if you add `BaseEstimator` as a base class (and avoid `*args` and `**kargs` in your constructor) you will get two extra methods (`get_params()` and `set_params()`) that will be useful for automatic hyperparameter tuning. For example, here is a small transformer class that adds the combined attributes we discussed earlier:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, room_ix]
            return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]
        else:
```

```
return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

In this example the transformer has one hyperparameter `add_bedrooms_per_room`, set to `True` by default (it is often helpful to provide sensible defaults). This hyperparameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not. More generally, you can add a hyperparameter to gate any data preparation step that you are not 100% sure about. The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely that you will find a great combination (and saving you a lot of time).

Feature scaling

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms range from about 6 to 39,320, while the median incomes only range from 0 to 15. Note that scaling the target values is generally not required.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

Min-max scaling (many people call this *normalization*) is quite simple: values are shifted and rescaled so that they end up ranging from 0 to 1. This is done by subtracting the min value and dividing by the max minus the min. Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the range if you don't want 0–1 for some reason.

Standardization is quite different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the variance so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (eg. neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers: for example, suppose a district had a median income equal to 100 (by mistake), then min-max scaling would crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called `StandardScaler` for standardization.



As with all the transformations, it is important to fit the scalers to the training data only, not to the full dataset (including the test set). Only then can you use them to transform the training set and the test set (and new data).

Transformation pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations. Here is a small pipeline for the numerical attributes:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

The `Pipeline` constructor takes a list of name/estimator pairs defining a sequence of steps. All but the last estimator must be transformers (ie. they must have a `fit_transform()` method). The names can be anything you like.

When you call the pipeline's `fit()` method, it calls `fit_transform()` sequentially on all transformers, passing the output of each call as the parameter to the next call, until it reaches the final estimator, for which it just calls the `fit()` method.

The pipeline exposes the same methods as the final estimator. In this example, the last estimator is a `StandardScaler` which is a transformer, so the pipeline has a `transform()` method which applies all the transforms to the data in sequence (it also has a `fit_transform` method that we could have used instead of calling `fit()` then `transform()`).

You now have a pipeline for numerical values, and you also need to apply the `LabelBinarizer` on the categorical values: how can you join these transformations into a single pipeline? Scikit-Learn provides a `FeatureUnion` class for this: you give it a list of transformers (which can be entire transformer pipelines), and when its `transform()` method is called it runs each transformer's `transform()` method in parallel, waits for their output, then concatenates them and returns the result (and of course calling its `fit()` method calls all each transformer's `fit()` method). A full pipeline handling both numerical and categorical attributes may look like this:

```

from sklearn.pipeline import FeatureUnion

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])
cat_pipeline = Pipeline([
    ('selector', DataFrameSelector(cat_attribs)),
    ('label_binarizer', LabelBinarizer()),
])
full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline),
])

```

And you can run the whole pipeline simply:

```

>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ 0.73225807, -0.67331551,  0.58426443, ...,  0.        ,
         0.        ,  0.        ],
       [-0.99102923,  1.63234656, -0.92655887, ...,  0.        ,
         0.        ,  0.        ],
       [...]
>>> housing_prepared.shape
(16513, 17)

```

Each sub-pipeline starts with a selector transformer: it simply transforms the data by selecting the desired attributes (numerical or categorical), dropping the rest and converting the resulting DataFrame to a NumPy array. There is nothing in Scikit-Learn to handle Pandas DataFrames²⁰ so we need to write a simple custom transformer for this task:

```

from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self

```

²⁰ But check out Pull Request #3886 which may introduce a `ColumnTransformer` class making attribute-specific transformations easy. You could also run `pip3 install sklearn-pandas` to get a `DataFrameMapper` class with similar objective.

```
def transform(self, X):
    return X[self.attribute_names].values
```

Select and train a model

At last! You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote transformation pipelines to clean up and prepare your data for machine learning algorithms automatically. You are now ready to select and train a Machine Learning model.

Training and evaluating on the training set

The good news is that thanks to all these previous steps things are now going to be much simpler than you might think. Let's first train a Linear Regression model, like we did in the previous chapter.

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Done! You now have a working Linear Regression model. Let's try it out on a few instances from the training set:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = preparation_pipeline.transform(some_data)
>>> print("Predictions:\t", lin_reg.predict(some_data_prepared))
Predictions: [ 303104.  44800.  308928.  294208.  368704.]
>>> print("Labels:\t", list(some_labels))
Labels: [359400.0, 69700.0, 302100.0, 301300.0, 351900.0]
```

It works, although the predictions are not exactly accurate (eg. the second prediction is off by more than 50%!). Let's measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error` function:

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.413493824875
```

Ok, this is better than nothing but clearly not a great score: most district's `median_housing_values` range between \$120,000 and \$265,000, so a typical prediction error of \$68,628 is not very satisfying. This is an example of a model underfitting the training data. When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough. As we saw in the previous chapter, the main ways to fix underfitting are to

select a more powerful model, to feed the training algorithm with better features or to reduce the constraints on the model. This model is not regularized, so this rules out the last option. You could try to add more features (eg. the log of the population), but first let's try a more complex model to see how it does.

Let's train a `DecisionTreeRegressor`: this is a powerful model, capable of finding complex non-linear relationships in the data (decision trees are presented in more detail in [Chapter 6](#)). The code should look familiar by now:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

Now that the model is trained, let's evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course it is much more likely that the model has badly overfit the data. How can you be sure? As we saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training, and part for model validation.

Better evaluation using cross validation

One way to evaluate the decision tree model would be to use the `train_test_split` function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set. A bit of work, but nothing too difficult and it would work fairly well.

A great alternative is to use Scikit-Learn's *cross-validation* feature. The following code performs *K-fold cross-validation*: it randomly splits the training set into 10 distinct subsets called *folds*, then it trains and evaluates the decision tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

```
from sklearn.cross_validation import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                        scoring="mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)
```



Scikit-Learn cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the MSE (ie. a negative value), which is why the code above computes `-scores` before calculating the square root.

Let's look at the results:

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 74678.4916885   64766.2398337   69632.86942005  69166.67693232
         71486.76507766   73321.65695983  71860.04741226  71086.32691692
         76934.2726093   69060.93319262]
Mean: 71199.4280043
Standard deviation: 3202.70522793
```

Now the decision tree doesn't look as good as it did earlier. In fact, it seems to perform worse than the Linear Regression model! Notice that cross-validation allows you to get not only an estimate of the performance of your model, but it also gives a measure of how precise this estimate is (ie. its standard deviation): the decision tree has a score of approx. 71,200, generally $\pm 3,200$. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always possible.

Let's compute the same scores for the Linear Regression model just to be sure:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                                 scoring="mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 70423.5893262   65804.84913139   66620.84314068  72510.11362141
         66414.74423281   71958.89083606  67624.90198297  67825.36117664
         72512.36533141   68028.11688067]
Mean: 68972.377566
Standard deviation: 2493.98819069
```

That's right, the decision tree model is overfitting so badly that it performs worse than the Linear Regression model.

Let's try one last model now: the `RandomForestRegressor`. As we will see in [Chapter 7](#), Random Forests work by training many decision trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models is called *Ensemble Learning* and it is often a great way to push ML algorithms even further. We will skip most of the code since it is essentially the same as for the other models:

```

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
22542.396440343684
>>> display_scores(forest_rmse_scores)
Scores: [ 53789.2879722   50256.19806622   52521.55342602   53237.44937943
          52428.82176158   55854.61222549   52158.02291609   50093.66125649
          53240.80406125   52761.50852822]
Mean: 52634.1919593
Standard deviation: 1576.20472269

```

Wow, this is much better, Random Forests look very promising. However, note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. Possible solutions for overfitting are to simplify the model, constrain it (ie. regularize it), or get a lot more training data. However, before you dive much deeper in Random Forests, you should try out many other models from various categories of Machine Learning algorithms (eg. several SVMs with different kernels, possibly a neural network, etc.), without spending too much time tweaking the hyperparameters: the goal is to short-list a few (2-5) promising models.



You should save every model you experiment with, so you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps also the actual predictions as well. This will allow you to easily compare scores across model types, and compare the types of errors they make. Scikit-Learn models can easily be saved using python's pickle module, or using `sklearn.externals.joblib` which is more efficient at serializing large NumPy arrays:

```

from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")

```

Fine-tune your model

Grid search

Let's assume that you now have a short-list of promising models. You now need to fine-tune them. One way to do that would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead you should get Scikit-Learn's `GridSearchCV` to search for you. All you need to do is tell it which hyperparameters you want it to experiment with, and what values to try out, and it will evaluate all the possible combinations of hyperparameter values, evaluating each combination using cross-validation. For example, here the following code searches for the best combination of hyperparameters values for the `RandomForestRegressor`:

```
from sklearn.grid_search import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```



When you have no idea what value a hyperparameter should have, a simple approach is to try out consecutive powers of 10 (or a smaller number if you want a more fine-grained search, as shown in this example with the `n_estimators` hyperparameter).

This `param_grid` tells Scikit-Learn to first evaluate all $3 \times 4 = 12$ combinations of `n_estimators` and `max_features` hyperparameter values specified in the first dict (don't worry about what these hyperparameters mean for now, they will be explained in [Chapter 7](#)), then try all $2 \times 3 = 6$ combinations of hyperparameter values in the second dict, but this time with the `bootstrap` hyperparameter set to `False` instead of `True` (which is the default value for this hyperparameter).

All in all, the grid search will explore $12 + 6 = 18$ combinations of `RandomForestRegressor` hyperparameter values, and it will train each model 5 times (since we are using 5-fold cross validation). In other words, all in all there will be $18 \times 5 = 90$ rounds of training! It may take quite a long time, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'max_features': 6, 'n_estimators': 30}
```



Since 30 is the maximum value of `n_estimators` that was evaluated, you should probably evaluate higher values as well, since the score may continue to improve.

You can also get the best estimator directly:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features=6, max_leaf_nodes=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=30, n_jobs=1, oob_score=False, random_state=None,
                      verbose=0, warm_start=False)
```



If GridSearchCV is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross validation, it re-trains it on the whole training set. This is usually a good idea since feeding it more data will likely improve its performance.

And of course the evaluation scores are also available:

```
>>> for params, mean_score, scores in grid_search.grid_scores_:
...     print(np.sqrt(-mean_score), np.sqrt(-scores).std(), params)
...
65432.9 1062.7 {'max_features': 2, 'n_estimators': 3}
55665.5 1427.4 {'max_features': 2, 'n_estimators': 10}
53033.7 780.7 {'max_features': 2, 'n_estimators': 30}
60781.3 1399.7 {'max_features': 4, 'n_estimators': 3}
52658.6 1434.2 {'max_features': 4, 'n_estimators': 10}
50483.3 834.7 {'max_features': 4, 'n_estimators': 30}
59411.7 1335.7 {'max_features': 6, 'n_estimators': 3}
52418.5 950.0 {'max_features': 6, 'n_estimators': 10}
49996.3 809.8 {'max_features': 6, 'n_estimators': 30}
58637.5 1277.2 {'max_features': 8, 'n_estimators': 3}
52265.4 991.7 {'max_features': 8, 'n_estimators': 10}
50113.1 863.1 {'max_features': 8, 'n_estimators': 30}
61837.3 1148.0 {'n_estimators': 3, 'max_features': 2, 'bootstrap': False}
54472.3 833.1 {'n_estimators': 10, 'max_features': 2, 'bootstrap': False}
60150.4 912.7 {'n_estimators': 3, 'max_features': 3, 'bootstrap': False}
52588.9 643.1 {'n_estimators': 10, 'max_features': 3, 'bootstrap': False}
58815.7 989.0 {'n_estimators': 3, 'max_features': 4, 'bootstrap': False}
52010.3 1102.9 {'n_estimators': 10, 'max_features': 4, 'bootstrap': False}
```

In this example, the best solution is obtained by setting the `max_features` hyperparameter to 6, and the `'n_estimators'` hyperparameter to 30. The RMSE score for this combination is 49,996.3, which is slightly better than the score you got earlier using the default hyperparameter values (which was 52,634). Congratulation, you have successfully fine-tuned your best model!



Don't forget that you can treat some of the data preparation steps as hyperparameters: for example, the grid search will automatically find out whether or not to add a feature you were not sure about (eg. using the `add_bedrooms_per_room` hyperparameter of your `CombinedAttributesAdder` transformer). It may similarly be used to automatically find the best way to handle outliers, missing features, feature selection, etc.

Randomized search

The grid search approach is fine when you are exploring relatively few combinations, like in the example above, but when the hyperparameter *search space* is large, it is often preferable to use `RandomizedSearchCV` instead. This class can be used in much the same way as the `GridSearchCV` class, but instead of trying out all possible combinations, it evaluates a given number of random combinations, by selecting a random value for each hyperparameter at every iteration. This approach has two main benefits:

- first, if you let the randomized search run for say 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter (instead of just a few values per hyperparameter with the grid search approach).
- second, you have more control over the computing budget you want to allocate to hyperparameter search, simply by setting the number of iterations.

Ensemble methods

Another way to fine-tune your system is to try to combine the models that perform best: the group (or “ensemble”) will often perform better than the best individual model (just like Random Forests perform better than the individual decision trees they rely on), especially if the individual models make very different types of errors. We will cover this topic in more detail in [Chapter 7](#).

Analyze the best models and their errors

You will often gain good insights on the problem by inspecting the best models. For example, the `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions.

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.14156423e-02,   6.76139189e-02,   4.44260894e-02,
       1.66308583e-02,   1.66076861e-02,   1.82402545e-02,
       1.63458761e-02,   3.26497987e-01,   6.04365775e-02,
       1.13055290e-01,   7.79324766e-02,   1.12166442e-02,
```

```
1.53344918e-01,    8.41308969e-05,    2.68483884e-03,  
3.46681181e-03])
```

Let's display these importance scores next to their corresponding attribute names:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]  
>>> cat_one_hot_attribs = list(encoder.classes_)  
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs  
>>> sorted(zip(feature_importances, attributes), reverse=True)  
[(0.32649798665134971, 'median_income'),  
(0.15334491760305854, 'INLAND'),  
(0.11305529021187399, 'pop_per_hhold'),  
(0.07793247662544775, 'bedrooms_per_room'),  
(0.071415642259275158, 'longitude'),  
(0.067613918945568688, 'latitude'),  
(0.060436577499703222, 'rooms_per_hhold'),  
(0.04442608939578685, 'housing_median_age'),  
(0.018240254462909437, 'population'),  
(0.01663085833886218, 'total_rooms'),  
(0.016607686091288865, 'total_bedrooms'),  
(0.016345876147580776, 'households'),  
(0.011216644219017424, '<1H OCEAN'),  
(0.0034668118081117387, 'NEAR OCEAN'),  
(0.0026848388432755429, 'NEAR BAY'),  
(8.4130896890070617e-05, 'ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (eg. apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (eg. adding extra features or on the contrary getting rid of uninformative ones, cleanup outliers, etc.).

Evaluate your system on the test set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to evaluate the final model on the test set. There is nothing special about this process, just get the predictors and the labels from your test set, run your `preparation_pipeline` to transform the data (call `transform()`, *not* `fit_transform()!`), and evaluate the final model on the test set.

```
final_model = grid_search.best_estimator_  
  
X_test = strat_test_set.drop("median_house_value", axis=1)  
y_test = strat_test_set["median_house_value"].copy()  
  
X_test_prepared = preparation_pipeline.transform(X_test)  
  
final_predictions = final_model.predict(X_test_prepared)
```

```
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse) # => evaluates to 48,209.6
```

The performance will usually be slightly worse than what you measured using cross validation if you did a lot of hyperparameter tuning (because your system ends up fine-tuned to perform well on the validation data, and will likely not perform as well on unknown datasets). It is not the case in this example, but when this happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set: the improvements would be unlikely to generalize to new data.

Now comes the project pre-launch phase: you need to present your solution, highlighting what you have learned, what worked and what did not, what assumptions were made and what your system's limitations are, document everything, create nice presentations with clear visualizations and easy to remember statements (eg. "the median income is the number one predictor of housing prices"), etc.

Launch, monitor and maintain your system

Perfect, you got approval to launch! You need to get your solution ready for production, in particular by plugging the production input data sources to your system, write tests, etc.

You also need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops. This is important not only to catch sudden breakage, but also to catch slow performance degradation. This is quite common because models tend to "rot" as data evolves over time, unless the models are regularly trained on fresh data.

Evaluating your system's performance will require sampling the system's predictions and evaluating them. This will generally require a human analysis. These analysts may be field experts, or workers on a crowdsourcing platform (such as Amazon Mechanical Turk or CrowdFlower). Either way, you need to plug in the human evaluation pipeline in your system.

You should also make sure you evaluate the system's input data quality. Sometimes performance will degrade slightly because of a poor quality signal (eg. a malfunctioning sensor sending random values, or another team's output becoming stale), but it may take a while before your system's performance degrades enough to trigger an alert. If you monitor your system's inputs, you may catch this earlier. Monitoring the inputs is particularly important for online learning systems.

Finally, you will generally want to train your models on a regular basis using fresh data. You should automate this process as much as possible. If you don't, you are very likely to refresh your model only every 6 months (at best), and your system's performance may fluctuate severely over time. If your system is an online learning system,

you should make sure you save snapshots of its state at regular interval so you can easily roll back to a previously working state.

Try it out!

Hopefully this chapter gave you a good idea of what a Machine Learning project looks like, and showed you some of the tools you can use to train a great system. As you can see, much of the work is in the data preparation step, building monitoring tools, setting up human evaluation pipelines, and automating regular model training. The Machine Learning algorithms are also important, of course, but it is probably preferable to be comfortable with the overall process and know 3 or 4 algorithms well rather than to spend all your time exploring advanced algorithms and not enough time on the overall process.

So if you have not already done so, now is a good time to pick up a laptop, select a dataset that you are interested in, and try to go through the whole process from A to Z. A good place to start is on a competition website such as <http://kaggle.com/>: you will have a dataset to play with, a clear goal and people to share the experience with.

Exercises

Using this chapter's housing dataset:

1. Try a Support Vector Machine Regressor (`sklearn.svm.SVR`), with various hyperparameters such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best SVR predictor perform?
2. Try replacing the `GridSearchCV` with a `RandomizedSearchCV`.
3. Try adding a transformer in the preparation pipeline to select only the most important attributes.
4. Try creating a single pipeline that does the full data preparation plus the final prediction.
5. Automatically explore some preparation options using `GridSearchCV`.

Solutions to these exercises are available in the online Jupyter notebooks at <https://github.com/ageron/handson-ml>.

CHAPTER 3

Classification

In [Chapter 1](#) we mentioned that the most common supervised learning tasks are regression (predicting values) and classification (predicting classes). In [Chapter 2](#) we explored a regression task, predicting housing values, using various algorithms such as Linear Regression, Decision Trees and Random Forests (which will be explained in further detail in later chapters). Now we will turn our attention to classification systems.

MNIST

In this chapter, we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the U.S. Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the “Hello World” of Machine Learning: whenever people come up with a new classification algorithm, they are curious to see how it will perform on MNIST. Whenever someone learns Machine Learning, sooner or later they tackle MNIST.

Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them. The following code fetches the MNIST dataset¹:

```
>>> from sklearn.datasets import fetch_mldata  
>>> mnist = fetch_mldata('MNIST original')  
>>> mnist  
{'COL_NAMES': ['label', 'data'],  
 'DESCR': 'mldata.org dataset: mnist-original',  
 'data': array([[0, 0, 0, ..., 0, 0, 0],  
 [0, 0, 0, ..., 0, 0, 0],
```

¹ By default Scikit-Learn caches downloaded datasets in a directory called `$HOME/scikit_learn_data`.

```
[0, 0, 0, ..., 0, 0, 0],  
...,  
[0, 0, 0, ..., 0, 0, 0],  
[0, 0, 0, ..., 0, 0, 0],  
[0, 0, 0, ..., 0, 0, 0]], dtype=uint8),  
'target': array([ 0.,  0.,  0., ...,  9.,  9.,  9.])}
```

Datasets loaded by Scikit-Learn generally have a similar dictionary structure:

- a `DESCR` key describing the dataset,
- a `data` key containing an array with one row per instance and one column per feature,
- and a `target` key containing an array with the labels.

Let's look at these arrays:

```
>>> X, y = mnist["data"], mnist["target"]  
>>> X.shape  
(70000, 784)  
>>> y.shape  
(70000,)
```

There are 70,000 images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). Let's take a peek at one digit from the dataset. All you need to do is grab an instance's feature vector, reshape it to a 28×28 array, and display it using Matplotlib's `imshow()` function:

```
%matplotlib inline  
import matplotlib  
import matplotlib.pyplot as plt  
  
some_digit = X[36000]  
some_digit_image = some_digit.reshape(28, 28)  
  
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,  
           interpolation="nearest")  
plt.axis("off")  
plt.show()
```



This looks like a 5, and indeed that's what the label tells us:

```
>>> y[36000]  
5.0
```

Figure 3-1 shows a few more images from the MNIST dataset to give you a feel of the complexity of the classification task:

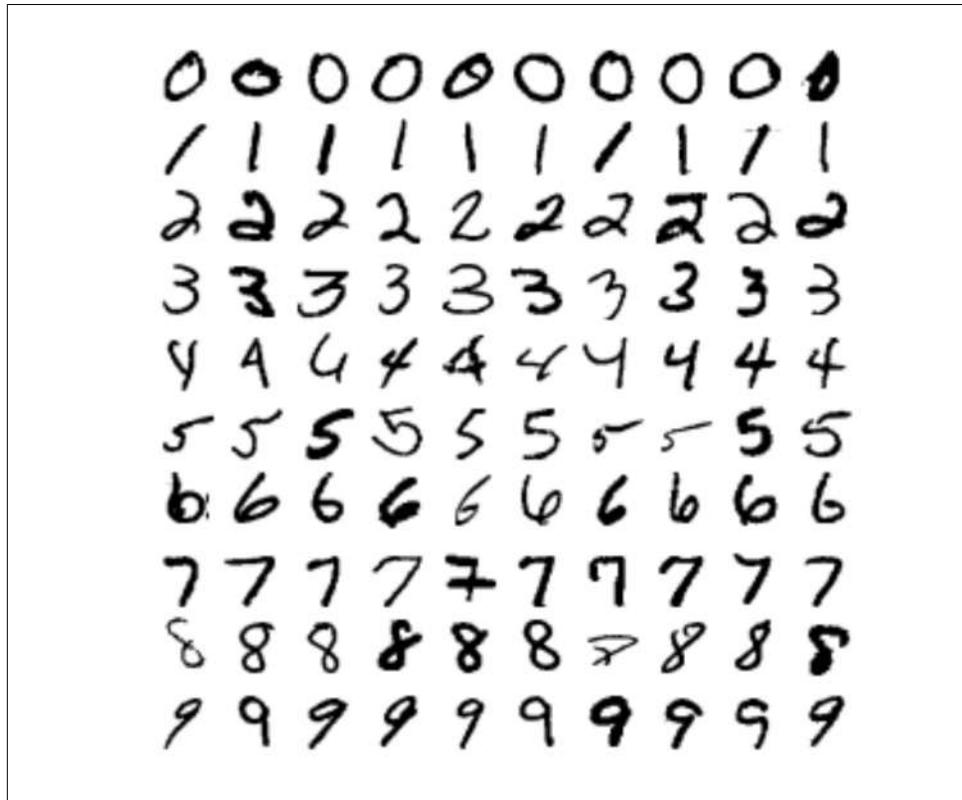


Figure 3-1. A few digits from the MNIST dataset

But wait! You should always create a test set and set it aside before inspecting the data closely. The MNIST dataset is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Let's also shuffle the training set: this will guarantee that all cross validation folds will be similar (you don't want one fold to be missing some digits). Moreover, some learning algorithms are sensitive to the order of the training instances and they perform

poorly if they get many similar instances in a row: shuffling the dataset ensures that this won't happen².

```
import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

Training a binary classifier

Let's simplify the problem for now and only try to identify one digit, for example the number 5. This “5-detector” will be an example of a *binary classifier*, capable of distinguishing between just two classes, 5 and not-5. Let's create the target vectors for this classification task:

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.
y_test_5 = (y_test == 5)
```

Ok, now let's pick a classifier and train it. A good place to start is with a *Stochastic Gradient Descent* classifier, using Scikit-Learn's `SGDClassifier` class. This classifier has the advantage of being capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time (which also makes SGD well suited for *online learning*), as we will see later. Let's create an `SGDClassifier` and train it on the whole training set:

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```



The `SGDClassifier` relies on randomness during training (hence the name “stochastic”). If you want reproducible results, you should set the `random_state` parameter.

Now you can use it to detect images of the number 5:

```
>>> sgd_clf.predict([some_digit])
array([ True], dtype=bool)
```

The classifier guesses that this image represents a 5 (`True`). Looks like it guessed right in this particular case! Now, let's evaluate this model's performance.

² Shuffling may be a bad idea in some contexts, for example if you are working on time series data (such as stock market prices or weather conditions). We will explore this in the next chapters.

Performance measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn many new concepts and acronyms!

Measuring accuracy using cross-validation

Just like you did in [Chapter 2](#), a good way to evaluate a model is to use cross-validation.

Implementing cross-validation

Occasionally you will need to have more control over the cross-validation process than what `cross_val_score()` and similar functions provide. In these cases, you can implement cross-validation yourself, it is actually fairly straightforward. The following code does roughly the same thing as the `cross_val_score()` code above, and prints the same result:

```
from sklearn.cross_validation import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(y_train_5, n_folds=3, random_state=42)

for train_index, test_index in skfolds:
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = (y_train_5[train_index])
    X_test_fold = X_train[test_index]
    y_test_fold = (y_train_5[test_index])

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # prints 0.9502, 0.96565 and 0.96495
```

The `StratifiedKFold` class performs stratified sampling (as explained in [Chapter 2](#)) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

Let's use the `cross_val_score()` function to evaluate your `SGDClassifier` model using K-fold cross-validation, with 3 folds. Remember that K-fold cross-validation means splitting the training set into K folds (in this case 3), then making predictions

and evaluating them on each fold using a model trained on the remaining folds (see [Chapter 2](#)):

```
>>> from sklearn.cross_validation import cross_val_score
>>> cross_val_score(sgd_clf, X_train_5, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 ,  0.96565,  0.96495])
```

Wow! Above 95% *accuracy* (ratio of correct predictions) on all cross validation folds? This looks amazing, doesn't it? Well before you get too excited, let's look at a very dumb classifier that just classifies every single image in the "not-5" class:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

Can you guess this model's accuracy? Let's find out:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909 ,  0.90715,  0.9128 ])
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time. Beats Nostradamus.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when dealing with *skewed datasets* (ie. when some classes are much more frequent than others).

Confusion matrix

A much better way to evaluate the performance of a classifier is to look at the *confusion matrix*. The general idea is to count the number of times instances of class A are classified as class B. For example, to know the number of times the classifier confused images of 5s with 3s, you would in the 5th row and 3rd column of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions, so they can be compared to the actual targets. You could make predictions on the test set, but let's keep it untouched for now (remember that you want to use the test set only at the very end of your project, once you have a classifier that you are ready to launch). Instead, you can use the `cross_val_predict()` function:

```
from sklearn.cross_validation import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Just like the `cross_val_score()` function, `cross_val_predict()` performs K-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set (“clean” meaning that the prediction is made by a model that never saw the data during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix  
>>> confusion_matrix(y_train_5, y_train_pred)  
array([[53272,  1307],  
       [ 1077,  4344]])
```

Each row in a confusion matrix represents an *actual class* while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,272 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 1,307 were wrongly classified as 5s (*false positives*). The second row considers the images of 5s (the *positive class*): 1,077 were wrongly classified as non-5s (*false negatives*), while the remaining 4,344 were correctly classified as 5s (*true positives*). A perfect classifier would have only true positives and true negatives, so its confusion matrix would have non-zero values only on its main diagonal (top left to bottom right):

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)  
array([[54579,     0],  
       [     0, 5421]])
```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions: this is called the *precision* of the classifier.

Equation 3-1. Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

TP is the number of True Positives, and FP is the number of False Positives.

A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct (precision = 1/1 = 100%). This would not be very useful since the classifier would ignore all but one positive instance. So precision is typically used along with another metric named *recall*, also called *sensitivity* or *True Positive Rate*.

(*TPR*): this is the ratio of positive instances that are correctly detected by the classifier.

Equation 3-2. Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN is of course the number of False Negatives.

If you are confused about the confusion matrix, Figure 3-2 may help.

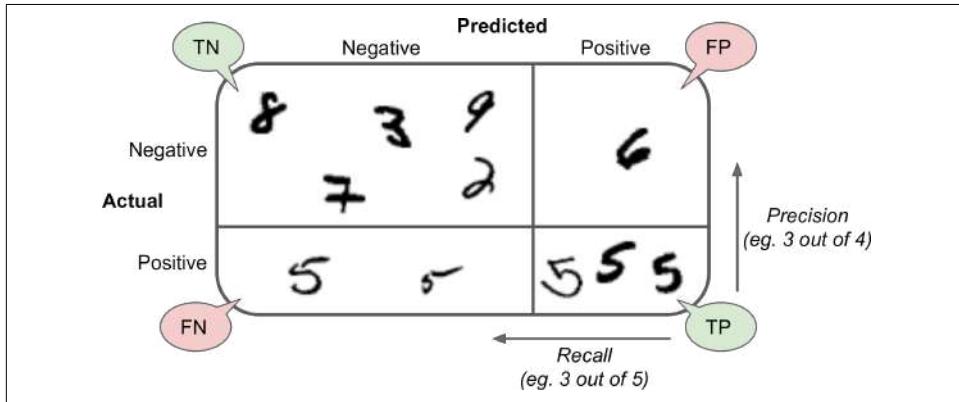


Figure 3-2. An illustrated confusion matrix

Precision and recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_pred)      # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred)    # == 4344 / (4344 + 1077)
0.79136690647482011
```

Now your 5-detector does not look as shiny as it did when you looked at its accuracy: when it claims an image represents a 5, it is correct only 77% of the time. Moreover, it only detects 79% of the 5s.

It is often convenient to combine precision and recall into a single metric called the *F₁ score*, in particular if you need a simple way to compare two classifiers. The F₁ score is the *harmonic mean* of precision and recall. Whereas the regular mean treats all values

equally, the harmonic mean gives much more weight to low values. As a result the classifier will only get a high F_1 score if both recall and precision are high.

Equation 3-3. F_1 score

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

To compute the F_1 score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score  
>>> f1_score(y_train_5, y_pred)  
0.78468208092485547
```

The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example if you train a classifier to detect videos that are safe for kids, you will probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters on surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).

Unfortunately, you can't have it both ways: increasing precision reduces recall, and *vice versa*. This is called the *precision/recall tradeoff*.

Precision/Recall tradeoff

To understand this tradeoff, let's look at how the `SGDClassifier` makes its classification decisions: for each instance, it computes a score based on a *decision function*, and if that score is greater than a threshold, it assigns the instance to the positive class, or else it assigns it to the negative class. [Figure 3-3](#) shows a few digits positioned from the lowest score on the left to the highest score on the right. Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual 5s) on the right of that threshold, and one false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). Now if you raise the threshold (move it to the arrow on the right), the false positive (the 6) now becomes a true negative, thereby increasing precision (up to 100% in this case), but one true positive now becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

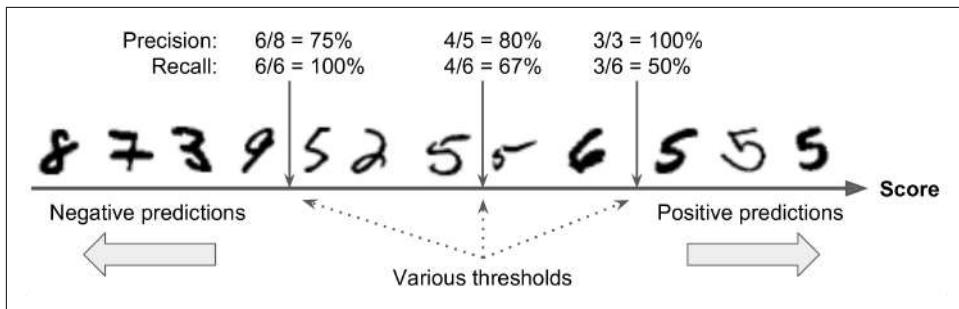


Figure 3-3. Decision threshold and precision/recall tradeoff

Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions: instead of calling the classifier's `predict()` method, you can call its `decision_function()` method which returns a score for each instance, and then make predictions based on those scores using any threshold you want:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([ 161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

The `SGDClassifier` uses a threshold equal to 0, so the above code returns the same result as the `predict()` method (ie. `True`). Let's raise the threshold:

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

This confirms that raising the threshold decreases recall: the image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 200,000.

So how can you decide which threshold to use? For this you will first need to get the scores of all instances in the training set using the `cross_val_predict()` function again, but this time specifying that you want it to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Now with these scores you can compute precision and recall for all possible thresholds using the `precision_recall_curve()` function:

```

from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)

```

Finally, you can plot precision and recall as functions of the threshold value using Matplotlib:

```

def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()

```

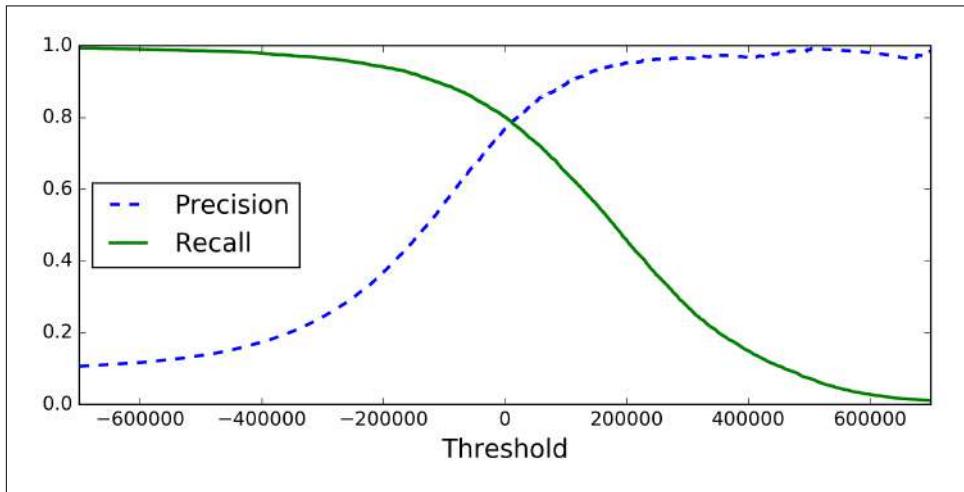


Figure 3-4. Precision and recall vs the decision threshold



You may wonder why the precision curve is bumpier than the recall curve in Figure 3-4. The reason for this is that precision may sometimes go down when you raise the threshold (although in general it will go up). To understand why, look back at Figure 3-3 and notice what happens when you start from the central threshold and you move it just one digit to the right: precision goes from 4/5 (80%) down to 3/4 (75%). On the other hand, recall can only go down when the threshold is increased, which explains why its curve looks smooth.

Now you can simply select the threshold value that gives you the best precision/recall tradeoff for your task. Another way to select a good precision/recall tradeoff is to plot precision directly against recall, as shown in Figure 3-5:

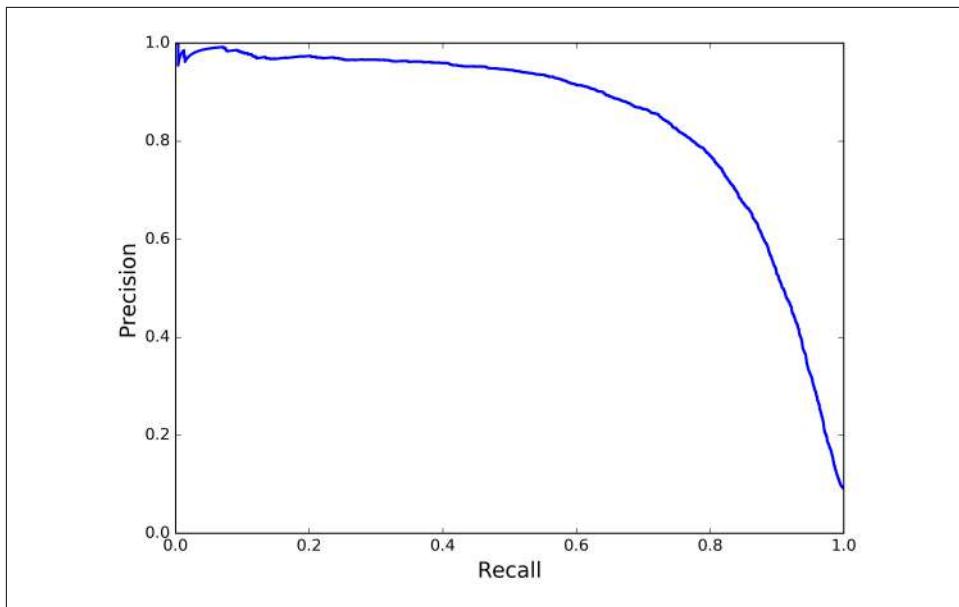


Figure 3-5. Precision vs recall

You can see that precision really starts to fall sharply around 80% recall. You will probably want to select a precision/recall tradeoff just before that drop, for example at around 60% recall. But of course the choice depends on your project.

So let's suppose you decide to aim for 90% precision. You look up the first plot (zooming in a bit) and you find that you need to use a threshold of about 70,000. To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can just run this code:

```
y_train_pred_90 = (y_scores > 70000)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.8998702983138781
>>> recall_score(y_train_5, y_train_pred_90)
0.63991883416343853
```

Great, you have a 90% precision classifier (or close enough)! As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high enough threshold, and you're done. Hmm, not so fast. A high-precision classifier is not very useful if its recall is too low!



If someone says “let’s reach 99% precision”, you should ask “at what recall?”.

The ROC curve

The *Receiver Operating Characteristic* (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision *vs* recall, the ROC curve plots the *True Positive Rate* (another name for recall) against the *False Positive Rate*. The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the *True Negative Rate*, which is the ratio of negative instances that are correctly classified as negative. The TNR is also called *specificity*. Hence the ROC curve plots *sensitivity* (recall) *vs* 1-*specificity*.

To plot the ROC curve, you first need to compute the TPR and FPR for various threshold values, using the `roc_curve()` function:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Then you can plot the FPR against the TPR using Matplotlib. This code produces the plot in [Figure 3-6](#):

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr, tpr)
plt.show()
```

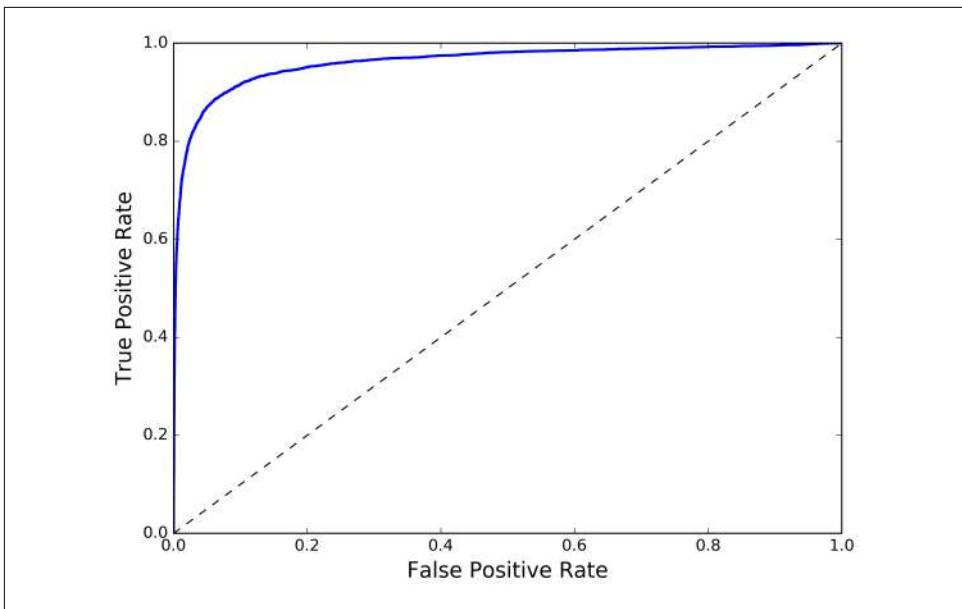


Figure 3-6. ROC curve

Once again there is a tradeoff: the higher the recall (TPR), the more false positives (FPR) the classifier produces. The dotted line represents the ROC curve of a purely random classifier: a good classifier stays as far away from that line as possible (towards the top left corner).

One way to compare classifiers is to measure the *Area Under the Curve* (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to compute the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score  
>>> roc_auc_score(y_train_5, y_scores)  
0.97061072797174941
```



Since the ROC curve is so similar to the Precision/Recall (or PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives, and the ROC curve otherwise. For example, looking at the ROC curve above (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement (the curve could be closer to the top right corner).

Let's train a `RandomForestClassifier` and compare its ROC curve and ROC AUC score to the `SGDClassifier`. First, you need to get scores for each instance in the training set. But due to the way it works (see [Chapter 7](#)), the `RandomForestClassifier` class does not have a `decision_function()` method. Instead it has a `predict_proba()` method. Scikit-Learn classifiers generally have one or the other. The `predict_proba()` method returns an array containing a row per instance and a column per class, each containing the probability that the given instance belongs to the given class (eg. 70% chance that the image represents a 5).

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

But to plot a ROC Curve, you need scores, not probabilities. A simple solution is to use the positive class's probability as the score:

```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

Now you are ready to plot the ROC curve. It is useful to plot the first ROC curve as well to see how they compare:

```
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="bottom right")
plt.show()
```

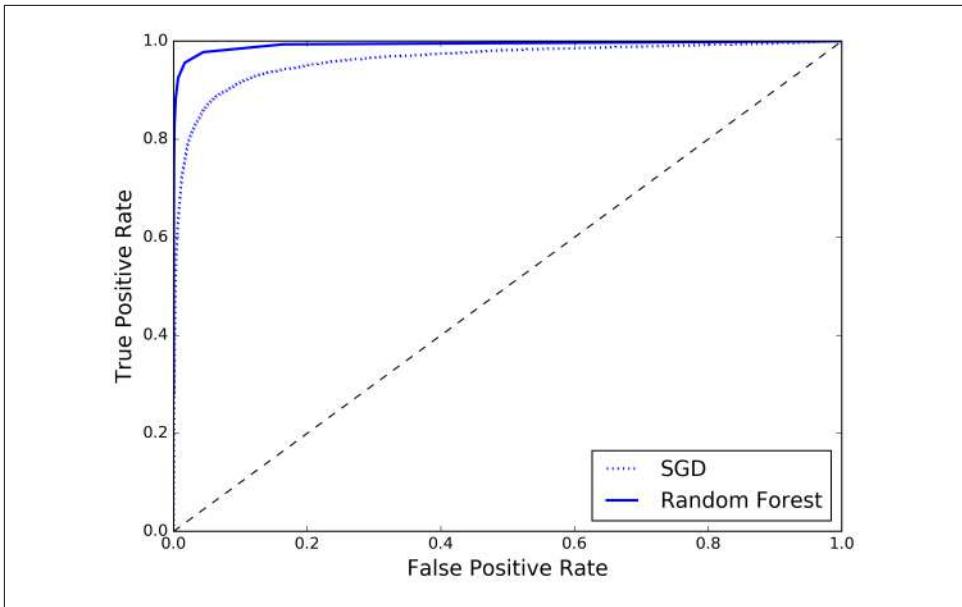


Figure 3-7. Comparing ROC curves

As you can see in Figure 3-7, the ‘RandomForestClassifier’s ROC curve looks much better than the ‘SGDClassifier’s: it comes much closer to the top left corner. As a result, its ROC AUC score is also significantly better:

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.99312433660038291
```

Try measuring the precision and recall scores: you should find 98.5% precision and 82.8% recall. Not too bad!

Hopefully you now know how to train binary classifiers, choose the appropriate metric for your task, evaluate your classifiers using cross-validation, select the precision/recall tradeoff that fits your needs and compare various models using ROC curves and ROC AUC scores. Now let’s try to detect more than just the 5s.

Multiclass classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some algorithms (such as Random Forest classifiers or naive Bayes classifiers) are capable of handling multiple classes directly. Others (such as support vector machine classifier or linear classifiers) are strictly binary classifiers. However, there are various strategies that you can use to perform multiclass classification using multiple binary classifiers.

For example, one way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector and so on). Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score. This is called the *one-vs-all* (OvA) strategy (also called *one-vs-the-rest*).

Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is called the *one-vs-one* (OvO) strategy. If there are N classes, you need to train $N \times (N-1)/2$ classifiers. For the MNIST problem, this means training 45 binary classifiers! When you want to classify an image you have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.

Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set, so for these algorithms OvO is preferred since it is faster to train many classifiers on small training sets than training few classifiers on large training sets. For most binary classification algorithms, however, OvA is preferred.

Scikit-Learn detects when you try to use a binary classification algorithm for a multi-class classification task, and it automatically runs OvA (except for SVM classifiers for which it uses OvO). Let's try this with the `SGDClassifier`:

```
>>> sgd_clf.fit(X_train, y_train) # y_train, not y_train_5  
>>> sgd_clf.predict([some_digit])  
array([ 5.])
```

That was easy! This code trains the `SGDClassifier` on the training set using the original targets classes from 0 to 9 (`y_train`), instead of the 5-vs-all target classes (`y_train_5`). Then it makes a prediction (a correct one in this case). Under the hood, Scikit-Learn actually trained 10 binary classifiers, then it got their decision scores for the image and it selected the class with the highest score.

To see that this is indeed the case, you can call the `decision_function()` method: instead of returning just one score per instance, it now returns 10 scores, one per class:

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])  
>>> some_digit_scores  
array([[-311402.62954431, -363517.28355739, -446449.5306454 ,  
       -183226.61023518, -414337.15339485,  161855.74572176,  
       -452576.39616343, -471957.14962573, -518542.33997148,  
       -536774.63961222]])
```

The highest score is indeed the one corresponding to class 5:

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> sgd_clf.classes[5]
5.0
```



When a classifier is trained, it stores the list of target classes in its `classes_` attribute, ordered by value. In this case, the index of each class in the `classes_` array conveniently matches the class itself (eg. the class at index 5 happens to be class 5), but in general you won't be so lucky.

If you want to force ScikitLearn to use one-vs-one or one-vs-all, you can use the `OneVsOneClassifier` or `OneVsRestClassifier` classes. Simply create an instance and pass a binary classifier to its constructor. For example this code creates a multiclass classifier using the OvO strategy, based on a `SGDClassifier`:

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

Training a `RandomForestClassifier` is just as easy:

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

This time Scikit-Learn did not have to run OvA or OvO because Random Forest classifiers can directly classify instances into multiple classes. You can call `predict_proba()` to get the list of probabilities that the classifier assigned to each instance for each class:

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. ,  0.1,  0. ,  0.8,  0. ,  0. ,  0. ]])
```

You can see that the classifier is fairly confident about its prediction: the 0.8 at the 5th index in the array means that the model estimates an 80% probability that the image represents a 5. It also thinks that the image could instead be a 0 or a 3 (10% chance each).

Now of course you want to evaluate these classifiers. As usual, you want to use cross-validation. Let's evaluate the `SGDClassifier`'s` accuracy using the ``cross_val_score()`` function:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([ 0.84063187,  0.84899245,  0.86652998])
```

It gets over 84% on all test folds. If you used a random classifier, you would get 10% accuracy, so this is not such a bad score, but you can still do much better. For example, simply scaling the inputs (as discussed in [Chapter 2](#)) increases accuracy above 90%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([ 0.91011798,  0.90874544,  0.906636 ])
```

Error analysis

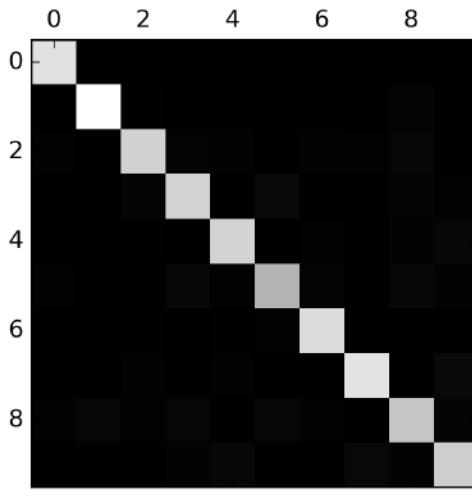
Of course if this were a real project you would follow the steps in your Machine Learning project checklist (see [Appendix B](#)), exploring data preparation options, trying out multiple models, short-listing the best ones and fine-tuning their hyperparameters using `GridSearchCV`, automating as much as possible as you did in the previous chapter. Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

First, you can look at the confusion matrix. You need to make predictions using the `cross_val_predict()` function, then call the `confusion_matrix()` function, just like you did earlier:

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5725,     3,   24,     9,   10,   49,   50,   10,   39,     4],
       [  2, 6493,   43,   25,    7,   40,    5,   10,  109,     8],
       [ 51,   41, 5321,  104,   89,   26,   87,   60,  166,   13],
       [ 47,   46,  141, 5342,    1,  231,   40,   50,  141,   92],
       [ 19,   29,   41,   10, 5366,    9,   56,   37,   86,  189],
       [ 73,   45,   36,   193,   64, 4582,  111,   30,  193,   94],
       [ 29,   34,   44,     2,   42,   85, 5627,   10,   45,     0],
       [ 25,   24,   74,   32,   54,   12,    6, 5787,   15,  236],
       [ 52,  161,   73,  156,   10, 163,   61,   25, 5027,  123],
       [ 43,   35,   26,   92, 178,   28,    2,  223,   82, 5240]])
```

That's a lot of numbers. It's often more convenient to look at an image representation of the confusion matrix, using Matplotlib's `matshow()` function:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



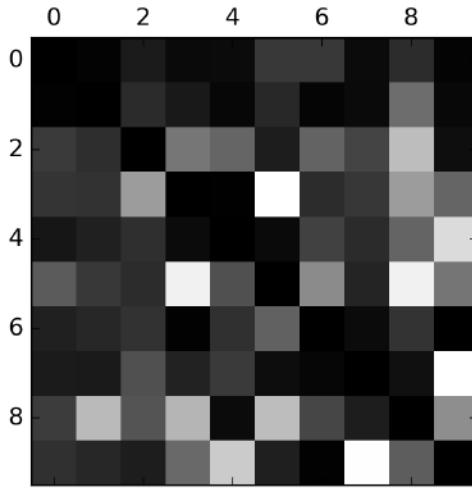
This confusion matrix looks fairly good, since most images are on the main diagonal, which means that they were classified correctly. The 5s look slightly darker than the other digits, which could mean that there are less images of 5s in the dataset or that the classifier does not perform as well on 5s as on other digits. In fact you can verify that both are the case.

Let's focus the plot on the errors. First you need to divide each value in the confusion matrix by the number of images in the corresponding class, so you can compare error rates instead of absolute number of errors (which would make abundant classes look unfairly bad):

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Now let's fill the diagonal with zeros to keep only the errors, and let's plot the result:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



Now you can clearly see the kinds of errors the classifier makes. Remember that rows represent actual classes while columns represent predicted classes. The columns for classes 8 and 9 are quite bright, which tells you that many images get misclassified as 8s or 9s. Similarly, the rows for classes 8 and 9 are also quite bright, telling you that 8s and 9s are often confused with other digits. Conversely, some rows are pretty dark, such as row 1: this means that most 1s are classified correctly (a few are confused with 8s, but that's about it). Notice that the errors are not perfectly symmetrical: for example, there are more 5s misclassified as 8s than the reverse.

Analyzing the confusion matrix can often give you insights on ways to improve your classifier. Looking at this plot, it seems that your efforts should be spent on improving classification of 8s and 9s, as well as fixing the specific 3/5 confusion. For example, you could try to gather more training data for these digits. Or you could engineer new features that would help the classifier, for example writing an algorithm to count the number of closed loops (eg. 8 has two, 6 has one, 5 has none). Or you could preprocess the images (eg. using Scikit-Image, Pillow or OpenCV) to make some patterns stand out more, such as closed loops.

Analyzing individual errors can also be a good way to gain insights on what your classifier is doing and why it is failing, but it is more difficult and time consuming. For example, let's plot examples of 3s and 5s:

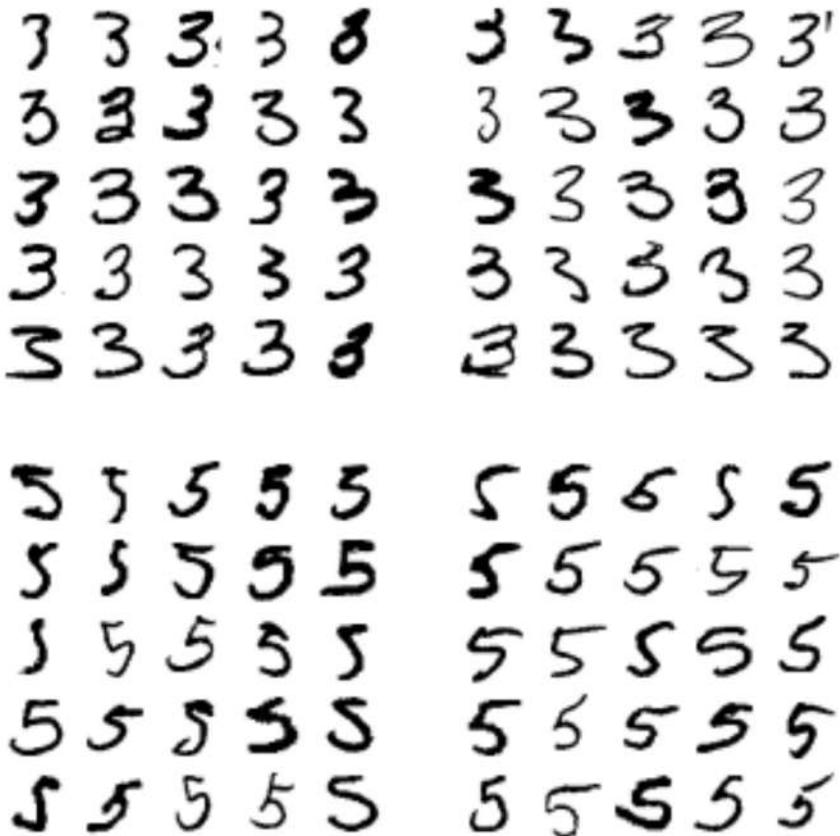
```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
```

```

X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()

```



The two 5×5 blocks on the left show digits classified as 3s, and the two 5×5 blocks on the right show images classified as 5s. Some of the digits that the classifier gets wrong (ie. in the bottom left and top right blocks) are so badly written that even a human would have trouble classifying them (eg. the 5 on the 8th row and 1st column truly looks like a 3). However, most misclassified images seem like obvious errors to us,

and it's hard to understand why the classifier made the mistakes it did³. The reason is that we used a simple `SGDClassifier`, which is a linear model: all it does is assign a weight per class to each pixel, and when it sees a new image it just sums up the weighted pixel intensities to get a score for each class. So since 3s and 5s differ only by a few pixels, this model will easily confuse them.

The main difference between 3s and 5s is the position of the small line that joins the top line to the bottom arc. If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and *vice versa*. In other words, this classifier is quite sensitive to image shifting and rotation. So one way to reduce the 3/5 confusion would be to preprocess the images to ensure that they are well centered and not too rotated. This will probably help reduce other errors as well.

Multilabel classification

Until now each instance has always been assigned to just one class. In some cases you may want your classifier to output multiple classes for each instance. For example, consider a face-recognition classifier: what should it do if it recognizes several people on the same picture? Of course it should attach one label per person it recognizes. Say the classifier has been trained to recognize 3 faces, Alice, Bob and Charlie, then when it is shown a picture of Alice and Charlie it should output [1, 0, 1] (meaning "Alice yes, Bob no, Charlie yes"). Such a classification system that outputs multiple binary labels is called a *multilabel classification system*.

We won't go into face recognition just yet, but let's look at a simpler example, just for illustration purposes:

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image: the first indicates whether or not the digit is large (7, 8 or 9) and the second indicates whether or not it is odd. The next lines create a `KNeighborsClassifier` instance (which supports multilabel classification, not all classifiers do) and we train

³ But remember that our brain is a fantastic pattern recognition system, and our visual system does a lot of complex preprocessing before any information reaches our consciousness, so the fact that it feels simple does not mean that it is.

it using the multiple targets array. Now you can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]], dtype=bool)
```

And it gets it right! The digit 5 is indeed not large (`False`) and odd (`True`).

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project. For example, one approach is to measure the F_1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score. This code computes the average F_1 score across all labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_train, cv=3)
>>> f1_score(y_train, y_train_knn_pred, average="macro")
0.96845540180280221
```

This assumes that all labels are equally important, which may not be the case. In particular, if you have much more pictures of Alice than pictures of Bob or Charlie, you may want to give more weight to the classifier's score on pictures of Alice. One simple option is to give each label a weight equal to its `support` (ie. the number of instances with that target label). To do this, simply set `average="weighted"` in the code above⁴.

Multioutput classification

The last type of classification task we are going to discuss here is called *multioutput-multiclass classification* (or simply *multioutput classification*). It is simply a generalization of multilabel classification where each label can be multiclass (ie. it can have more than two possible values).

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). It is thus an example of a multioutput classification system.

⁴ Scikit-Learn offers a few other averaging options and multilabel classifier metrics, see the documentation for more details.

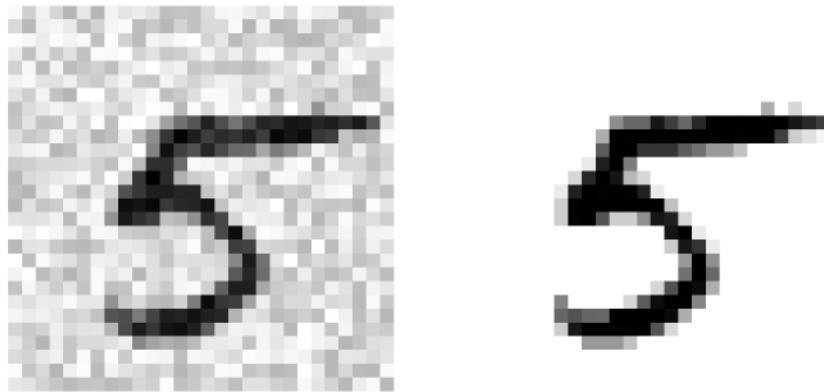


The line between classification and regression is sometimes blurry, such as in this example: arguably, predicting pixel intensity is more akin to regression than to classification. Moreover, multioutput systems are not limited to classification tasks: you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities using NumPy's `randint` function. The target images will be the original images:

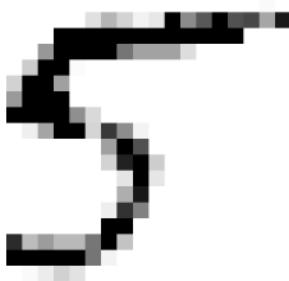
```
noise = rnd.randint(0, 100, (len(X_train), 784))
noise = rnd.randint(0, 100, (len(X_test), 784))
X_train_mod = X_train + noise
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Let's take a peek at an image from the test set (yes, we're snooping the test data, you should be frowning right now):



On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean this image:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



Looks close enough to the target! This concludes our tour of classification: hopefully you should now know how to select good metrics for classification tasks, pick the appropriate precision/recall tradeoff, compare classifiers and more generally build good classification systems for a variety of tasks.

Exercises

1. Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set. Hint: the `KNeighborsClassifier` works quite well for this task, you just need to find good hyperparameter values (try a grid search on the `weights` and `n_neighbors` hyperparameters).
2. Write a function that can shift an MNIST image in any direction (left, right, up or down) by one pixel⁵. Then for each image in the training set, create 4 shifted copies (one per direction) and add them to the training set. Finally, train your best model on this expanded training set and measure its accuracy on the test set. You should observe that your model performs even better now! This technique of artificially growing the training set is called *data augmentation* or *training set expansion*.

⁵ You can use the `shift()` function from the `scipy.ndimage.interpolation` module: for example, `shift(image, [2, 1], cval=0)` shifts the image 2 pixels down and 1 pixel to the right)

3. Tackle the Titanic dataset. A great place to start is on Kaggle.com at <https://www.kaggle.com/c/titanic>.
4. Build a spam classifier (a more challenging exercise):
 - Download examples of spam and ham from Apache SpamAssassin's public datasets at <https://spamassassin.apache.org/publiccorpus/>
 - Unzip the datasets and familiarize yourself with the data format.
 - Split the datasets into a training set and a test set.
 - Write a data preparation pipeline to convert each email into a feature vector: your preparation pipeline should transform an email into a (sparse) vector indicating the presence or absence of each possible word. For example if all emails only ever contain 4 words "Hello", "how", "are", "you", then the email "Hello you Hello Hello you" would be converted into a vector [1, 0, 0, 1] (meaning ["Hello" is present, "how" is absent, "are" is absent, "you" is present]), or [3, 0, 0, 2] if you prefer to count the number of occurrences of each word.
 - You may want to add hyperparameters to your preparation pipeline to control whether or not strip off email headers, convert each email to lowercase, remove punctuation, replace all URLs with "URL", replace all numbers by "NUMBER", or even perform *stemming* (ie. trim off word endings, there are python libraries available to do this).
 - Then try out several classifiers and see if you can build a great spam classifier, with both high recall and high precision.

Solutions to these exercises are available in the online Jupyter notebooks at <https://github.com/ageron/handson-ml>.

CHAPTER 4

Training linear models

So far we have treated Machine Learning algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what's under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch, all this without knowing how the algorithms actually work. Indeed, in many situations you don't really need to know the implementation details.

However, having a good understanding of how the algorithms work can help you quickly home in on the appropriate model and hyperparameters for your task: in particular, you will want to know what data regimes an algorithm is good at so you can avoid wasting your time evaluating a model that is unlikely to perform well or that will take forever to train. Understanding the learning algorithms will also help you debug issues and perform error analysis more efficiently. Finally, it will be hard to understand neural networks if you don't understand how their components work (many of which are simple linear models).

So in this chapter we will look closely at the linear models we used in the previous chapters. In the process, we will discuss some techniques such as learning curves, early stopping or regularization, which are useful far beyond linear models.



There will be quite a few math equations in this chapter, using basic notions of linear algebra and calculus. To understand these equations, you will need to know what vectors and matrices are, how to transpose them, what is the dot product, what matrix inverse is and what partial derivatives are. If you are unfamiliar with these notions, please go through the linear algebra and calculus introductory tutorials available as Jupyter notebooks in the online supplemental material. For those who are truly allergic to mathematics, you should still go through this chapter and simply skip the equations, hopefully the text should be sufficient to understand most of the concepts.

Linear Regression

In [Chapter 1](#), we looked at a simple regression model of life satisfaction: $life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$.

This model is just a linear function of the input feature `GDP_per_capita`. θ_0 and θ_1 are the model's parameters.

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*):

Equation 4-1. Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- \hat{y} is the predicted value,
- n is the number of features,
- x_i is the i^{th} feature value,
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights).

This can be written much more concisely using a vectorized form:

Equation 4-2. Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta}^T \cdot \mathbf{x}$$

- θ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n ,

- θ^T is the transpose of θ (a row vector instead of a column vector),
- \mathbf{x} is the instance's *feature vector*, containing x_0 to x_n , with x_0 always equal to 1,
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and \mathbf{x} ,
- h_θ is the hypothesis function, using the model parameters θ .

In [Chapter 2](#) we saw that the most common performance measure of a regression model is the Root Mean Square Error (RMSE) ([Equation 2-1](#)). To train a Linear Regression model, you need to find the value of θ that minimizes the RMSE. In practice, it is simpler to minimize the Mean Square Error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root)¹.

The MSE of a Linear Regression hypothesis h_θ on a training set \mathbf{X} is calculated using the following formula:

Equation 4-3. MSE cost function

$$\text{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

Most of these notations were presented in [Chapter 2](#) (see “[Notations](#)” on page 60), the only difference is that we write h_θ instead of just h in order to make it clear that the model is parametrized by the vector θ . To simplify notations, we will just write $\text{MSE}(\theta)$ instead of $\text{MSE}(\mathbf{X}, h_\theta)$.

The Normal Equation

To find the value of θ that minimizes the MSE, it turns out that there is a *closed-form solution*, in other words a mathematical equation that gives the result directly. This equation is called the *Normal Equation*.²

Equation 4-4. Normal Equation

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

¹ It is often the case that a learning algorithm will try to optimize a different function than the performance measure, but in general it will lead to the same optimal parameter values (or close enough).

² The demonstration that this equation returns the value of θ that minimizes the MSE is outside the scope of this book

- $\hat{\theta}$ is the value of θ that minimizes the MSE cost function.
- y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Let's generate some linear-looking data to test this equation on:

```
import numpy as np
import numpy.random as rnd

X = 2 * rnd.rand(100, 1)
y = 4 + 3 * X + rnd.randn(100, 1)
```

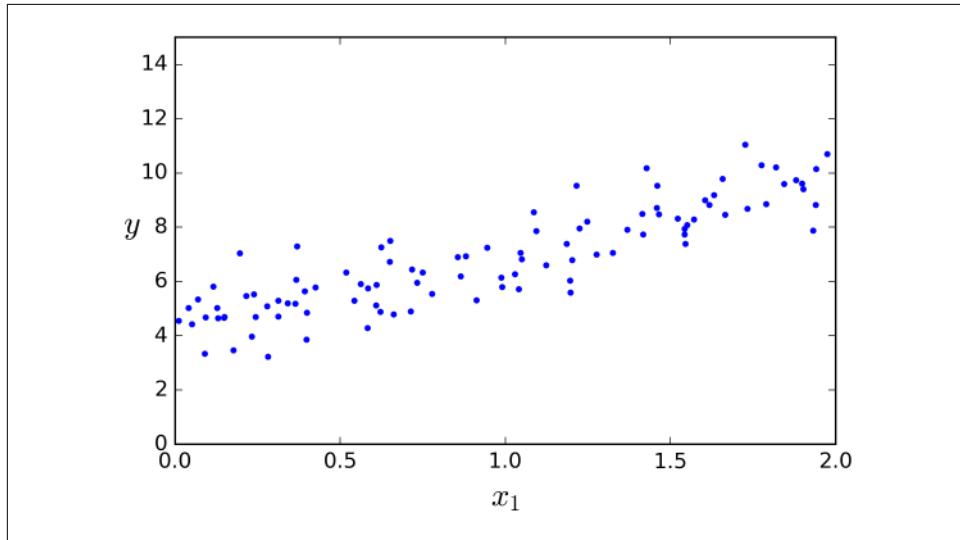


Figure 4-1. Randomly generated linear dataset

Now let's compute $\hat{\theta}$ using the Normal Equation:

```
import numpy.linalg as LA # NumPy's linear algebra module

X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = LA.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

The actual function that we used to generate the data is $y = 4 + 3x_0 + \text{Gaussian noise}$. Let's see what the equation found:

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

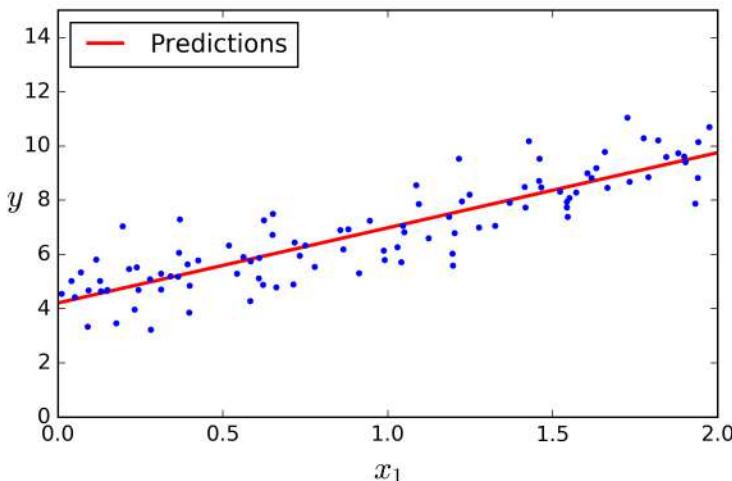
We would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$ instead of $\theta_0 = 3.865$ and $\theta_1 = 3.139$. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

Now you can make predictions using $\hat{\theta}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Let's plot this model's predictions:

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



The equivalent code using Scikit-Learn looks like this³:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

³ Note that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`).

Computational complexity

The Normal Equation computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$, which is an $n \times n$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation). In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.



The Normal Equation gets very slow when the number of features grows large (eg. 100,000).

On the positive side, this equation is linear with regards to the number of instances in the training set (it is $O(m)$) so it handles large training sets efficiently, provided they can fit in memory.

Also, once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast: the computational complexity is linear with regards to both the number of instances you want to make predictions on and the number of features. In other words, making predictions on twice as many instances (or twice as many features) will just take roughly twice as much time.

Now we will look at very different ways to train a Linear Regression model, better suited for cases where there are a large number of features, or too many training instances to fit in memory.

Gradient Descent

Gradient descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

Suppose you are lost in the mountains in a dense fog, you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with regards to the parameter vector θ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

Concretely, you start by filling θ with random values (this is called *random initialization*), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (eg. the MSE), until the algorithm *converges* to a minimum (see [Figure 4-2](#)).

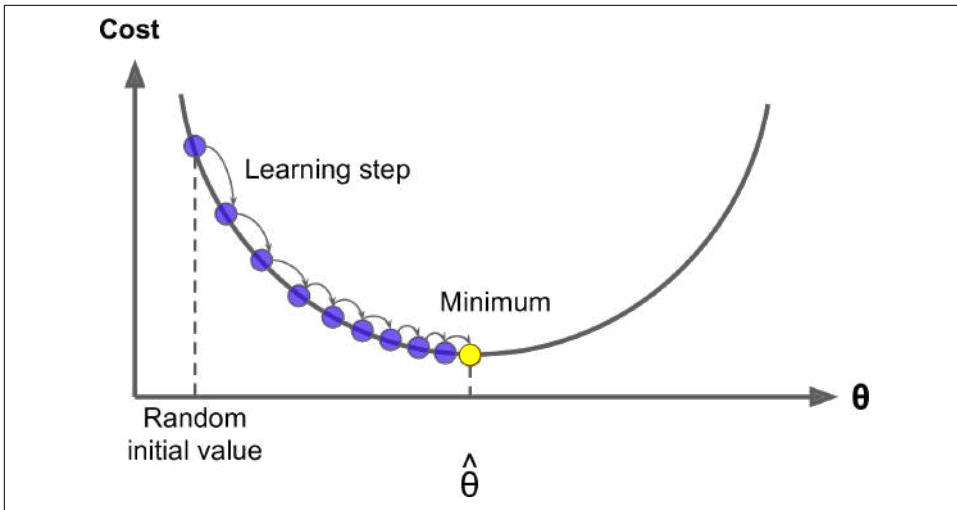


Figure 4-2. Gradient descent

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 4-3).

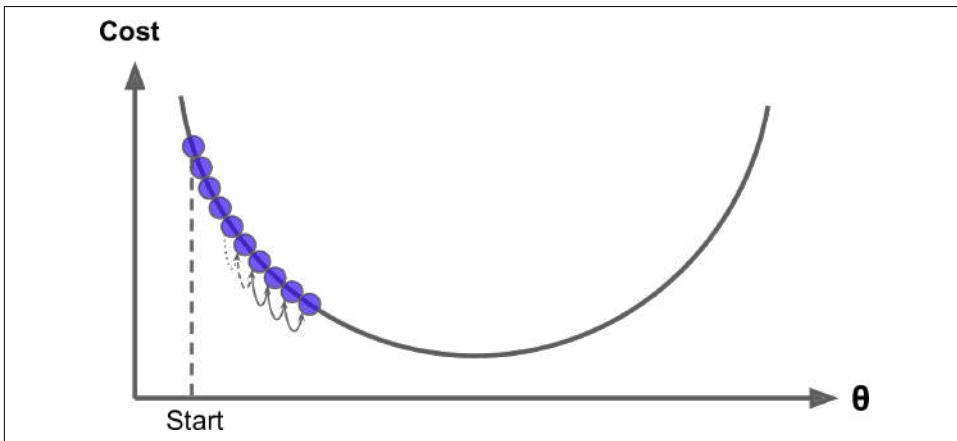


Figure 4-3. Learning rate too small

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 4-4).

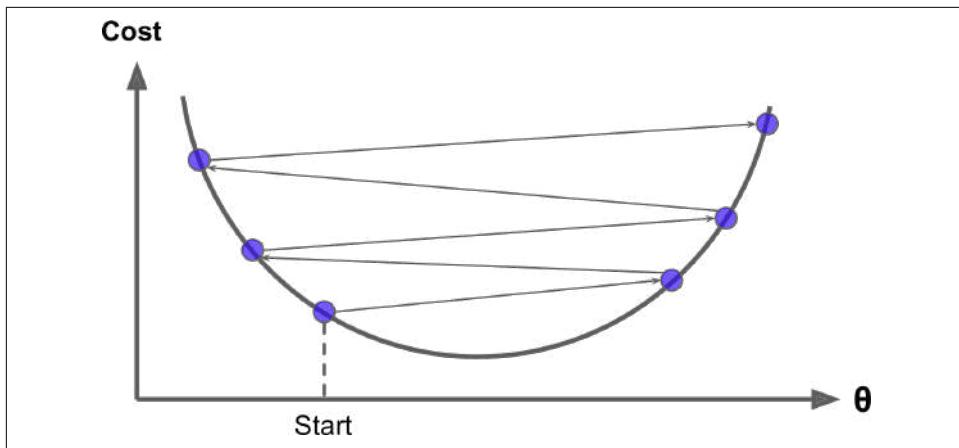


Figure 4-4. Learning rate too large

Finally, not all cost functions look like nice regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult. Figure 4-5 shows the two main challenges with Gradient Descent: if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*. If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum.

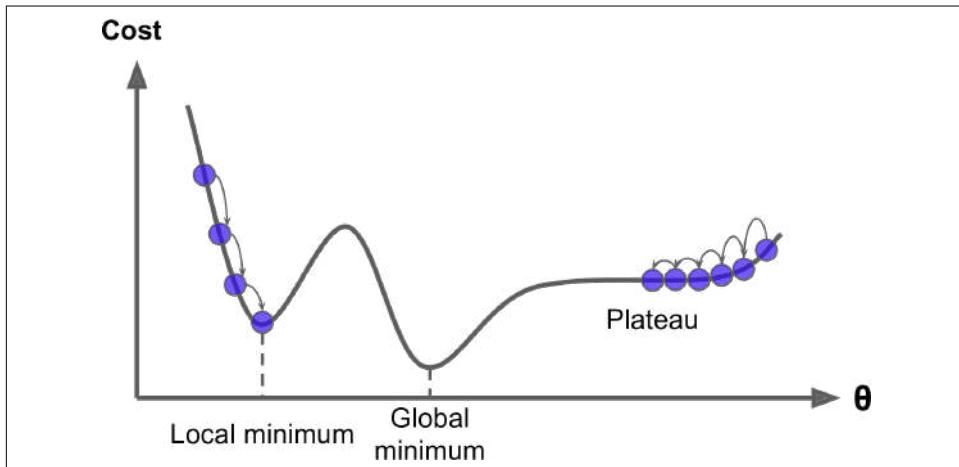


Figure 4-5. Gradient Descent pitfalls

Fortunately, the MSE cost function happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve. This implies that there are no local minima, just one global minimum. It is

also a continuous function with a slope that never changes abruptly⁴. These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

In fact, the MSE cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. [Figure 4-6](#) shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right)⁵.

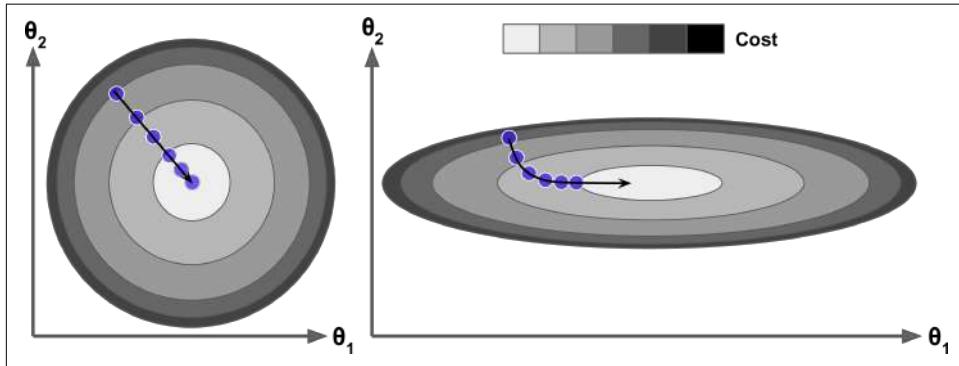


Figure 4-6. Gradient descent with and without feature scaling

As you can see, on the left the Gradient Descent algorithm goes straight towards the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.



When using Gradient Descent, you should ensure that all features have a similar scale (eg. using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a search in the model's *parameter space*: the more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a nee-

⁴ Technically speaking, its derivative is *Lipschitz continuous*.

⁵ Since feature 1 is smaller, it takes a larger change in θ_1 to affect the cost function, which is why the bowl is elongated along the θ_1 axis.

dle in a 300-dimensional haystack is much trickier than in 3 dimensions. Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl.

Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θ_j ; in other words, you need to calculate how much the cost function will change if you change θ_j just a little bit. This is called a *partial derivative*. It is like asking “what is the slope of the mountain under my feet if I face East?”, then asking the same question facing North (and so on for all other dimensions, if you can imagine a universe with more than 3 dimensions). The following equation computes the partial derivative of the MSE with regards to parameter θ_j , noted $\frac{\partial}{\partial \theta_j} \text{MSE}(\theta)$:

Equation 4-5. MSE partial derivatives

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Instead of computing these gradients individually, you can use the following vectorized equation to compute them all in one go. The gradient vector, noted $\nabla_{\theta} \text{MSE}(\theta)$, contains all the partial derivatives of the MSE (one for each model parameter):

Equation 4-6. MSE gradient vector

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$



Notice that this formula involves calculations over the full training set \mathbf{X} , at each Gradient Descent step! This is why the algorithm is called *Batch Gradient Descent*: it uses the whole batch of training data at every step. As a result it is terribly slow on very large training sets (but we will see much faster Gradient Descent algorithms below). However, Gradient Descent scales well with the number of features: training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\theta} \text{MSE}(\theta)$ from θ . This is where the learning rate η comes into play⁶: multiply the gradient vector by η to determine the size of the downhill step:

Equation 4-7. Gradient descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Let's look at a quick implementation of this algorithm:

```
eta = 0.1 # learning rate
n_iterations = 1000

theta = rnd.randn(2,1) # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta)) - y
    theta = theta - eta * gradients
```

That wasn't too hard! Let's look at the resulting `theta`:

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```

Hey, that's exactly what the Normal Equation found! Gradient descent worked perfectly. But what if you had used a different learning rate `eta`? [Figure 4-7](#) shows the first 10 steps of Gradient Descent using 3 different learning rates (the dashed line represents the starting point).

⁶ Eta (η) is the 7th letter of the Greek alphabet.

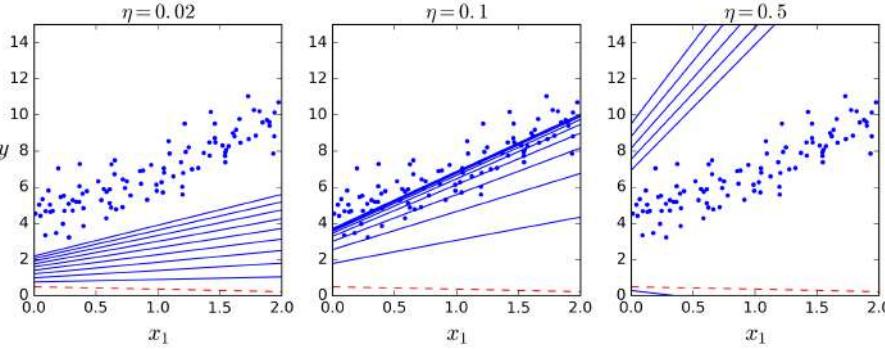


Figure 4-7. Gradient Descent with various learning rates

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution. On the right, the learning rate is too high, the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

To find a good learning rate, you can use grid search (see [Chapter 2](#)). However, you may want to limit the number of iterations so that grid search can eliminate models that take too long to converge.

You may wonder how to set the number of iterations. If it is too low, you will still be far away from the optimal solution when the algorithm stops, but if it is too high, you will waste time while the model parameters do not change anymore. A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny, that is when its norm becomes smaller than a tiny number ϵ (called the *tolerance*), because this happens when Gradient Descent has (almost) reached the minimum.

Convergence rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), it can be shown that Batch Gradient Descent with a fixed learning rate has a *convergence rate* of $O\left(\frac{1}{\text{iterations}}\right)$: in other words, if you divide the tolerance ϵ by 10 (to have a more precise solution), then the algorithm will have to run about 10 times more iterations.

Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, *Stochastic Gradient Descent* just picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration (SGD can be implemented as an out-of-core algorithm⁷).

On the other hand, due to its stochastic (ie. random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see [Figure 4-8](#)). So once the algorithm stops, the final parameter values are good, but not optimal.

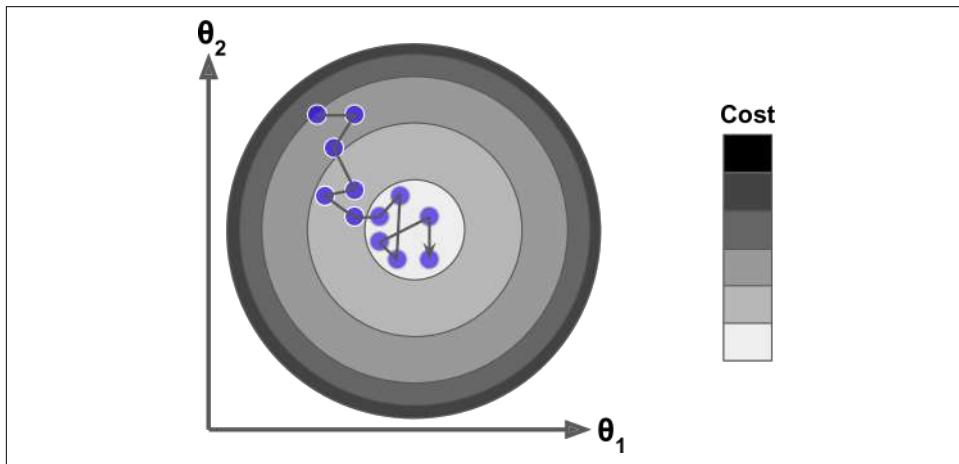


Figure 4-8. Stochastic Gradient Descent

When the cost function is very irregular (as in [Figure 4-5](#)), this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate: the steps start out large (which helps make

⁷ Out-of-core algorithms are discussed in [Chapter 1](#)

quick progress and escape local minima) then they get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is called *simulated annealing*, because it resembles the process of annealing in metallurgy where molten metal is slowly cooled down. The function that determines the learning rate at each iteration is called the *learning schedule*. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen half way to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

This code implements Stochastic Gradient Descent using a simple learning schedule:

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = rnd.randn(2,1) # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = rnd.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

By convention we iterate by rounds of m iterations: each round is called an *epoch*. While the Batch Gradient Descent code iterated 1,000 times through the whole training set, this code goes through the training set only 50 times and reaches a fairly good solution:

```
>>> theta
array([[ 4.21076011],
       [ 2.74856079]])
```

Figure 4-9 shows the first 10 steps of training (notice how irregular the steps are).

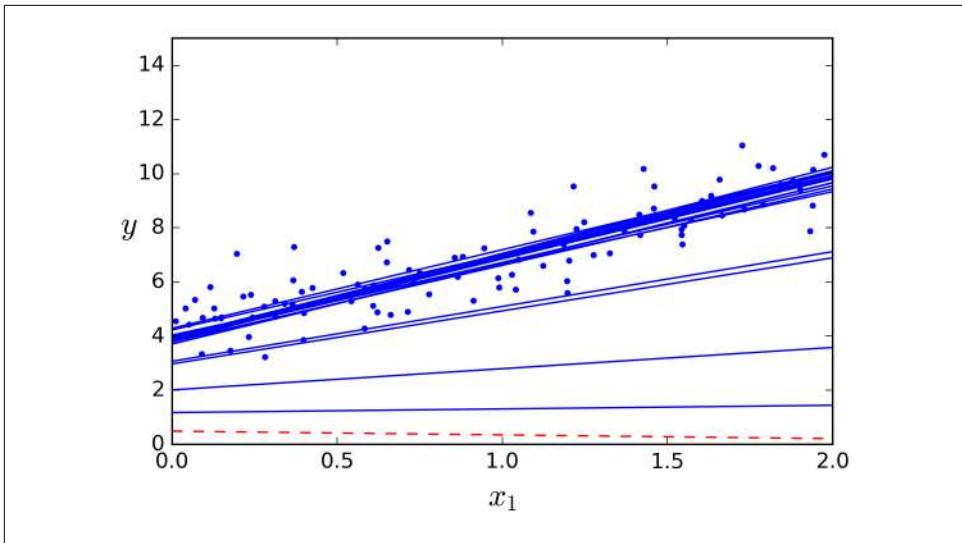


Figure 4-9. Stochastic Gradient Descent first 10 steps

Note that since instances are picked randomly, some instances may be picked several times per epoch while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set, then go through it instance by instance, then shuffle it again, and so on. However this generally converges more slowly.

To perform Linear Regression using SGD with Scikit-Learn, you can use the `SGDRegressor` class which defaults to optimizing the squared error cost function. The following code runs 50 epochs, starting with a learning rate of 0.1 (`eta0=0.1`), using the default learning schedule (different from the one above), and it does not use any regularization (`penalty=None`, more details on this below).

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Once again, you find a solution very close to the one returned by the Normal Equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.18380366]), array([ 2.74205299]))
```

Mini-batch Gradient Descent

The last Gradient Descent algorithm we will look at is called *Mini-batch Gradient Descent*. It is quite simple to understand once you know batch and Stochastic Gradient Descent: at each step, instead of computing the gradients based on the full train-

ing set (as in Batch GD) or based on just one instance (as in stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

The algorithm's progress in parameter space is less erratic than with SGD, especially with fairly large mini-batches: as a result Mini-batch GD will end up walking around a bit closer to the minimum than SGD. But on the other hand it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression as we saw earlier). [Figure 4-10](#) shows the paths taken by the 3 Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum while both Stochastic GD and Mini-Batch GD continue to walk around. However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

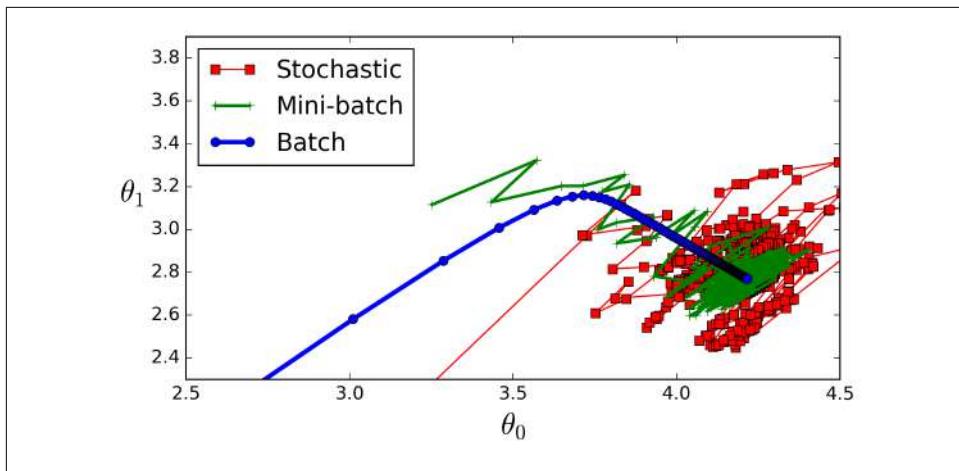


Figure 4-10. Gradient descent paths in parameter space

Let's compare the algorithms we discussed so far for Linear Regression⁸ (recall that m is the number of training instances and n is the number of features).

⁸ While the Normal Equation can only perform Linear Regression, the Gradient Descent algorithms can be used to train many other models, as we will see.

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large m	Out-of-core support	Large n	Hyper params	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor
Mini-Batch GD	Fast	Yes	Fast	≥ 2	Yes	n/a



There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

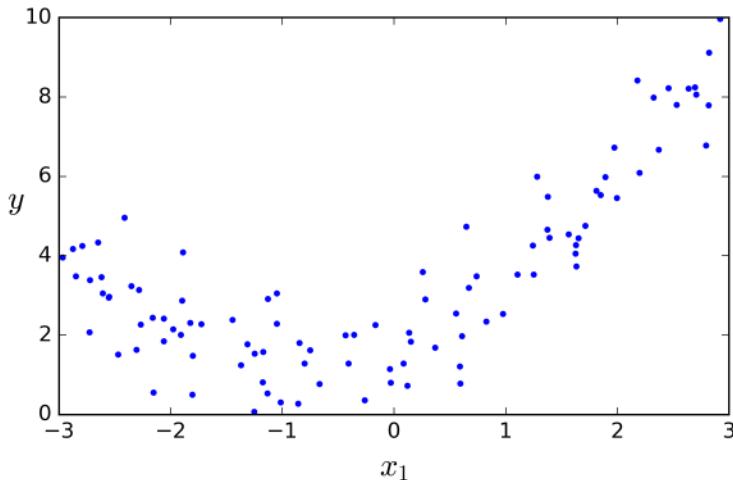
Polynomial regression

What if your data is actually more complex than a simple straight line? Surprisingly, you can actually use a linear model to fit non-linear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features: this technique is called *polynomial regression*.

Let's look at an example: first, let's generate some non-linear data, based on a simple *quadratic equation*⁹ (plus some noise):

```
m = 100
X = 6 * rnd.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + rnd.randn(m, 1)
```

⁹ A quadratic equation is of the form $y = ax^2 + bx + c$.

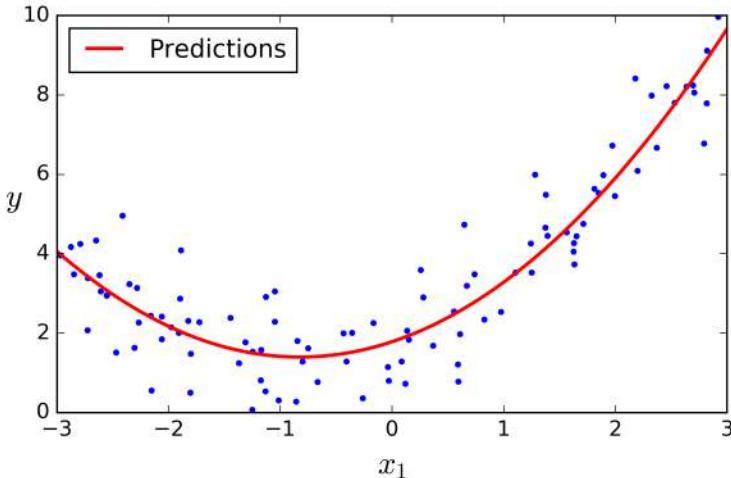


Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `PolyynomialFeatures` class to transform our training data, adding the square (2nd degree polynomial) of each feature in the training set as new features (in this case there is just one feature):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

`X_poly` now contains the original feature of `X` plus the square of this feature. Now you can fit a `LinearRegression` model to this extended training data:

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```



Not bad, the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

Note that when there are multiple features, polynomial regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do). This is made possible by the fact that `PolynomialFeatures` also adds all combinations of features up to the given degree. For example, if there were two features a and b , `PolynomialFeatures` with `degree=3` would not only add the features a^2 , a^3 , b^2 and b^3 , but also the combinations ab , a^2b and ab^2 .



`PolynomialFeatures(degree=d)` transforms an array containing n features into an array containing $\frac{(n+d)!}{d! n!}$ features, where $n!$ is the factorial of n , equal to $1 \times 2 \times 3 \times \dots \times n$. Beware of the combinatorial explosion of the number of features!

Learning curves

If you perform high-degree polynomial regression, you will likely fit the training data much better than with plain Linear Regression. For example, Figure 4-11 applies a 300 degree polynomial model to the training data above, and compares the result with a pure linear model and a quadratic model (2nd degree polynomial). Notice how the 300 degree polynomial model wiggles around to get as close as possible to the training instances.

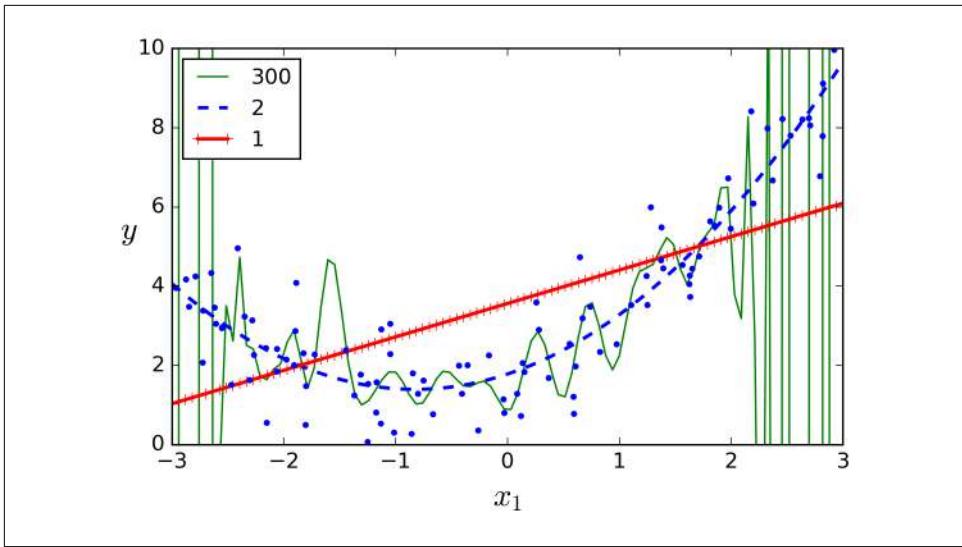


Figure 4-11. High degree polynomial regression

Of course, this high degree polynomial model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model (it makes sense since the data was generated using a quadratic model. But in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

In Chapter 2 you used cross validation to get an estimate of a model's generalization performance: if a model performs well on the training data but generalizes poorly according to the cross validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

Another way is to look at the *learning curves*: these are plots of the model's performance on the training set and the validation set as a function of the training set size. To generate the plots, simply train the model several times on different sized subsets of the training set. The following code defines a function that plots the learning curves of a model given some training data:

```
from sklearn.metrics import mean_squared_error
from sklearn.cross_validation import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train))
        val_errors.append(mean_squared_error(y_val_predict, y_val))

    plt.plot(np.sqrt(train_errors), 'r')
    plt.plot(np.sqrt(val_errors), 'b')
    plt.xlabel('Training set size')
    plt.ylabel('RMSE')
```

```

y_train_predict = model.predict(X_train[:m])
y_val_predict = model.predict(X_val)
train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
val_errors.append(mean_squared_error(y_val_predict, y_val))
plt.plot(np.sqrt(train_errors), "r+-", linewidth=2, label="train")
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")

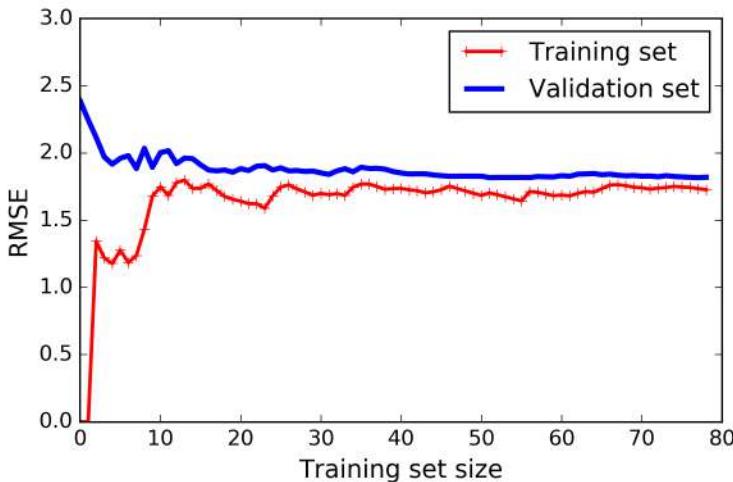
```

Let's look at the learning curves of the plain Linear Regression model (a straight line):

```

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)

```



This deserves a bit of explanation. First, let's look at the performance on the training data: when there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau: at that point, adding new instances to the training set doesn't make the average error much better or worse. Now let's look at the performance of the model on the validation data: when the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big. Then as the model is shown more training examples, it learns and thus the validation error slowly goes down. However, once again a straight line cannot do a good job modeling the data, so the error ends up at a plateau, very close to the other curve.

These learning curves are typical of an underfitting model: both curves have reached a plateau, they are close and fairly high.

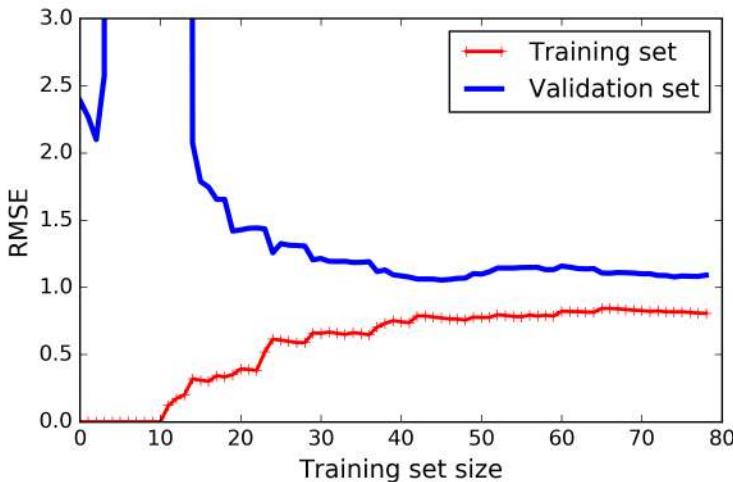


If your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.

Now let's look at the learning curves of a 10th degree polynomial model on the same data:

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("sgd_reg", LinearRegression()),
])
plot_learning_curves(polynomial_regression, X, y)
```



These learning curves look a bit like the previous ones, but there are two very important differences:

1. The error on the training data is much lower than with the Linear Regression model.
2. There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. However, if you used a much larger training set, the two curves would continue to get closer.



One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

The Bias/Variance tradeoff

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

- *Bias*: this part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.¹⁰
- *Variance*: this part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance, and thus to overfit the training data.
- *Irreducible error*: this part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (eg. fix the data sources, such as broken sensors, or detect and remove outliers, etc.).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a tradeoff.

Regularized linear models

As we saw in [Chapter 1](#) and [Chapter 2](#), a good way to reduce overfitting is to regularize the model (ie. to constrain it): the less degrees of freedom it has, the harder it will be for it to overfit the data. For example, a simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge regression, Lasso regression and Elastic Net which implement three different ways to constrain the weights.

¹⁰ This notion of bias is not to be confused with the bias term of linear models.

Ridge regression

Ridge regression (also called *Tikhonov regularization*) is a regularized version of Linear Regression: a *regularization term* equal to $\alpha \sum_{i=1}^n \theta_i^2$ is added to the cost function. This forces the learning algorithm to not only fit the data but also to keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to evaluate the model's performance using the unregularized performance measure.



It is quite common for the cost function used during training to be different from the performance measure used for testing. Apart from regularization, another reason why they might be different is that a good training cost function should have optimization-friendly derivatives, while the performance measure used for testing should be as close as possible to the final objective. A good example of this is a classifier trained using a cost function such as the log loss (discussed below) but evaluated using precision/recall.

The hyperparameter α controls how much you want to regularize the model. If $\alpha = 0$ then Ridge regression is just Linear Regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. The following equation presents the Ridge regression cost function¹¹.

Equation 4-8. Ridge regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define \mathbf{w} as the vector of feature weights (θ_1 to θ_n), then the regularization term is simply equal to $\frac{1}{2} (\|\mathbf{w}\|_2)^2$, where $\|\cdot\|_2$ represents the ℓ_2 norm of the weight vector¹². For Gradient Descent, just add $\alpha \mathbf{w}$ to the MSE gradient vector (Equation 4-6).

¹¹ It is common to use the notation $J(\theta)$ for cost functions that don't have a short name: we will often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

¹² Norms are discussed in Chapter 2.



It is important to scale the data (eg. using a `StandardScaler`) before performing Ridge regression, as it is sensitive to the scale of the input features. This is true of most regularized models.

Figure 4-12 shows several Ridge models trained on some linear data using different α value. On the left plain Ridge models are used, leading to linear predictions. On the right, the data is first expanded using `PolynomialFeatures(degree=10)`, then scaled using a `StandardScaler`, and finally the Ridge models are applied to the resulting features: this is polynomial regression with Ridge regularization. Note how increasing α leads to flatter (ie. less extreme, more reasonable) predictions: this reduces the model's variance but increases its bias.

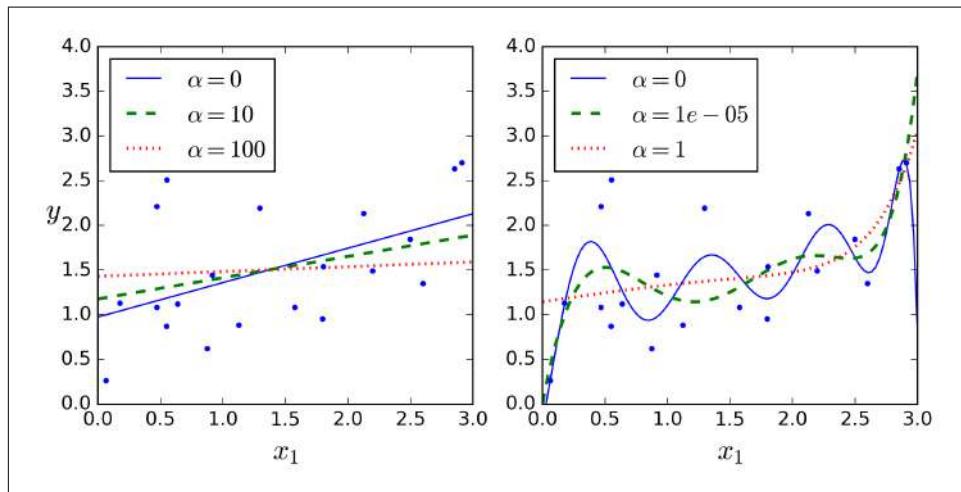


Figure 4-12. Ridge regression

Just like Linear Regression, Ridge regression can be performed either by computing a closed-form equation or by performing Gradient Descent. The pros and cons are the same. Equation 4-9 shows the closed-form solution (where \mathbf{A} is the $n \times n$ identity matrix¹³ except with a 0 in the top left cell, corresponding to the bias term).

Equation 4-9. Ridge regression closed-form solution

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

¹³ A square matrix full of 0s except for 1s on the main diagonal (top left to bottom right).

Here is how to perform Ridge regression with Scikit-Learn using a closed-form solution (a variant of the above using a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[ 1.55071465]])
```

And using Stochastic Gradient Descent¹⁴:

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> ridge_reg.predict([[1.5]])
array([[ 1.55064646]])
```

The `penalty` hyperparameter sets the type of regularization term to use. Specifying "l2" indicates that you want SGD to add a regularization term to the cost function equal to half the square of the ℓ_2 norm of the weight vector: this is simply Ridge regression.

Lasso regression

Least Absolute Shrinkage and Selection Operator regression (simply called *Lasso regression*) is another regularized version of Linear Regression: just like Ridge regression, it adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm.

Equation 4-10. Lasso regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

Figure 4-13 shows the same thing as Figure 4-12 but replacing Ridge models with Lasso models and using smaller α values.

¹⁴ Alternatively you can use the `Ridge` class with the "sag" solver. Stochastic Average GD is a variant of SGD. For more details, see this presentation by Mark Schmidt *et al.* from the University of British Columbia: <http://goo.gl/vxVyA2>

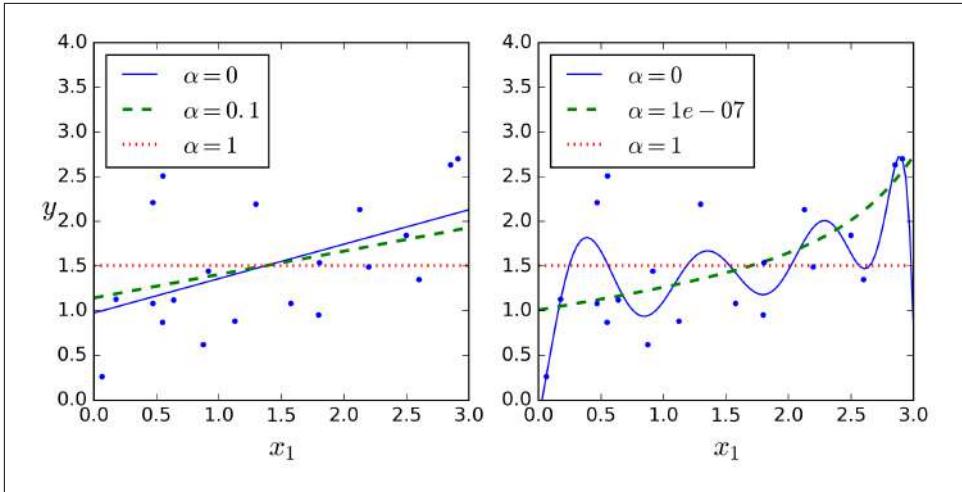


Figure 4-13. Lasso regression

An important characteristic of Lasso regression is that it tends to completely eliminate the weights of the least important features (ie. set them to zero). For example, the dashed line in the right plot on Figure 4-13 (with $\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high degree polynomial features are equal to zero. In other words, Lasso regression automatically performs feature selection and outputs a *sparse model* (ie. with few non-zero feature weights).

You can get an intuition of why this is the case by looking at Figure 4-14: on the top left plot, the background contours (ellipses) represent an unregularized MSE cost function ($\alpha = 0$), and the white circles show the Batch Gradient Descent path with that cost function. The foreground contours (diamonds) represent the ℓ_1 penalty, and the triangles show the BGD path for this penalty only ($\alpha \rightarrow \infty$). Notice how the path first reaches $\theta_1 = 0$ then rolls down a gutter until it reaches $\theta_2 = 0$. On the top right plot, the contours represent the same cost function plus an ℓ_1 penalty with $\alpha = 0.5$. The global minimum is on the $\theta_2 = 0$ axis. BGD first reaches $\theta_2 = 0$ then rolls down the gutter until it reaches the global minimum. The two bottom plots show the same thing but using an ℓ_2 penalty instead. The regularized minimum is closer to $\theta = 0$ than the unregularized minimum, but the weights do not get fully eliminated.

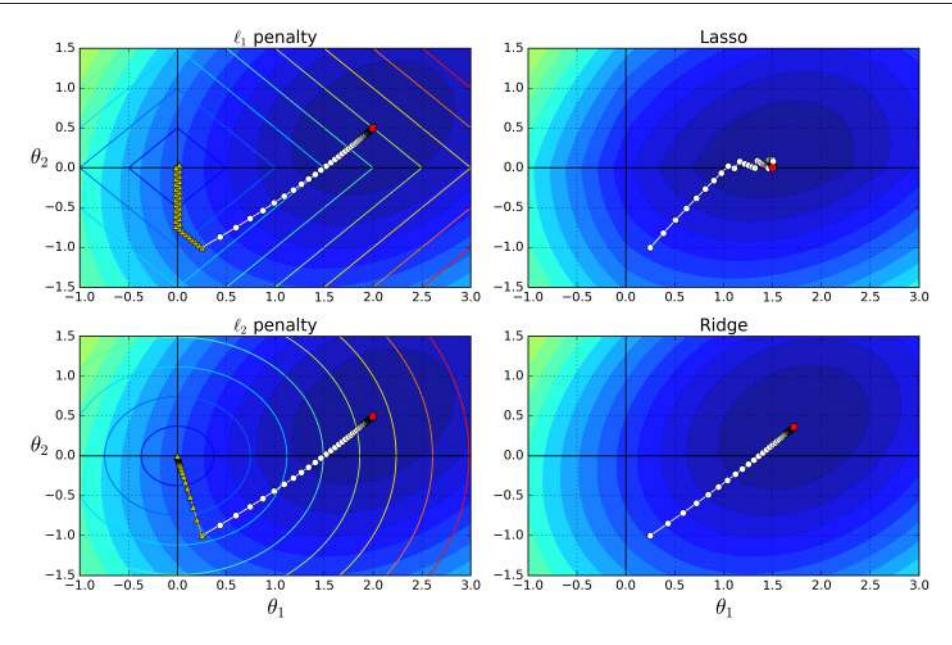


Figure 4-14. Lasso vs Ridge regularization



On the Lasso cost function, the BGD path tends to bounce across the gutter towards the end. This is because the slope changes abruptly at $\theta_2 = 0$. You need to gradually reduce the learning rate in order to actually converge to the global minimum.

The Lasso cost function is not differentiable at $\theta_i = 0$ (for $i = 1, 2, \dots, n$), but Gradient Descent still works fine if you use a *subgradient vector* \mathbf{g} ¹⁵ instead when any $\theta_i = 0$. **Equation 4-11** shows a subgradient vector equation you can use for Gradient Descent with the Lasso cost function.

Equation 4-11. Lasso regression subgradient vector

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

¹⁵ You can think of a subgradient vector at a non-differentiable point as an intermediate vector between the gradient vectors around that point.

Here is a small Scikit-Learn example using the `Lasso` class. Note that you could instead use an `SGDRegressor(penalty="l1")`.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174])
```

Elastic Net

Elastic Net is a middle ground between Ridge regression and Lasso regression: the regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio r : when $r = 0$, Elastic Net is equivalent to Ridge regression, and when $r = 1$, it is equivalent to Lasso regression.

Equation 4-12. Elastic Net cost function

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

So when should you use Linear Regression, Ridge, Lasso or Elastic Net? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. Ridge is a good default, but if you suspect that only few features are actually useful, you should prefer Lasso or Elastic Net since they tend to reduce the useless features' weights down to zero as we discussed above. In general, Elastic Net is preferred over Lasso since Lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Here is a short example using Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds the mix ratio r):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

Early stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called *early stopping*. [Figure 4-15](#) shows a complex model (in this case a high-degree polynomial regression model) being trained using Batch Gradient Descent: as the epochs go by, the algorithm learns and its prediction error (RMSE) on the training set

naturally goes down, and so does its prediction error on the validation set. However, after a while the validation error stops decreasing and actually starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch”.

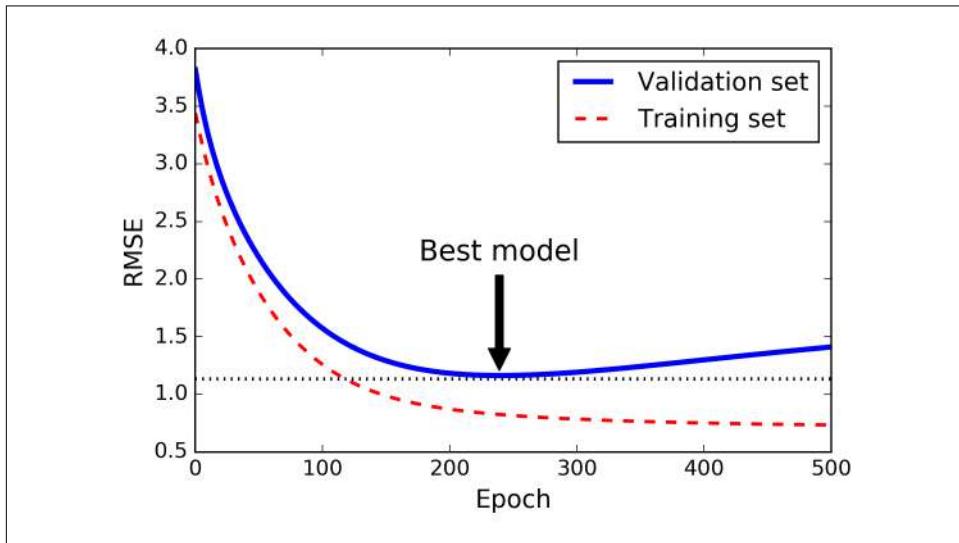


Figure 4-15. Early stopping regularization



With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then rollback the model parameters to the point where the validation error was at a minimum.

Here is a basic implementation of early stopping:

```
from sklearn.base import clone

sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                      learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # continues where it left off
    if sgd_reg.score(X_val_poly_scaled) < minimum_val_error:
        minimum_val_error = sgd_reg.score(X_val_poly_scaled)
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

```

y_val_predict = sgd_reg.predict(X_val_poly_scaled)
val_error = mean_squared_error(y_val_predict, y_val)
if val_error < minimum_val_error:
    minimum_val_error = val_error
    best_epoch = epoch
    best_model = clone(sgd_reg)

```

Note that with `warm_start=True`, when the `fit()` method is called, it just continues training where it left off instead of restarting from scratch.

Logistic Regression

As we discussed in [Chapter 1](#), some regression algorithms can be used for classification as well (and *vice versa*). *Logistic Regression* (also called *logit regression*) is commonly used to estimate the probability that an instance belongs to a particular class (eg. what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labelled “1”), or else it predicts that it does not (ie. it belongs to the negative class, labelled “0”). This makes it a binary classifier.

Estimating probabilities

So how does it work? Just like a Linear Regression model, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the *logistic* of this result:

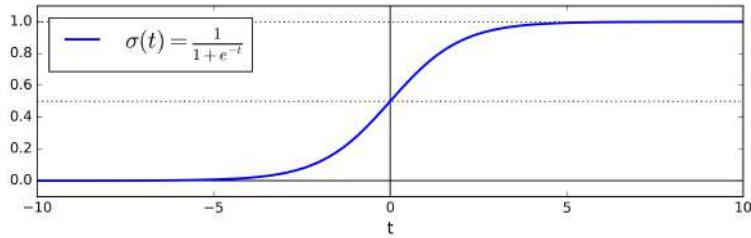
Equation 4-13. Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \cdot \mathbf{x})$$

The logistic (also called the *logit*, noted $\sigma(\cdot)$) is a *sigmoid function* (ie. “S” shaped) which outputs a number between 0 and 1. It is defined as:

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



Once the Logistic Regression model has estimated the probability $\hat{p} = h_\theta(\mathbf{x})$ that an instance \mathbf{x} belongs to the positive class, it can make its prediction \hat{y} easily:

Equation 4-15. Logistic Regression model prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a Logistic Regression model predicts 1 if $\theta^T \cdot \mathbf{x}$ is positive, and 0 if it is negative.

Training and cost function

Good, now you know how a Logistic Regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This idea is captured by the following cost function for a single training instance \mathbf{x} :

Equation 4-16. Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1 - \hat{p}) & \text{if } y = 0. \end{cases}$$

This cost function makes sense because $-\log(t)$ grows very large when t approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand $-\log(t)$ is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

The cost function over the whole training set is simply the average cost over all training instances. It can be written in a single expression (as you can verify easily), called the *log loss*:

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

The bad news is that there is no known closed-form equation to compute the value of θ that minimizes this cost function (there is no equivalent of the Normal Equation). But the good news is that this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough). The partial derivatives of the cost function with regards to the j^{th} model parameter θ_j is given by the following equation:

Equation 4-18. Logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

This equation looks very much like [Equation 4-5](#): for each instance it computes the prediction error and multiplies it by the j^{th} feature value, then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives you can use it in the Batch Gradient Descent algorithm. That's it, you now know how to train a Logistic Regression model. For Stochastic GD you would of course just take one instance at a time, and for Mini-batch you would use a mini-batch at a time.

Decision boundaries

Let's use the Iris dataset to illustrate Logistic Regression: this is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different types: Iris-Setosa, Iris-Versicolour and Iris-Virginica. Let's try to build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let's load the data:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

Now let's train a Logistic Regression model:

```

from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)

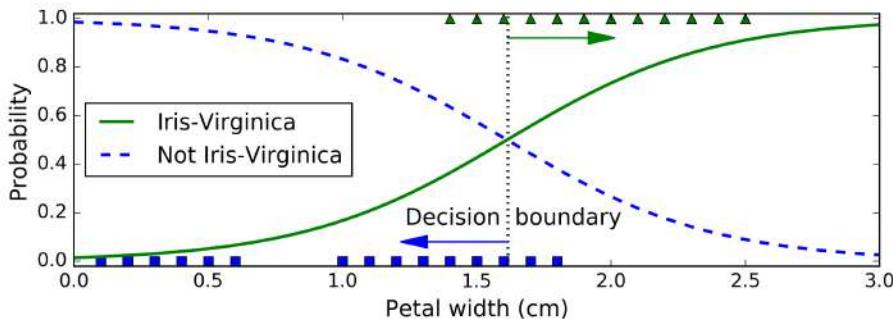
```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 to 3 cm:

```

X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b-", label="Not Iris-Virginica")
# + more Matplotlib code to make the image look pretty

```



The petal width of Iris-Virginica flowers (represented by triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an Iris-Virginica (it outputs a high probability to that class), while below 1 cm it is highly confident that it is not an Iris-Virginica (high probability for the “Not Iris-Virginica” class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely, therefore there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%: if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an Iris-Virginica, or else it will predict that it is not (even if it is not very confident):

```

>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])

```

Figure 4-16 shows the same dataset but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can estimate the probability that a new flower is an Iris-Virginica based on these two features. The dashed line represents the points where the model estimates a 50% probability: this is the model's

decision boundary. Note that it is a linear boundary¹⁶. Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top right line have over 90% chance of being Iris-Virginica according to the model.

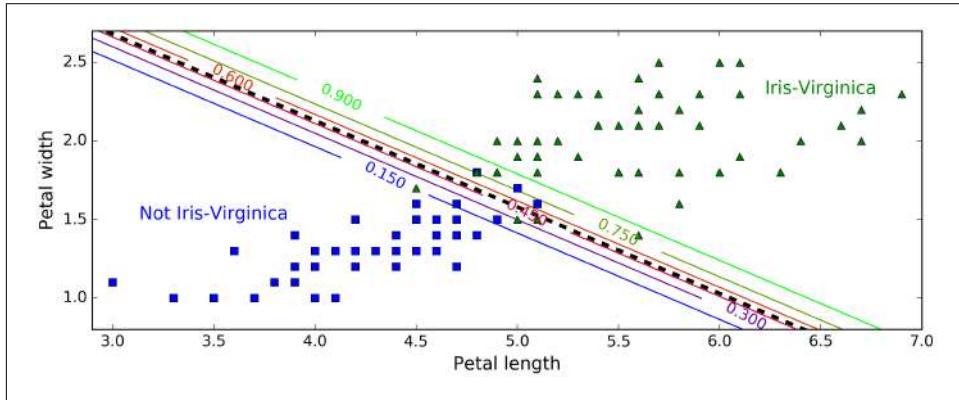


Figure 4-16. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ_1 or ℓ_2 penalties. Scikit-Learn actually adds an ℓ_2 penalty by default.



The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not `alpha` (like in other linear models), but its inverse: `C`. The higher the value of `C` the *less* the model is regularized.

Softmax regression

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in [Chapter 3](#)). This is called *Softmax Regression*, or *Multinomial Logistic Regression*.

The idea is quite simple: when given an instance \mathbf{x} , the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k , then it estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the

¹⁶ It is the set of points \mathbf{x} such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, which defines a straight line.

scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for Linear Regression prediction:

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}$$

Note that each class has its own dedicated parameter vector θ_k . All these vectors are typically stored as rows in a *parameter matrix* Θ .

Once you have computed the score of every class for the instance \mathbf{x} , you can estimate the probability \hat{p}_k that the instance belongs to class k by running the scores through the softmax function: it computes the exponential of every score then normalizes them (dividing by the sum of all the exponentials).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance \mathbf{x} .
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance \mathbf{x} belongs to class k given the scores of each class for that instance.

Just like the Logistic Regression classifier, the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score):

Equation 4-21. Softmax Regression classifier prediction

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k (\theta_k^T \cdot \mathbf{x})$$

- The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of k that maximizes the estimated probability $\sigma(\mathbf{s}(\mathbf{x}))_k$.



The Softmax Regression classifier only predicts one class at a time (ie. it is multiclass, not multioutput) so it should only be used with mutually exclusive classes such as different types of plants. You cannot use it to recognize multiple people in one picture.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the following cost function, called the *cross entropy*, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities match the target classes (we will use it again several times in the following chapters).

Equation 4-22. Cross entropy cost function

$$J(\Theta) = - \frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log (\hat{p}_k^{(i)})$$

- $y_k^{(i)}$ is equal to 1 if the target class for the i^{th} instance is k , otherwise it is equal to 0.

Notice that when there are just two classes ($K = 2$), this cost function is equivalent to the Logistic Regression's cost function (log loss, [Equation 4-17](#)).

Cross entropy

Cross entropy originated from information theory. Suppose you want to efficiently transmit information about the weather every day. If there are 8 options (sunny, rainy, etc.), you could encode each option using 3 bits since $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other 7 options on 4 bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will just be equal to the entropy of the weather itself (ie. its intrinsic unpredictability). But if your assumptions are wrong (eg. if it rains often), cross entropy will be greater by an amount called the *Kullback-Leibler divergence*.

The cross entropy between two probability distributions p and q is defined as $H(p, q) = - \sum_x p(x) \log q(x)$ (at least when the distributions are discrete).

The gradient vector of this cost function with regards to θ_k is given by the following equation:

Equation 4-23. Cross entropy gradient vector for class k

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use Gradient Descent (or any other optimization algorithm) to find the parameter matrix Θ that minimizes the cost function.

Let's use Softmax Regression to classify the Iris flowers into all three classes. Scikit-Learn's `LogisticRegression` uses one-vs-all by default when you train it on more than two classes, but you can set the `multi_class` hyperparameter to `multinomial` to switch it to Softmax Regression instead. You must also specify a solver that supports Softmax Regression, such as the "`lbfgs`" solver (see Scikit-Learn's documentation for more details). It also applies ℓ_2 regularization by default, which you can control using the hyperparameter `C`.

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

So next time you find an Iris with 5 cm long and 2 cm wide petals, you can ask your model to tell you what type of Iris it is, and it will answer Iris-Virginica (class 2) with 94.2% probability (or Iris-Versicolour with 5.8% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07,  5.75276067e-02,  9.42471760e-01]])
```

Figure 4-17 shows the resulting decision boundaries, represented by the background colors: notice that the decision boundaries between any two classes are linear. The figure also shows the probabilities for the Iris-Versicolour class, represented by the curved lines (eg. the line labelled with 0.450 represents the 45% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

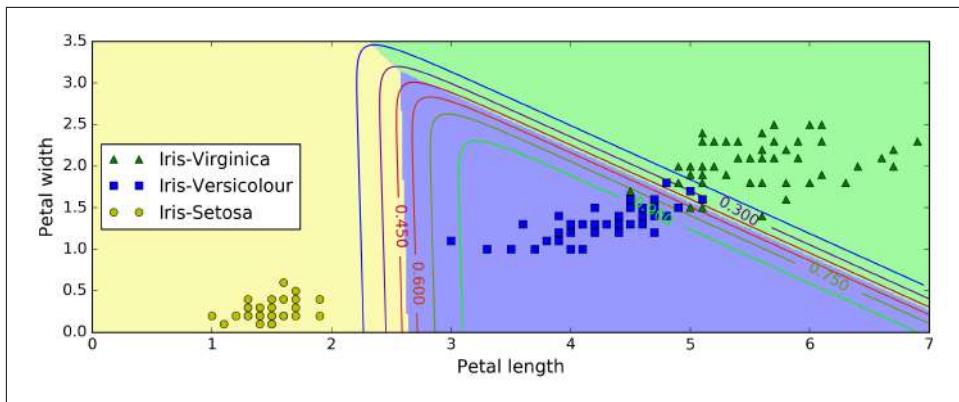


Figure 4-17. Softmax Regression decision boundaries

Exercises

1. What Linear Regression training algorithm can you use if you have a training set with millions of features?
2. Suppose the features in your training set have very different scales: what algorithms might suffer from this, and how? What can you do about it?
3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?
4. Do all Gradient Descent algorithms lead to the same model provided you let them run long enough?
5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch: if you notice that the validation error consistently goes up, what is likely going on? How can you fix this?
6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?
7. Which Gradient Descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest. Which will actually converge? How can you make the others converge as well?
8. Suppose you are using Polynomial regression, you plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?
9. Suppose you are using Ridge regression and you notice that the training error and the validation error are almost equal and fairly high: would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?

10. Why would you want to use:
 - Ridge regression instead of Linear Regression?
 - Lasso instead of Ridge regression?
 - Elastic Net instead of Lasso?
11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime, should you implement two Logistic Regression classifiers or one Softmax Regression classifier?
12. Implement Batch Gradient Descent with early stopping for Softmax Regression (without using Scikit-Learn).

Solutions to these exercises are available in [Appendix A](#).

Support Vector Machines

A *Support Vector Machine* (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or non-linear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have them in their toolbox. They are particularly well suited for classification of complex but small or medium sized datasets.

This chapter will explain the core concepts of SVMs, how to use them and how they work.

Linear SVM classification

The fundamental idea behind SVMs is best explained with some pictures. [Figure 5-1](#) shows part of the Iris dataset that was introduced at the end of [Chapter 4](#). The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers: the model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier: this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

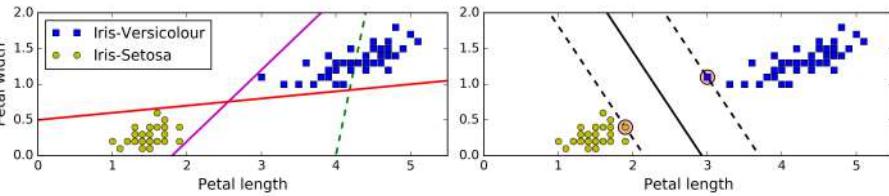


Figure 5-1. Large margin classification

Notice that adding more training instances “off the street” will not affect the decision boundary at all: it is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure 5-1).



SVMs are sensitive to the feature scales, as you can see on Figure 5-2: on the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (eg. using Scikit-Learn’s StandardScaler), the decision boundary looks much better (on the right plot).

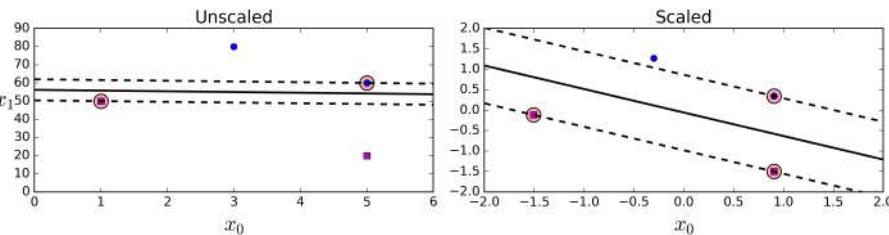


Figure 5-2. Sensitivity to feature scales

Soft margin classification

If we strictly impose that all instances be off the street and on the right side, this is called *hard margin classification*. There are two main issues with hard margin classification: first it only works if the data is linearly separable, and second it is quite sensitive to outliers. Figure 5-3 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin, and on the right the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier, and it will probably not generalize as well.

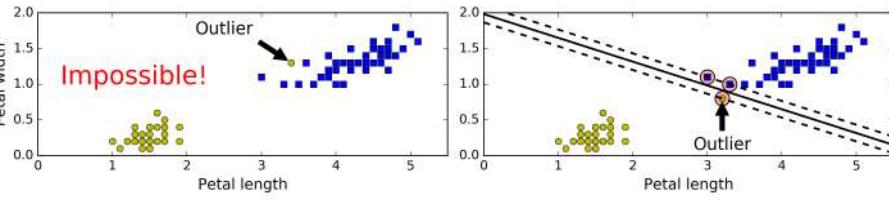


Figure 5-3. Hard margin sensitivity to outliers

To avoid these issues it is preferable to use a more flexible model: the objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (ie. instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

In Scikit-Learn's SVM classes, you can control this balance using the C hyperparameter: a smaller C value leads to a wider street but more margin violations. Figure 5-4 shows the decision boundaries and margins of two soft margin SVM classifiers on a non-linearly separable dataset: on the left, using a high C value the classifier makes fewer margin violations but ends up with a smaller margin. On the right, using a low C value the margin is much larger, but many instances end up on the street. However, it seems likely that the second classifier will generalize better: in fact even on this training set it makes less prediction errors, since most of the margin violations are actually on the correct side of the decision boundary.

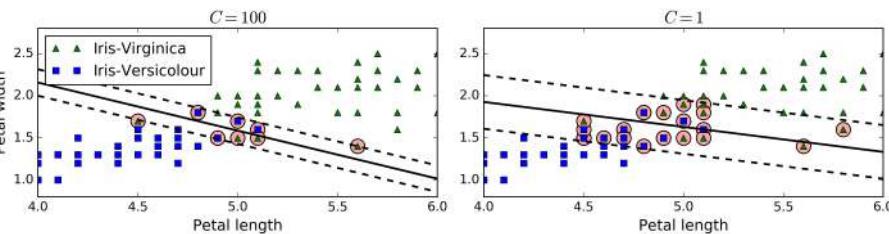


Figure 5-4. Few margin violations vs large margin



If your SVM model is overfitting, you can try regularizing it by reducing C .

The following Scikit-Learn code loads the Iris dataset, scales the features, then trains a linear SVM model (using the `LinearSVC` class with $C = 0.1$ and the *hinge loss* func-

tion, described below) to detect Iris-Virginica flowers. The resulting model is represented on the right of [Figure 5-4](#).

```
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica

svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
))
svm_clf.fit(X_scaled, y)
```

Then, as usual, you can use the model to make predictions:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```



Unlike Logistic Regression classifiers, SVM classifiers do not output probabilities for each class.

Alternatively, you could use the `SVC` class, using `SVC(kernel="linear", C=1)`, but it is much slower, especially with large training sets, so it is not recommended. Another option is to use the `SGDClassifier` class, with `SGDClassifier(loss="hinge", alpha=1/(m*C))`: this applies regular Stochastic Gradient Descent (see [Chapter 4](#)) to train a linear SVM classifier. It does not converge as fast as the `LinearSVC` class, but it can be useful to handle huge datasets that do not fit in memory (out-of-core training), or to handle online classification tasks.



The `LinearSVC` class regularizes the bias term, so you should center the training set first by subtracting its mean. This is automatic if you scale the data using the `StandardScaler`. Moreover, make sure you set the `loss` hyperparameter to "hinge", as it is not the default value. Finally, for better performance you should set the `dual` hyperparameter to `False`, unless there are more features than training instances (we will discuss duality below).

Non-linear SVM classification

Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling non-linear datasets is to add more features, such as polynomial features (as you did in [Chapter 4](#)): in some cases this can result in a linearly separable dataset. Consider the left plot in [Figure 5-5](#): it represents a simple dataset with just one feature x_1 . This dataset is not linearly separable, as you can see. But if you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset is perfectly linearly separable.

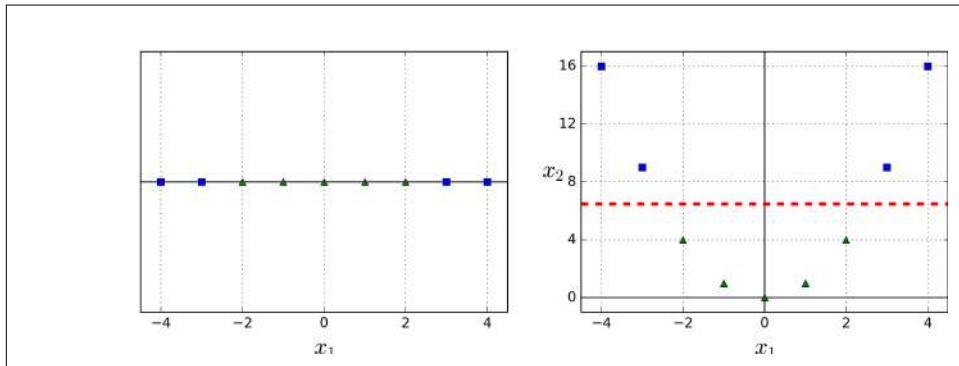


Figure 5-5. Adding features to make a dataset linearly separable

To implement this idea using Scikit-Learn, you can create a `Pipeline` containing a `PolynomialFeatures` transformer (discussed in “[Polynomial regression](#)” on page [145](#)), followed by a `StandardScaler` and a `LinearSVC`. Let’s test this on the moons dataset (see [Figure 5-6](#)).

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline((
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
))

polynomial_svm_clf.fit(X, y)
```

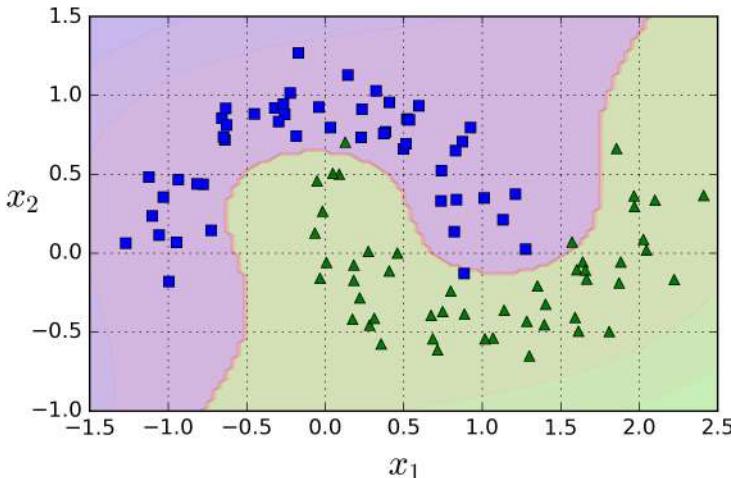


Figure 5-6. Linear SVM classifier using polynomial features

Polynomial kernel

Adding polynomial features is simple to implement and it can work great with all sorts of Machine Learning algorithms (not just SVMs), but at a low polynomial degree it cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making it too slow.

Fortunately, when using SVMs it is possible to apply an almost miraculous mathematical technique called the *kernel trick* (it is explained below). It makes it possible to get the same result as if you added many polynomial features, even with very high degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features since you don't actually add any features. This trick is implemented by the SVC class. Let's test it on the moons dataset.

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
))
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a 3rd degree polynomial kernel. It is represented on the left of Figure 5-7. On the right is another SVM classifier using a 10th degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials vs low-degree polynomials.

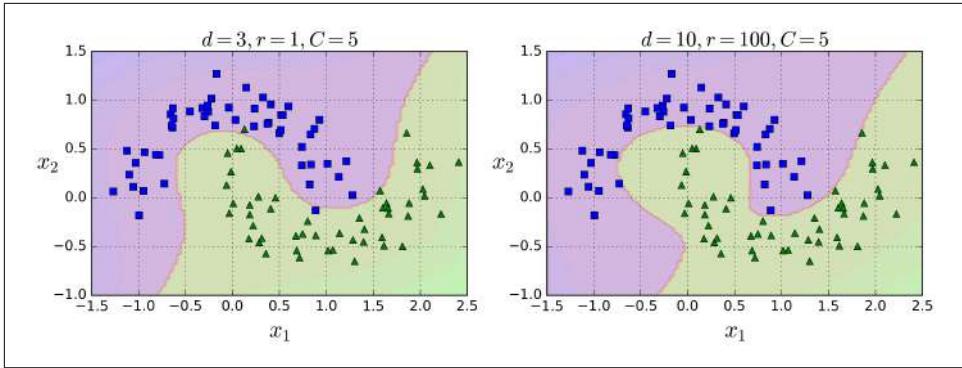


Figure 5-7. SVM classifiers with a polynomial kernel



A common approach to find the right hyperparameter values is to use grid search (see [Chapter 2](#)). It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good intuition of what each hyperparameter actually does can also help you search in the right part of the hyperparameter space.

Adding similarity features

Another technique to tackle non-linear problems is to add features computed using a *similarity function* that measures how much each instance resembles a particular *landmark*. For example, let's take the one-dimensional dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in [Figure 5-8](#)). Next, let's define the similarity function to be the Gaussian *Radial Basis Function* (*RBF*) with $\gamma = 0.3$ (see [Equation 5-1](#)).

Equation 5-1. Gaussian RBF

$$\phi\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

It is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark). Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$: it is located at a distance of 1 from the first landmark, and 2 from the second landmark. Therefore its new features are $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. The plot on the right of [Figure 5-8](#) shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

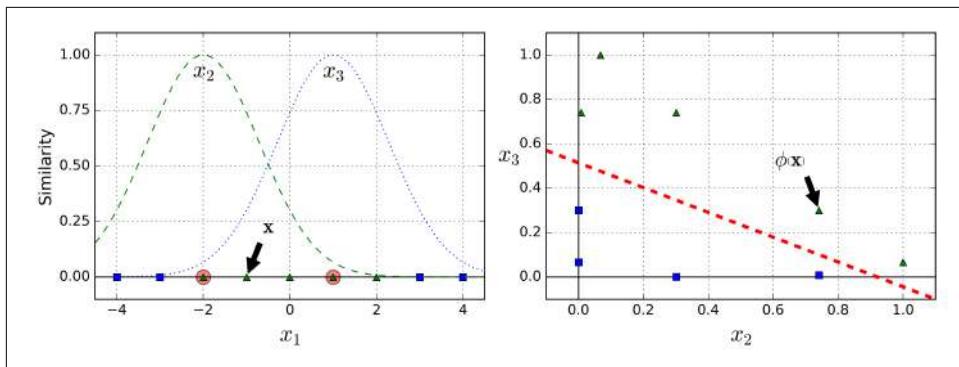


Figure 5-8. Similarity features using the Gaussian RBF

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. This creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with m instances and n features gets transformed into a training set with m instances and m features (assuming you drop the original features). If your training set is very large you end up with an equally large number of features.

Gaussian RBF kernel

Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm but it may be computationally expensive to compute all the additional features, especially on large training sets. However, once again the kernel trick does its magic when using SVMs: it makes it possible to obtain a similar result as if you had added many similarity features, without actually having to add them. Let's try the Gaussian RBF kernel using the SVC class.

```
rbf_kernel_svm_clf = Pipeline(
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
)
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented on the bottom left of Figure 5-9. The other plots show models trained with different values of hyperparameters γ and C . Increasing γ makes the bell-shape curve narrower (see the left plot of Figure 5-8), and as a result each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small γ value makes the bell-shaped curve wider, so instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization

hyperparameter: if your model is overfitting, you should reduce it, and if it is underfitting, you should increase it (similar to the C hyperparameter).

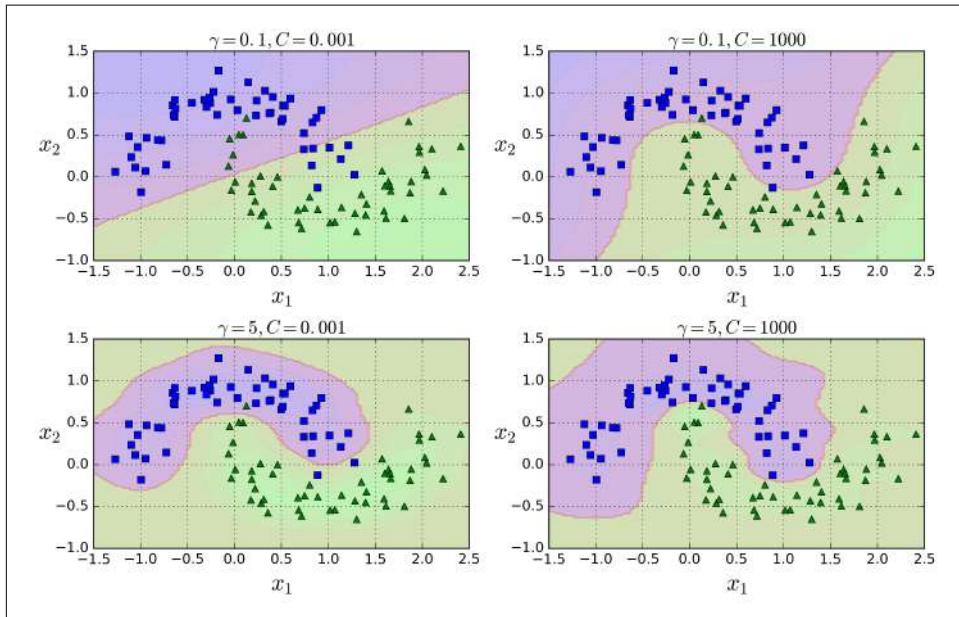


Figure 5-9. SVM classifiers using an RBF kernel

Other kernels exist but are used much more rarely, for example some kernels are specialized for specific data structures: *string kernels* are sometimes used when classifying text documents or DNA sequences (eg. using the *string subsequence kernel* or kernels based on the *Levenshtein distance*).



With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first (remember that `LinearSVC` is much faster than `SVC(kernel='linear')`), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should try the Gaussian RBF kernel as well: it works well in most cases. Then if you have spare time and computing power, you can also experiment with a few other kernels using cross validation and grid search, especially if there are kernels specialized for your training set's data structure.

Computational complexity

The `LinearSVC` class is based on the *liblinear* library which implements an optimized algorithm for linear SVMs¹. It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features: its training time complexity is roughly $O(m \times n)$.

The algorithm takes longer if you require a very high precision: this is controlled by the tolerance hyperparameter ϵ (called `tol` in Scikit-Learn). It will take $O(1/\log(\epsilon))$ time to reach ϵ -accuracy (ie. to be within ϵ of the optimum). So if you divide `tol` by 100 to get 100 times more accuracy, the algorithm will run roughly 10 times longer. In most classification tasks, the default tolerance is fine.

The `SVC` class is based on the *libsvm* library which implements an algorithm that supports the kernel trick². The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (eg. hundreds of thousands of instances). This algorithm is perfect for complex but small or medium training sets. However it scales well with the number of features, especially with *sparse features* (ie. when each instance has few non-zero features): in this case the algorithm scales roughly with the average number of non-zero features. [Table 5-1](#) compares Scikit-Learn's SVM classification classes.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
<code>LinearSVC</code>	$O(m \times n / \log(\epsilon))$	No	Yes	No
<code>SGDClassifier</code>	$O(m \times n / \epsilon^2)$	Yes	Yes	No
<code>SVC</code>	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM regression

As we mentioned earlier, the SVM algorithm is quite versatile: not only does it support linear and non-linear classification, but it also supports linear and non-linear regression. The trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM regression tries to fit as many instances as possible *on* the street while limiting margin violations (ie. instances *off* the street). The width of the street is controlled by a hyperparameter ϵ . [Figure 5-10](#) shows two linear SVM regression models trained on some random lin-

¹ "A Dual Coordinate Descent Method for Large-scale Linear SVM", Lin *et al.* (2008), <http://goo.gl/R635CH>

² "Sequential Minimal Optimization" (SMO), Platt (1998), Microsoft Research, <http://goo.gl/a8HkE3>

ear data, one with a large margin ($\epsilon = 1.5$) and the other with a small margin ($\epsilon = 0.5$).

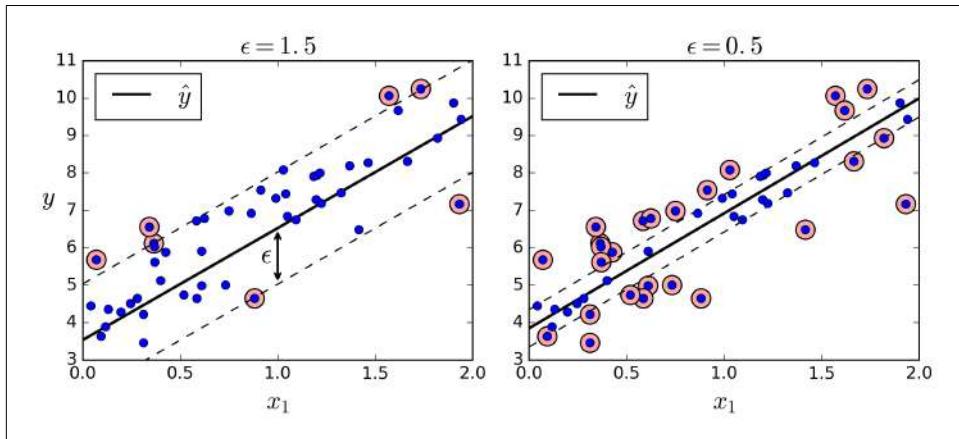


Figure 5-10. SVM regression

Adding more training instances within the margin does not affect the model's predictions: the model is said to be ϵ -insensitive.

You can use Scikit-Learn's `LinearSVR` class to perform linear SVM regression. The following code produces the model represented on the left of Figure 5-10 (the training data should be scaled and centered first):

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

To tackle non-linear regression tasks, you can use a kernelized SVM model. For example, Figure 5-11 shows SVM regression on a random quadratic training set, using a 2nd degree polynomial kernel. There is little regularization on the left plot (ie. a large C value), and much more regularization on the right plot (ie. a small C value).

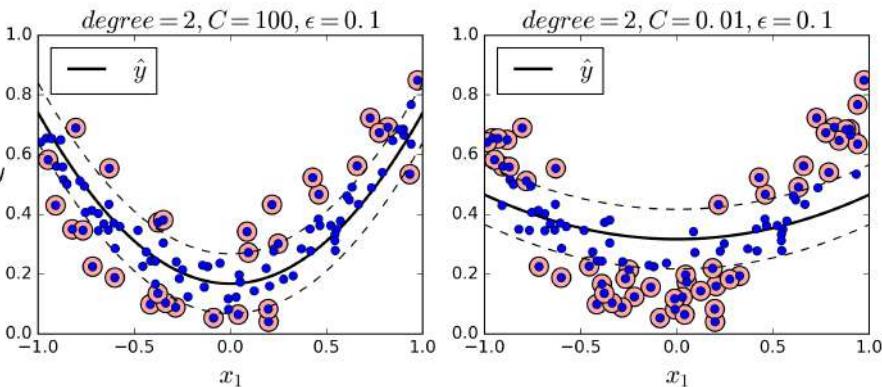


Figure 5-11. SVM regression using a 2nd degree polynomial kernel

The following code produces the model represented on the left of Figure 5-11 using Scikit-Learn's SVR class (which supports the kernel trick). The SVR class is the regression equivalent of the SVC class, and the LinearSVR class is the regression equivalent of the LinearSVC class. The LinearSVR class scales linearly with the size of the training set (just like the LinearSVC class), while the SVR class gets much too slow when the training set grows large (just like the SVC class).

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



SVMs can also be used for outlier detection, see Scikit-Learn's documentation for more details.

Under the hood

This section explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. You can safely skip it and go straight to the exercises at the end of this chapter if you are just getting started with Machine Learning, and come back later when you want to get a deeper understanding of SVMs.

First, a word about notations: in Chapter 4 we used the convention of putting all the model parameters in one vector θ , including the bias term θ_0 and the input feature weights θ_1 to θ_n , and adding a bias input $x_0 = 1$ to all instances. In this chapter, we will use a different convention, which is more convenient (and more frequent) when

dealing with SVMs: the bias term will be called b and the feature weights vector will be called \mathbf{w} . No bias feature will be added to the input feature vectors.

Decision function and predictions

The linear SVM classifier model predicts the class of a new instance \mathbf{x} by simply computing the decision function $\mathbf{w}^T \cdot \mathbf{x} + b = w_1x_1 + \dots + w_nx_n + b$: if the result is positive, the predicted class \hat{y} is the positive class (1), or else it is the negative class (0).

Equation 5-2. Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

Figure 5-12 shows the decision function that corresponds to the model on the right of Figure 5-4: it is a 2-dimensional plane since this dataset has 2 features (petal width and petal length). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of two planes, which is a straight line (represented by the thick solid line).³

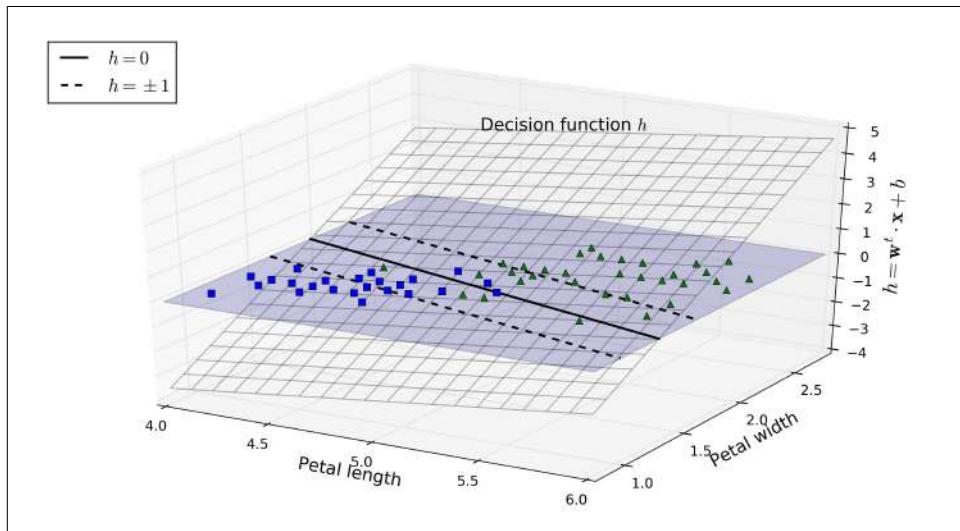


Figure 5-12. Decision function for the Iris dataset

³ More generally, when there are n features, the decision function is an n -dimensional *hyperplane*, and the decision boundary is an $(n - 1)$ -dimensional hyperplane.

The dashed lines represent the points where the decision function is equal to 1 or -1: they are parallel and at equal distance to the decision boundary, forming a margin around it. Training a linear SVM classifier means finding the value of \mathbf{w} and b that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

Training objective

Consider the slope of the decision function: it is equal to the norm of the weight vector: $\|\mathbf{w}\|$. If we divide this slope by 2, the points where the decision function is equal to ± 1 are going to be twice further away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. Perhaps this is easier to visualize in 2D on [Figure 5-13](#). The smaller the weight vector \mathbf{w} , the larger the margin.

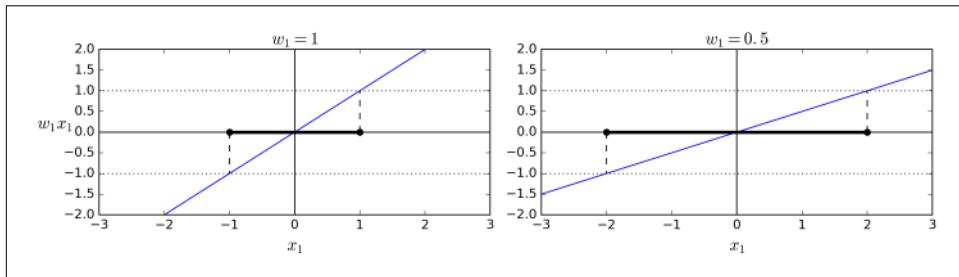


Figure 5-13. A smaller weight vector results in a larger margin

So we want to minimize $\|\mathbf{w}\|$ to get a large margin. However, if we also want to avoid any margin violation (hard margin), then we need the decision function to be greater than 1 for all positive training instances, and lower than -1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the following *constrained optimization* problem.

Equation 5-3. Hard margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} \\ & \text{subject to} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$



We are minimizing $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$, which is equal to $\frac{1}{2}\|\mathbf{w}\|^2$, rather than minimizing $\|\mathbf{w}\|$: this is because it will give the same result (since the value of \mathbf{w} and b that minimize a value also minimize half of its square), but $\frac{1}{2}\|\mathbf{w}\|^2$ has a nice and simple derivative (it is just \mathbf{w}) while $\|\mathbf{w}\|$ is not differentiable at $\mathbf{w} = \mathbf{0}$. Optimization algorithms work much better on differentiable functions.

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance⁴: $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin. We now have two conflicting objectives: making the slack variables as small as possible to reduce the margin violations, and making $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$ as small as possible to increase the margin. This is where the C hyperparameter comes in: it allows us to define the trade-off between these two objectives. This gives us the following constrained optimization problem.

Equation 5-4. Soft margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Quadratic Programming

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *Quadratic Programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems

⁴ Zeta (ζ) is the 8th letter of the Greek alphabet.

using a variety of techniques that are outside the scope of this book⁵. The general problem formulation is given by [Equation 5-5](#):

Equation 5-5. Quadratic Programming problem

$$\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2} \mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p}$$

subject to $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$

where $\begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix,} \\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector,} \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector.} \end{cases}$

Note that the expression $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$ actually defines n_c constraints: $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$ for $i = 1, 2, \dots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the i^{th} row of \mathbf{A} and $b^{(i)}$ is the i^{th} element of \mathbf{b} .

You can easily verify that if you set the QP parameters in the following way, you get the hard margin linear SVM classifier objective:

- $n_p = n + 1$, where n is the number of features (the $+1$ is for the bias term),
- $n_c = m$, where m is the number of training instances,
- \mathbf{H} is the $n_p \times n_p$ identity matrix, except with a zero in the top left cell (to ignore the bias term),
- $\mathbf{f} = \mathbf{0}$, an n_p -dimensional vector full of 0s,
- $\mathbf{b} = \mathbf{1}$, an n_c -dimensional vector full of 1s,
- $\mathbf{a}^{(i)} = -t^{(i)}\dot{\mathbf{x}}^{(i)}$, where $\dot{\mathbf{x}}^{(i)}$ is equal to $\mathbf{x}^{(i)}$ with an extra bias feature $\dot{x}_0 = 1$.

So one way to train a hard margin linear SVM classifier is just to use an off-the-shelf QP solver by passing it the parameters above. The resulting vector \mathbf{p} will contain the bias term $b = p_0$ and the feature weights $w_i = p_i$ for $i = 1, 2, \dots, m$. Similarly you can

⁵ To learn more about Quadratic Programming, you can start by reading “Convex Optimization” by S. Boyd and L. Vandenberghe (<http://goo.gl/FGXuLw>) or watch this series of video lectures by R. Brown: <http://goo.gl/rTo3Af>

use a QP solver to solve the soft margin problem (see the exercises at the end of the chapter).

However to use the kernel trick we are going to look at a different constrained optimization problem.

The dual problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*. The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can even have the same solutions as the primal problem. Luckily, the SVM problem happens to meet these conditions⁶, so you can choose to solve the primal problem or the dual problem, both will have the same solution. [Equation 5-6](#) shows the dual form of the linear SVM objective (if you are interested in knowing how to derive the dual problem from the primal problem, see [???](#)).

Equation 5-6. Dual form of the linear SVM objective

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ \text{subject to} \quad & \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Once you find the vector $\hat{\alpha}$ that minimizes this equation (using a QP solver), you can compute $\hat{\mathbf{w}}$ and \hat{b} that minimize the primal problem by using [Equation 5-7](#):

Equation 5-7. From the dual solution to the primal solution

$$\begin{aligned} \hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} (\hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}) \right) \end{aligned}$$

The dual problem is faster to solve than the primal when the number of training instances is smaller than the number of features. More importantly, it makes the kernel trick possible, while the primal does not. So what is this kernel trick anyway?

⁶ The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.

Kernelized SVM

Suppose you want to apply a 2nd degree polynomial transformation to a 2-dimensional training set (such as the moons training set), then train a linear SVM classifier on the transformed training set. [Equation 5-8](#) shows the 2nd degree polynomial mapping function ϕ that you want to apply:

Equation 5-8. Second degree polynomial mapping

$$\phi(\mathbf{x}) = \phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Notice that the transformed vector is 3-dimensional instead of 2-dimensional. Now let's look at what happens to a couple 2-dimensional vectors \mathbf{a} and \mathbf{b} if we apply this 2nd degree polynomial mapping then compute the dot product of the transformed vectors:

Equation 5-9. Kernel trick for a 2nd degree polynomial mapping

$$\begin{aligned} \phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 \\ &= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2 \end{aligned}$$

How about that? The dot product of the transformed vectors is equal to the square of the dot product of the original vectors: $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$.

Now here is the key insight: if you apply the transformation ϕ to all training instances, then the dual problem (see [???](#)) will contain the dot product $\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(j)})$. But if ϕ is the 2nd degree polynomial transformation defined above ([Equation 5-8](#)), then you can replace this dot product of transformed vectors simply by $(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^2$. So you don't actually need to transform the training instances at all: just replace the dot product by its square in [???](#). The result will be strictly the same as if you went through the trouble of actually transforming the training set then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient. This is the essence of the kernel trick.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ is called a 2nd degree *polynomial kernel*. In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$ based only on the original vectors \mathbf{a} and \mathbf{b} , without having to compute (or even to know about) the transformation ϕ . [Equation 5-10](#) lists some of the most commonly used kernels.

Equation 5-10. Common kernels

$$\text{Linear: } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polynomial: } K(\mathbf{a}, \mathbf{b}) = \left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r \right)^d$$

$$\text{Gaussian RBF: } K(\mathbf{a}, \mathbf{b}) = \exp \left(-\gamma \| \mathbf{a} - \mathbf{b} \|^2 \right)$$

$$\text{Sigmoid: } K(\mathbf{a}, \mathbf{b}) = \tanh \left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r \right)$$

Mercer's theorem

According to *Mercer's theorem*, if a function $K(\mathbf{a}, \mathbf{b})$ respects a few mathematical conditions called *Mercer's conditions* (K must be continuous, symmetric in its arguments so $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function ϕ that maps \mathbf{a} and \mathbf{b} into another space (possibly with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$. So you can use K as a kernel since you know ϕ exists, even if you don't know what ϕ is. In the case of the Gaussian RBF kernel, it can be shown that ϕ actually maps each training instance to an infinite-dimensional space, so it's a good thing you don't need to actually perform the mapping!

Note that some frequently used kernels (such as the Sigmoid kernel) don't respect all Mercer's conditions, yet they generally work well in practice.

There is still one loose end we must tie: [Equation 5-7](#) shows how to go from the dual solution to the primal solution in the case of a linear SVM classifier, but if you apply the kernel trick you end up with equations that include $\phi(\mathbf{x}^{(i)})$. In fact, $\widehat{\mathbf{w}}$ must have the same number of dimensions as $\phi(\mathbf{x}^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\widehat{\mathbf{w}}$? Well the good news is that you can plug in the formula for $\widehat{\mathbf{w}}$ from [Equation 5-7](#) into the deci-

sion function for a new instance $\mathbf{x}^{(n)}$, and you get an equation with only dot products between input vectors. This makes it possible to use the kernel trick, once again.

Equation 5-11. Making predictions with a kernelized SVM

$$\begin{aligned} h_{\hat{\mathbf{w}}, \hat{b}}(\phi(\mathbf{x}^{(n)})) &= \hat{\mathbf{w}}^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} = \left(\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} \\ &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} (\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(n)})) + \hat{b} \\ &= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b} \end{aligned}$$

Note that since $\alpha^{(i)} \neq 0$ only for support vectors, making predictions involves computing the dot product of the new input vector $\mathbf{x}^{(n)}$ with only the support vectors, not all the training instances. Of course, you also need to compute the bias term \hat{b} , using the same trick.

Equation 5-12. Computing the bias term using the kernel trick

$$\begin{aligned} \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \hat{\mathbf{w}}^T \cdot \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \cdot \phi(\mathbf{x}^{(i)}) \right) \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right) \end{aligned}$$

If you are starting to get a headache, it's perfectly normal: it's an unfortunate side effect of the kernel trick.

Online SVMs

Before concluding this chapter, let's take a quick look at online SVM classifiers (recall that online learning means learning incrementally, typically as new instances arrive).

For linear SVM classifiers, one method is to use Gradient Descent (eg. using `SGDClassifier`) to minimize the following cost function, which is derived from the primal

problem. Unfortunately it converges much more slowly than the methods based on QP.

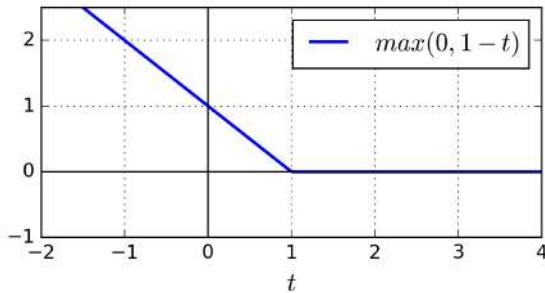
Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b))$$

The first sum in the cost function will push the model to have a small weight vector \mathbf{w} , leading to a larger margin. The second sum computes the total of all margin violations: an instance's margin violation is equal to 0 if it is located off the street and on the correct side, or else it is proportional to the distance to the correct side of the street. Minimizing this term ensures that the model makes as little margin violations as possible.

Hinge loss

The function $\max(0, 1 - t)$ is called the *hinge loss* function (represented below). It is equal to 0 when $t \geq 1$. Its derivative (slope) is equal to -1 if $t < 1$ and 0 if $t > 1$. It is not differentiable at $t = 1$, but just like for Lasso regression (“[Lasso regression](#)” on page 154) you can still use Gradient Descent using any *subderivative* at $t = 0$ (ie. any value between -1 and 0).



It is also possible to implement online kernelized SVMs, for example using “Incremental and Decremental SVM learning”⁷, or “Fast Kernel Classifiers with Online and Active Learning”⁸. However, these are implemented in Matlab and C++. For large

⁷ Cauwenberghs, Poggio (2001), NIPS: <http://goo.gl/JEqVui>

⁸ Bordes, Ertekin, Weston, Bottou (2005), Journal of Machine Learning Research: <http://goo.gl/42K9kG>

scale non-linear problems, you may want to consider using neural networks instead (see [Part II](#)).

Exercises

1. What is the fundamental idea behind Support Vector Machines?
2. What is a support vector?
3. Why is it important to scale the inputs when using SVMs?
4. Can an SVM classifier output a confidence score when it classifies an instance? What about a probability?
5. Should you use the primal or the dual form of the SVM problem to train a model on a training set with millions of instances and hundreds of features?
6. Say you trained an SVM classifier with an RBF kernel. It seems to underfit the training set: should you increase or decrease γ (`gamma`)? What about C ?
7. How should you set the QP parameters (\mathbf{H} , \mathbf{f} , \mathbf{A} and \mathbf{b}) to solve the soft margin linear SVM classifier problem using an off-the-shelf QP solver?
8. Train a `LinearSVC` on a linearly separable dataset. Then train an `SVC` and a `SGDClassifier` on the same dataset. See if you can get them to produce roughly the same model.
9. Train an SVM classifier on the MNIST dataset. Since SVM classifiers are binary classifiers, you will need to use one-vs-all to classify all 10 digits. You may want to tune the hyperparameters using small validation sets to speed up the process. What accuracy can you reach?
10. Train an SVM regressor on the California housing dataset.

Solutions to these exercises are available in [Appendix A](#).

CHAPTER 6

Decision Trees

Like SVMs, *Decision Trees* are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are very powerful algorithms, capable of fitting complex datasets: for example, in [Chapter 2](#) you trained a `DecisionTreeRegressor` model on the California Housing dataset, fitting it perfectly (actually overfitting it).

Decision trees are also the fundamental components of Random Forests (see [Chapter 7](#)) which are among the most powerful Machine Learning algorithms available today.

In this chapter we will start by discussing how to train, visualize and make predictions with Decision Trees, then we will go through the CART training algorithm used by Scikit-Learn, and we will discuss how to regularize trees and use them for regression tasks. Finally, we will discuss some of the limitations of Decision Trees.

Training and visualizing a Decision Tree

To understand Decision Trees, let's just build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the Iris dataset (see [Chapter 4](#)):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

You can visualize the trained Decision Tree by first using the `export_graphviz()` method to output a graph definition file `iris_tree.dot`.

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file=image_path("iris_tree.dot"),
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

Then you can convert this `.dot` file to a variety of formats such as PDF or PNG using the `dot` command line tool from the `graphviz` package¹. This command line converts the `.dot` file to a `.png` image file.

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

This is what your first decision tree looks like:

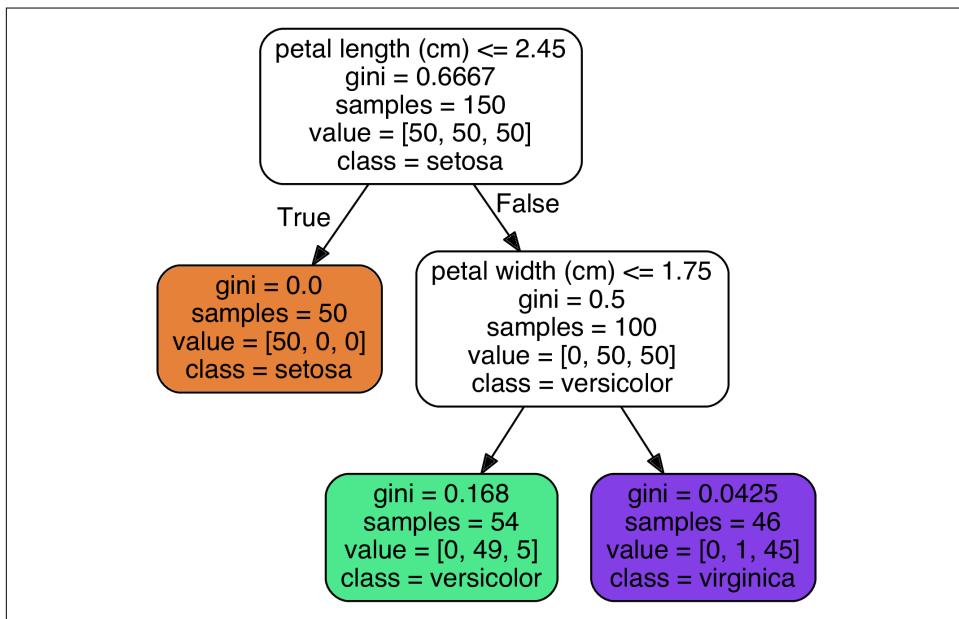


Figure 6-1. Iris Decision Tree

¹ Graphviz is an open source graph visualization software, available at <http://www.graphviz.org/>.

Making predictions

Let's see how the tree represented in [Figure 6-1](#) makes predictions. Suppose you find an Iris flower and you want to classify it. You start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm? If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a *leaf node* (ie. it does not have any children nodes), so it does not ask any question: you can simply look at the predicted class for that node: the decision tree predicts that your flower is an Iris-Setosa (`class=setosa`).

Now suppose you find another flower but this time the petal length is greater than 2.45 cm: you must move down to the root's right child node (depth 1, right), which is not a leaf node so it asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an Iris-Versicolour (depth 2, left). If not, it is likely an Iris-Virginica (depth 2, right). It's really that simple.



One of the many qualities of Decision Trees is that they require very little data preparation. In particular, they don't require feature scaling or centering at all.

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), among which 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom right node applies to 0 Iris-Setosa, 1 Iris-Versicolour and 45 Iris-Virginica. Finally, a node's `gini` attribute measures its *impurity*: a node is “pure” (`gini=0`) if all training instances it applies to belong to the same class. For example, since the depth 1 left node applies only to Iris-Setosa training instances, it is pure and its `gini` score is 0. [Equation 6-1](#) shows how the training algorithm computes the gini score G_i of the i^{th} node. For example, the depth 2 left node has a gini score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$. Another *impurity measure* is discussed below.

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

where $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.



Scikit-Learn uses the CART algorithm which produces only *binary trees*: non-leaf nodes always have 2 children (ie. questions only have yes/no answers). However, other algorithms such as ID3 can produce Decision Trees with nodes that have more than 2 children.

Figure 6-2 shows this Decision Tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the left area is pure (only Iris-Setosa), it cannot be split any further. However, the right area is impure so the depth 1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the Decision Tree stops right there. However, if you set `max_depth` to 3, then the two depth 2 nodes would each add another decision boundary (represented by the dotted lines).

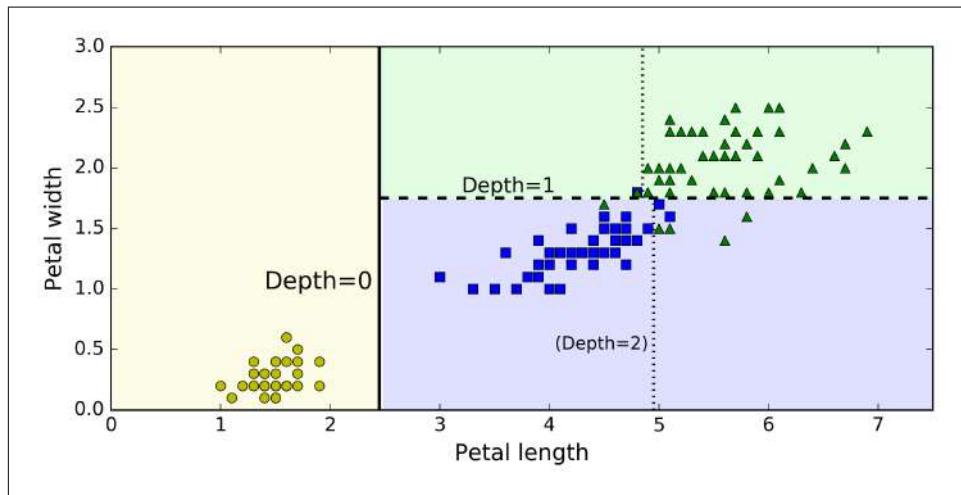


Figure 6-2. Decision Tree decision boundaries

Model interpretation: white box vs black box

As you can see Decision Trees are fairly intuitive and their decisions are easy to interpret. Such models are often called *white box models*. In contrast, as we will see, Random Forests or Neural Networks are generally considered *black box models*: they make great predictions and you can easily check the calculations that they performed to make these predictions, nevertheless it is usually hard to explain in simple terms why the predictions were made. For example, if a Neural Network says that a particular person appears on a picture, it is hard to know what actually contributed to this prediction: did the model recognize that person's eyes? Her mouth? Her nose? Her Shoes? Or even the couch that she was sitting on? Conversely, Decision Trees provide

nice and simple classification rules that can even be applied manually if need be (eg. for flower classification).

Estimating class probabilities

A Decision Tree can also estimate the probability that an instance belongs to a particular class k : first it traverses the tree to find the leaf node for this instance, then it returns the ratio of training instances of class k in this node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth 2 left node, so the Decision Tree should output the following probabilities: 0% for Iris-Setosa (0/54), 90.7% for Iris-Versicolour (49/54), and 9.3% for Iris-Virginica (5/54). And of course if you ask it to predict the class, it should output Iris-Versicolour (class 1) since it has the highest probability. Let's check this:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom right rectangle of [Figure 6-2](#), for example if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an Iris-Virginica in this case).

The CART training algorithm

Scikit-Learn uses the *Classification And Regression Tree* (CART) algorithm to train Decision Trees (also called “growing” trees). The idea is really quite simple: the algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (eg. “petal length ≤ 2.45 cm”). How does it choose k and t_k ? The answer is that it searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). The cost function that the algorithm tries to minimize is given by [Equation 6-2](#).

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} & \text{measures the impurity of the left/right subset,} \\ m_{\text{left/right}} & \text{is the number of instances in the left/right subset.} \end{cases}$

Once it has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets and so on, recursively. It stops recursing once it rea-

ches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described below) control additional stopping conditions (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` and `max_leaf_nodes`).



As you can see, the CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a reasonably good solution, but it is not guaranteed to be the optimal solution.

Unfortunately, finding the optimal tree is known to be an *NP-Complete* problem²: it requires $O(\exp(m))$ time, making the problem intractable even for fairly small training sets. This is why we must settle for a “reasonably good” solution.

Computational complexity

Making predictions requires traversing the Decision Tree from the root to a leaf. Decision Trees are generally approximately balanced, so traversing the Decision Tree requires going through roughly $O(\log_2(m))$ nodes³. Since each node only requires checking the value of one feature, the overall prediction complexity is just $O(\log_2(m))$, independent of the number of features. So predictions are very fast, even when dealing with large training sets.

However, the training algorithm compares all features (or less if `max_features` is set) on all samples at each node. This results in a training complexity of $O(n \times m \log(m))$. For small training sets (less than a few thousand instances), Scikit-Learn can speed up training by presorting the data (set `presort=True`), but this slows down training considerably for larger training sets.

Gini impurity or entropy?

By default, the Gini impurity measure is used, but you can select the *entropy* impurity measure instead by setting the `criterion` hyperparameter to "entropy". The concept

² P is the set of problems that can be solved in polynomial time. NP is the set of problems whose solutions can be verified in polynomial time. An NP-Hard problem is a problem that any NP problem can be reduced to in polynomial time. An NP-Complete problem is both NP and NP-Hard. A major open mathematical question is whether or not P=NP. If P≠NP (which seems likely), then no polynomial algorithm will ever be found for any NP-Complete problem.

³ \log_2 is the binary logarithm. It is equal to $\log_2(m) = \log(m)/\log(2)$.

of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. It later spread to a wide variety of domains, including Shannon's *information theory* where it measures the average information content of a message⁴: entropy is zero when all messages are identical. In Machine Learning, it is frequently used as an impurity measure: a set's entropy is zero when it contains instances of only one class. **Equation 6-3** shows the definition of the entropy of the i^{th} node. For example, the depth 2 left node in **Figure 6-1** has an entropy equal to $-\frac{49}{54} \log\left(\frac{49}{54}\right) - \frac{5}{54} \log\left(\frac{5}{54}\right) \approx 0.31$.

Equation 6-3. Entropy

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log(p_{i,k})$$

So should you use Gini impurity or entropy? The truth is most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees⁵.

Regularization hyperparameters

Decision Trees make very little assumptions about the training data (as opposed to linear models which obviously assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely, and most likely overfitting it. Such a model is often called a *non-parametric model*, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a *parametric model* such as a linear model has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

To avoid overfitting the training data, you need to restrict the Decision Tree's freedom during training: as you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the

⁴ A reduction of entropy is often called an *information gain*.

⁵ See S. Raschka's interesting analysis for more details: <http://goo.gl/UndTrO>

`max_depth` hyperparameter (the default value is `None`, which means unlimited). Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the Decision Tree: `min_samples_split` (the minimum number of samples a node must have before it can be split), `min_samples_leaf` (the minimum number of samples a leaf node must have), `min_weight_fraction_leaf` (same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances), `max_leaf_nodes` (maximum number of leaf nodes) and `max_features` (maximum number of features that are evaluated for splitting at each node). Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.



Other algorithms work by first training the Decision Tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not *statistically significant*: standard statistical tests such as the χ^2 test are used to estimate the probability that the improvement is purely the result of chance (which is called the *null hypothesis*). If this probability, called the *p-value*, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Figure 6-3 shows two Decision Trees trained on the moons dataset (introduced in Chapter 5). On the left, the Decision Tree is trained with the default hyperparameters (ie. no restrictions), and on the right the Decision Tree is trained with `min_samples_leaf=4`. It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.

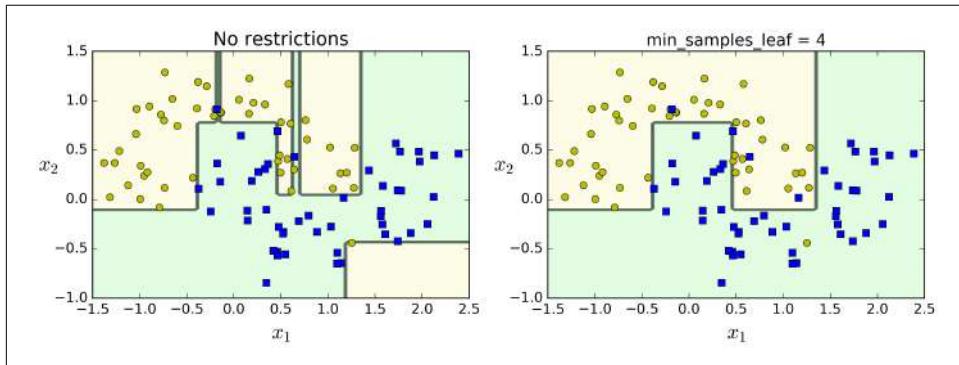


Figure 6-3. Regularization using `min_samples_leaf`

Regression

Decision Trees are also capable of performing regression tasks. Let's build a regression tree using Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = DecisionTreeRegressor(max_depth=2)  
tree_reg.fit(X, y)
```

The resulting tree is represented on [Figure 6-4](#).

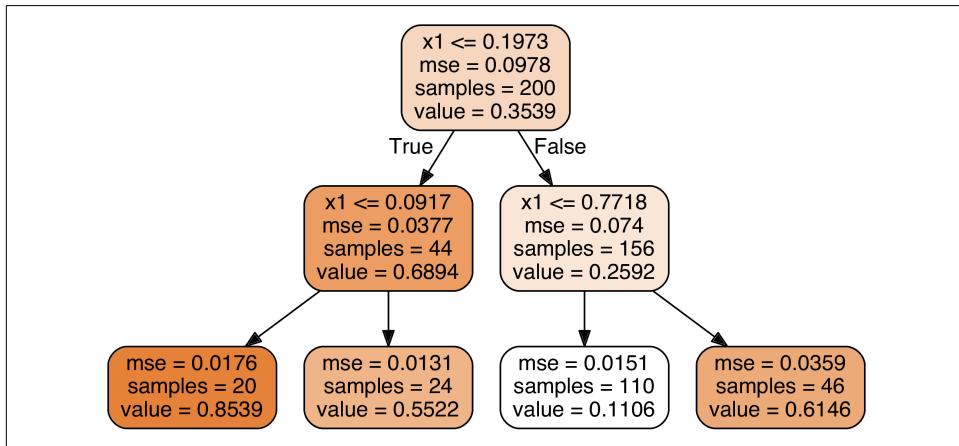


Figure 6-4. A Decision Tree for regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.6$, you traverse the tree starting at the root, and you eventually reach the leaf node that predicts `value=0.1106`. This prediction is simply the average target value of the 110 training instances associated to this leaf node. This prediction results in a Mean Squared Error (MSE) equal to 0.0151 over these 110 instances.

This model's predictions are represented on the left of [Figure 6-5](#). If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

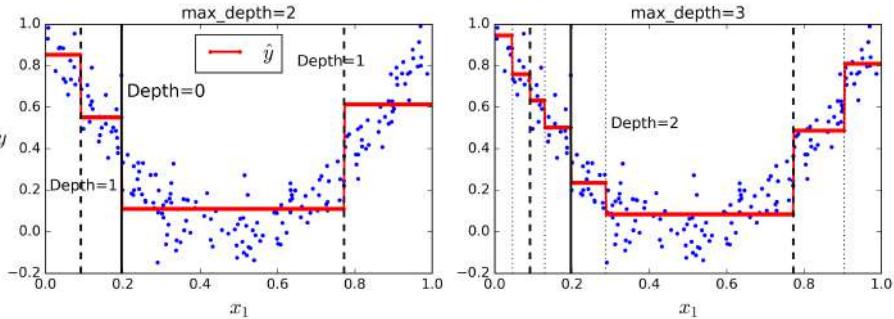


Figure 6-5. Predictions of two Decision Tree regression models

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. [Equation 6-4](#) shows the cost function that the algorithm tries to minimize.

[Equation 6-4. CART cost function for regression](#)

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \quad \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

Just like for classification tasks, Decision Trees are prone to overfitting when dealing with regression tasks. Without any regularization (ie. using the default hyperparameters), you get the predictions on the left of [Figure 6-6](#). It is obviously overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right of [Figure 6-6](#).

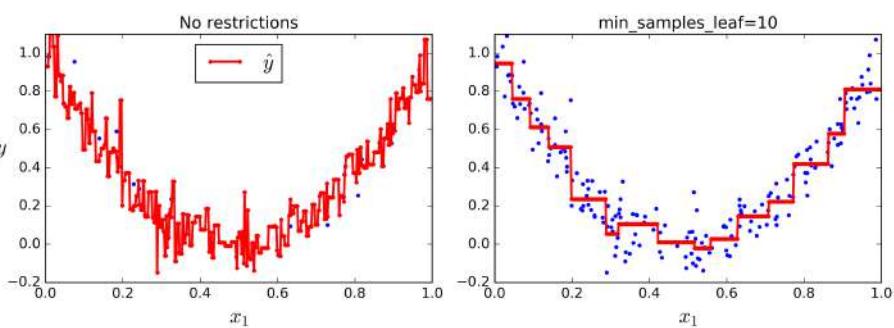


Figure 6-6. Regularizing a Decision Tree regressor

Instability

Hopefully by now you are convinced that Decision Trees have a lot going for them: they are simple to understand and interpret, easy to use, versatile and powerful. However they do have a few limitations. First, as you may have noticed, Decision Trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to training set rotation. For example, [Figure 6-7](#) shows a simple linearly separable dataset: on the left, a Decision Tree can split it easily, while on the right, after simply rotating the dataset by 45°, the decision boundary looks unnecessarily convoluted: although both Decision Trees fit the training set perfectly, it is very likely that the model on the right will not generalize well. One way to limit this problem is to use PCA (see [Chapter 8](#)) which often results in a better orientation of the training data.

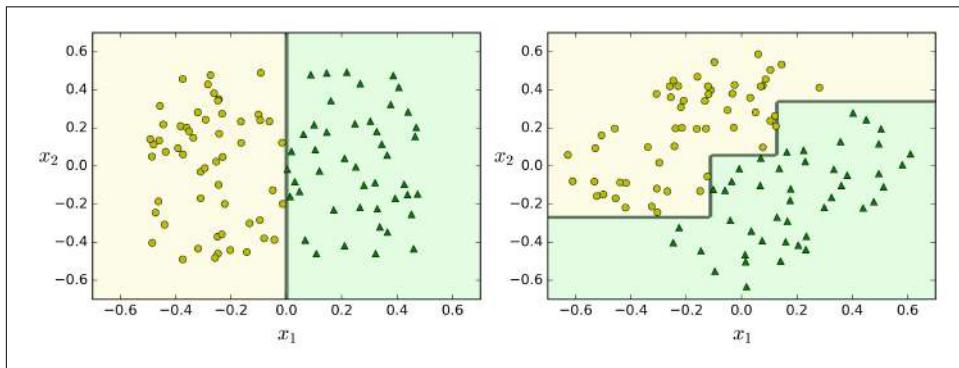


Figure 6-7. Sensitivity to training set rotation

More generally, the main issue with Decision Trees is that they are very sensitive to small variations in the training data. For example, if you just remove the widest Iris-Versicolour from the Iris training set (the one with petals 4.8 cm long and 1.8 cm wide) and train a new Decision Tree, you may get the model represented on [Figure 6-8](#). As you can see, it looks very different from the previous Decision Tree ([Figure 6-2](#)). Actually, since the training algorithm used by Scikit-Learn is stochastic⁶ you may get very different models even on the same training data (unless you set the `random_state` hyperparameter).

⁶ It randomly selects the set of features to evaluate at each node.

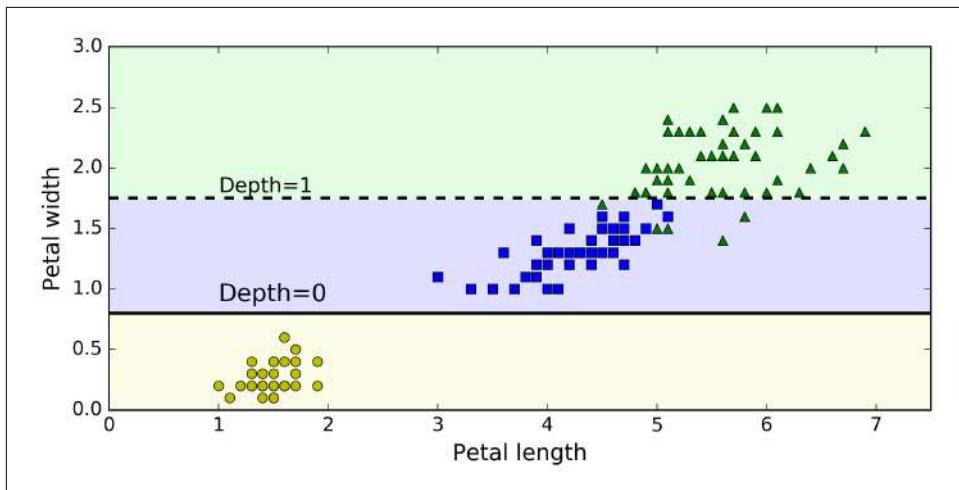


Figure 6-8. Sensitivity to training set details

Random Forests can limit this instability by averaging predictions over many trees, as we will see in the next chapter.

Exercises

1. What is the approximate depth of a Decision Tree trained (without restrictions) on a training set with one million instances?
2. Is a node's Gini impurity generally lower or greater than its parent's? Is it *generally* lower/greater, or *always* lower/greater?
3. If a Decision Tree is overfitting the training set, is it a good idea to try decreasing `max_depth`?
4. If a Decision Tree is underfitting the training set, is it a good idea to try scaling the input features?
5. If it takes 1 hour to train a Decision Tree on a training set containing one million instances, roughly how much time will it take to train another Decision Tree on a training set containing 10 million instances?
6. If your training set contains 100,000 instances, will setting `presort=True` speed up training?
7. Train and fine-tune a Decision Tree for the moons dataset.
 - a. Generate a moons dataset using `make_moons(n_samples=10000, noise=0.4)`.
 - b. Split it into a training set and a test set using `train_test_split()`.

- c. Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.
 - d. Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85 to 87% accuracy.
8. Grow a forest.
- a. Continuing the previous exercise, generate 1,000 subsets of the training set, each containing 100 instances selected randomly. Hint: you can use Scikit-Learn's `ShuffleSplit` class for this.
 - b. Train one Decision Tree on each subset, using the best hyperparameter values found above. Evaluate these 1,000 Decision Trees on the test set: since they were trained on smaller sets, these Decision Trees will likely perform worse than the first Decision Tree, achieving only about 80% accuracy.
 - c. Now comes the magic. For each test set instance, generate the predictions of the 1,000 Decision Trees, and keep only the most frequent prediction (you can use SciPy's `mode()` function for this). This gives you *majority-vote predictions* over the test set.
 - d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a Random Forest classifier!

Solutions to these exercises are available in [Appendix A](#).

Ensemble Learning and Random Forests

Suppose you ask a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*, thus this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

For example, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predictions of all individual trees, then predict the class that gets the most votes (see the last exercise in [Chapter 6](#)). Such an ensemble of Decision Trees is called a *Random Forest*: despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

Moreover, as we discussed in [Chapter 2](#), Ensemble methods are often used near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solution in Machine Learning competitions often involve several Ensemble methods (most famously in Netflix Prize competition).

In this chapter we will discuss the most popular Ensemble methods, including *Bagging*, *Boosting*, *Stacking* and a few others. We will also explore Random Forests, which are ensembles of Decision Trees.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy: you may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see [Figure 7-1](#)).

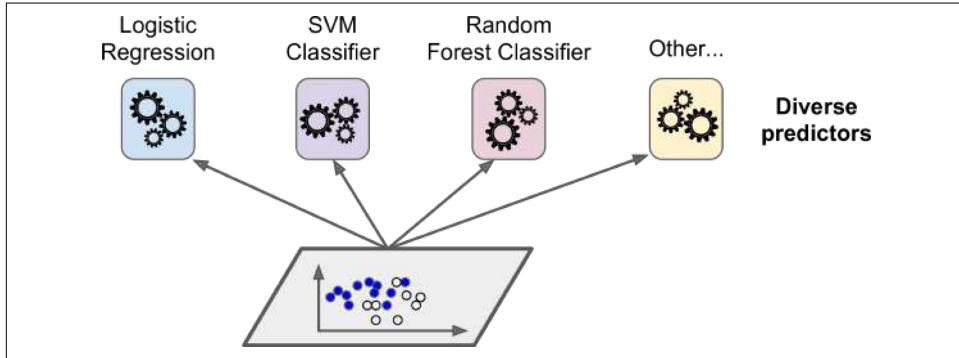


Figure 7-1. Training diverse classifiers

A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting classifier* (see [Figure 7-2](#)).

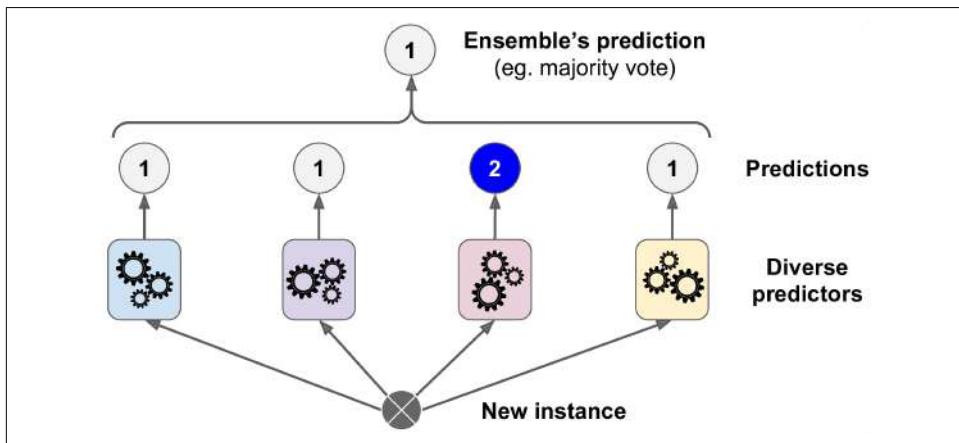


Figure 7-2. Hard voting classifier predictions

Somewhat surprisingly, this Voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners and they are sufficiently diverse.

How is this possible? The following analogy can help shed some light on this mystery: suppose you have a slightly biased coin which has a 51% chance of coming up heads, and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (eg. with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). [Figure 7-3](#) shows 10 series of biased coin tosses: you can see that as the number of tosses increases the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

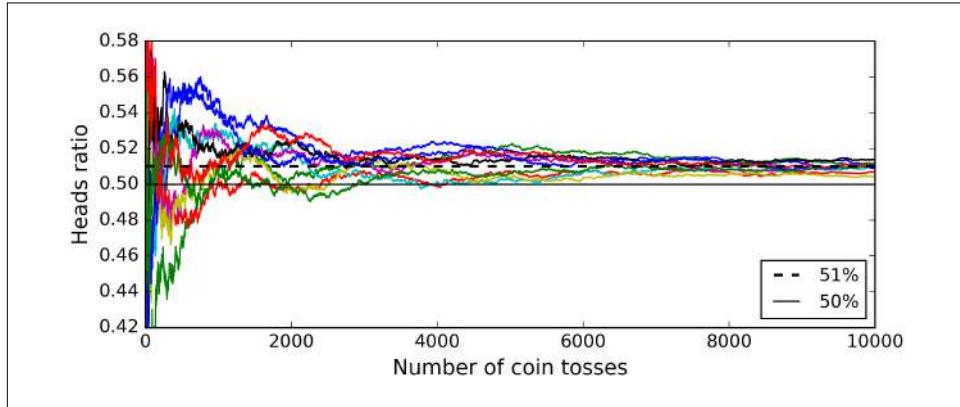


Figure 7-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, which is clearly not the case since they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.



Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms: this increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

The following code creates and trains a Voting classifier in Scikit-Learn, composed of 3 diverse classifiers (the training set is the moons dataset, introduced in [Chapter 5](#)):

```

from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard'
)
voting_clf.fit(X_train, y_train)

```

Let's look at each classifier's accuracy on the test set:

```

>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896

```

There you have it! The Voting classifier slightly outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (ie. they have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is replace `voting="hard"` by `voting="soft"` and ensure that all classifiers can estimate class probabilities. This is not the case of the `SVC` class by default, so you need to set its `probability` hyperparameter to `True` (this will make the `SVC` use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). If you modify the code above to use soft voting, you will find that the Voting classifier achieves over 91% accuracy!

Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as discussed above. Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set. When

sampling is performed *with* replacement, this method is called *Bagging*¹ (short for *Bootstrap Aggregating*²). When sampling is performed *without* replacement, it is called *Pasting*³.

In other words, both Bagging and Pasting allow training instances to be sampled several times across multiple predictors, but only Bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented on [Figure 7-4](#).

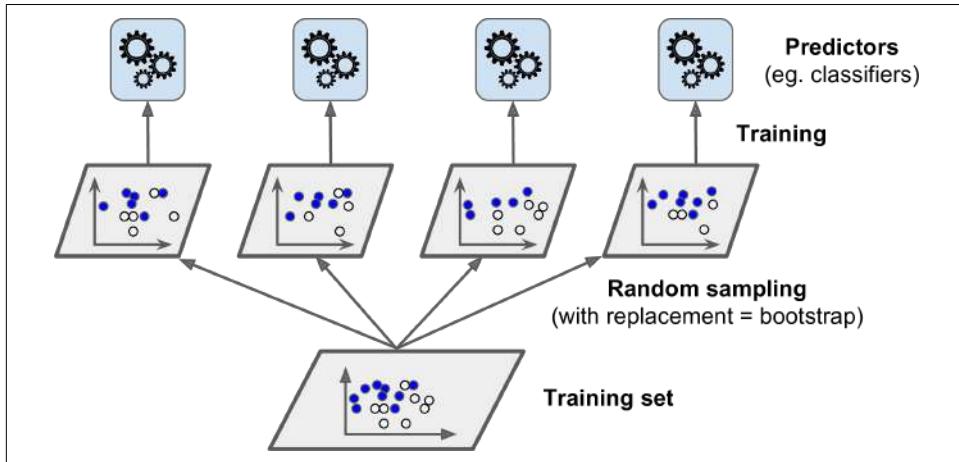


Figure 7-4. Pasting/Bagging training set sampling and training

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* (ie. the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance⁴: generally the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

As you can see in [Figure 7-4](#), predictors can all be trained in parallel, using different CPU cores, or even different servers. Similarly, predictions can be made in parallel.

¹ “Bagging Predictors”, L. Breiman (1996): <http://goo.gl/o42tml>

² In statistics, resampling with replacement is called *bootstrapping*.

³ “Pasting small votes for classification in large databases and on-line”, L. Breiman (1999): <http://goo.gl/BXm0pm>

⁴ Bias and variance were introduced in [Chapter 4](#).

This is one of the reasons why Bagging and Pasting are such popular methods: they scale very well.

Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both Bagging and Pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers⁵, each trained on 100 training instances randomly sampled from the training set with replacement (this is an example of Bagging, but if you want to use Pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (-1 tells Scikit-Learn to use all available cores).

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1
)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```



The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (ie. if it has a `predict_proba()` method), which is the case of Decision Trees classifiers.

Figure 7-5 compares the decision boundary of a single Decision Tree with the decision boundary of a Bagging ensemble of 500 trees (from the code above), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: it has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

⁵ `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of instances to sample is equal to the size of the training set times `max_samples`.

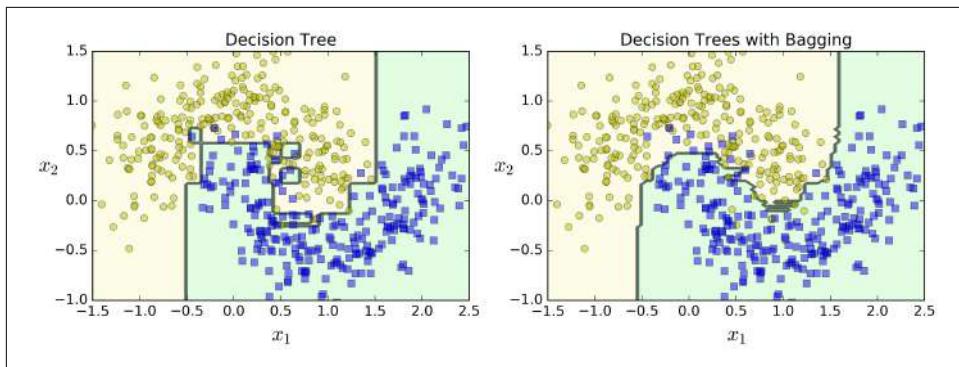


Figure 7-5. A single Decision Tree vs a Bagging ensemble of 500 trees

Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so Bagging ends up with a slightly higher bias than Pasting, but this also means that predictors end up being less correlated so the ensemble's variance is reduced. Overall, Bagging often results in better models which explains why it is generally preferred. However, if you have spare time and CPU power you can use cross-validation to evaluate both Bagging and Pasting and select the one that works best.

Out-of-bag evaluation

With Bagging, some instances may be sampled several times for any given predictor while others may not be sampled at all. By default a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor⁶. The remaining 37% training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation. The ensemble itself can be evaluated by averaging out the oob evaluations of each predictor.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
```

⁶ As m grows, this ratio approaches $1 - \exp(-1) \approx 63.212\%$.

```

>>>      )
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.9306666666666664

```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 93.1% accuracy on the test set. Let's verify this:

```

>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9360000000000005

```

We get 93.6% accuracy on the test set, close enough!

The oob decision function for each training instance is also available through the `oob_decision_function_` variable. In this case (since the base estimator has a `predict_proba()` method) the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the 2nd training instance has a 60.6% probability of belonging to the positive class (and 39.4% of belonging to the negative class).

```

>>> bag_clf.oob_decision_function_
array([[ 0.        ,  1.        ],
       [ 0.60588235,  0.39411765],
       [ 1.        ,  0.        ],
       ...
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.48958333,  0.51041667]])

```

Random Patches and Random Subspaces

The `BaggingClassifier` class also supports sampling the features as well. This is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus each predictor will be trained on a random subset of the input features.

This is particularly useful when dealing with high-dimensional inputs (such as images). Sampling both training instances and features is called the *Random Patches* Ensemble method⁷. Keeping all training instances (ie. `bootstrap=False` and `max_samples=1.0`) but sampling features (ie. `bootstrap_features=True` and/or `max_features` smaller than 1.0) is called the *Random Subspaces* method⁸.

⁷ “Ensembles on Random Patches”, G. Louppe and P. Geurts (2012): <http://goo.gl/B2EcM2>

⁸ “The random subspace method for constructing decision forests”, Tin Kam Ho (1998): <http://goo.gl/NPi5vH>

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

As we discussed above, a Random Forest⁹ is an ensemble of Decision Trees, generally trained using the Bagging method (or sometimes Pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees¹⁰ (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a Random Forest classifier with 500 trees (each limited to maximum 16 nodes), using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself¹¹.

The Random Forest algorithm introduces extra randomness when growing trees: instead of searching for the very best feature when splitting a node (see [Chapter 6](#)), it searches for the best feature among a random subset of features. This results in a greater tree diversity, which (once again) trades a higher bias for a lower variance, generally yielding an overall better model. The following `BaggingClassifier` is roughly equivalent to the `RandomForestClassifier` above:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1
)
```

⁹ “Random Decision Forests”, Tin Kam Ho (1995): <http://goo.gl/zVOGQ1>

¹⁰ The `BaggingClassifier` class remains useful if you want a bag of something else than Decision Trees.

¹¹ There are a few notable exceptions: `splitter` is absent (forced to "random"), `presort` is absent (forced to `False`), `max_samples` is absent (forced to 1.0), and `base_estimator` is absent (forced to `DecisionTreeClassifier` with the provided hyperparameters).

Extra-Trees

When growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (as discussed above). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do).

A forest of such extremely random trees is simply called an *Extremely Randomized Trees* ensemble¹² (or *Extra-Trees* for short). Once again, this trades more bias for a lower variance. It also makes Extra-Trees much faster to train than regular Random Forests since finding the best possible threshold for each feature at every node is one of the most time consuming tasks when growing a tree.

You can create an Extra-Trees classifier using Scikit-Learn's `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class.



It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`. Generally the only way to know is to try both and compare them using cross-validation (and tuning the hyperparameters using grid search).

Feature importance

Lastly, if you look at a single Decision Tree, important features are likely to appear closer to the root of the tree, while unimportant features will often appear closer to the leaves (or not at all). It is therefore possible to get an estimate of a feature's importance by computing the average depth at which it appears across all trees in the forest. Scikit-Learn computes this automatically for every feature after training. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the Iris dataset (introduced in [Chapter 4](#)) and outputs each feature's importance. It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2% respectively).

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
sepal length (cm) 0.112492250999
```

¹² "Extremely Randomized Trees", P. Geurts, D. Ernst, L. Wehenkel (2005): <http://goo.gl/RHGEA4>

```
sepal width (cm) 0.0231192882825  
petal length (cm) 0.441030464364  
petal width (cm) 0.423357996355
```

Similarly, if you train a Random Forest classifier on the MNIST dataset (introduced in [Chapter 3](#)), and plot each pixel's importance, you get the image represented in [Figure 7-6](#).

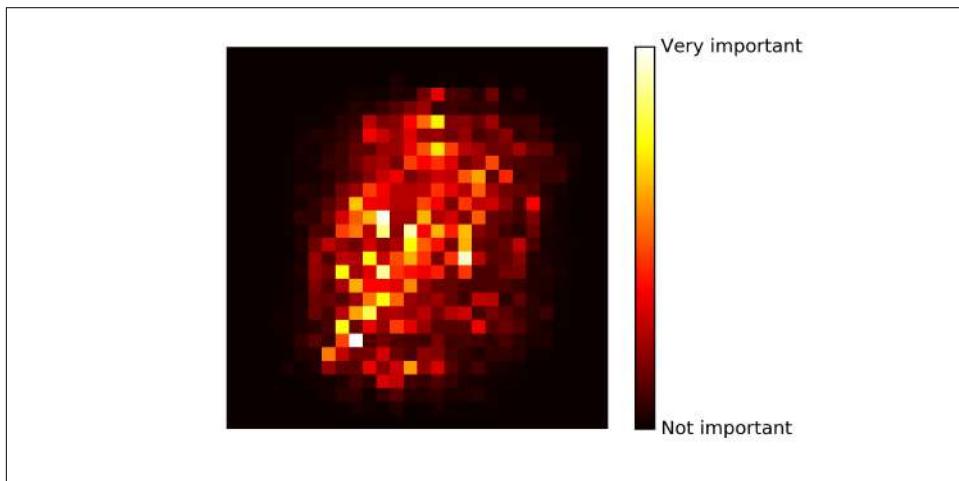


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

Boosting

Boosting (originally called *Hypothesis Boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most Boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many Boosting methods available, but by far the most popular are *AdaBoost*¹³ (short for *Adaptive Boosting*) and *Gradient Boosting*. Let's start with AdaBoost.

AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predic-

¹³ Introduced in 1997 by Y. Freund and R. Schapire: <http://goo.gl/OIduRW>

tors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, to build an AdaBoost classifier, a first base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, weights are updated, and so on (see [Figure 7-7](#)).

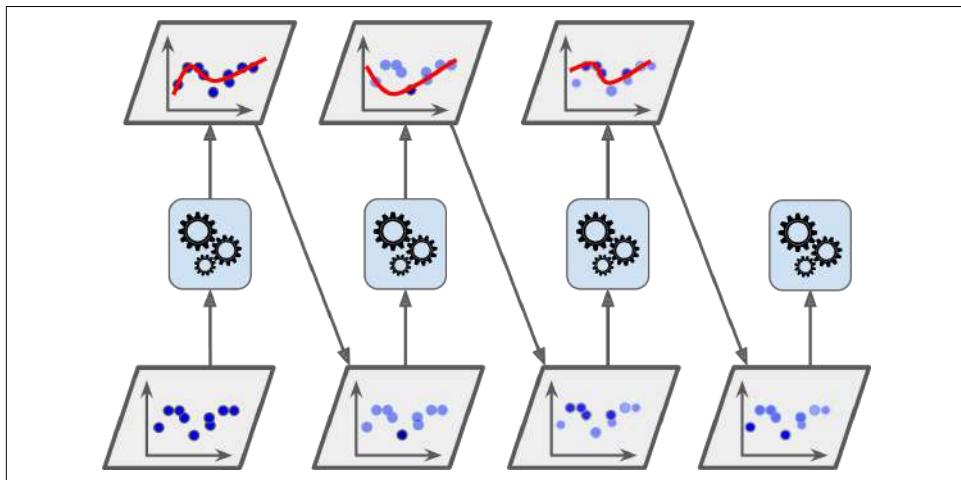


Figure 7-7. AdaBoost sequential training with instance weight updates

[Figure 7-8](#) shows the decision boundaries of 5 consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel¹⁴). The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on. The plot on the right represents the same sequence of predictors except that the learning rate is halved (ie. the misclassified instance weights are boosted half as much at every iteration). As you can see this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

¹⁴ This is just for illustrative purposes: SVMs are generally not good base predictors for AdaBoost, because they are slow and tend to be unstable with AdaBoost.

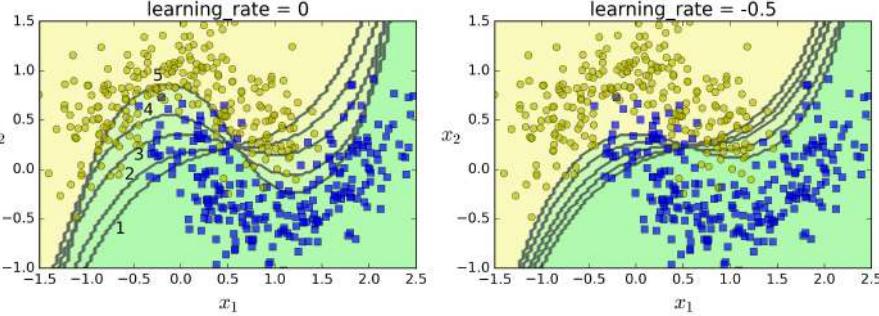


Figure 7-8. Decision boundaries of consecutive predictors

Once all predictors are trained, the ensemble makes predictions very much like Bagging or Pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.



There is one important drawback to this sequential learning technique: it cannot be parallelized (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as Bagging or Pasting.

Let's take a closer look at the AdaBoost algorithm. Each instance weight $w^{(i)}$ is initially set to $\frac{1}{m}$. A first predictor is trained and its weighted error rate r_1 is computed on the training set.

Equation 7-1. Weighted error rate of the j^{th} predictor

$$r_j = \frac{\sum_{i=1}^m w^{(i)}_{\hat{y}_j^{(i)} \neq y^{(i)}}}{\sum_{i=1}^m w^{(i)}} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance.}$$

The predictor's weight α_j is then computed using [Equation 7-2](#), where η is the learning rate hyperparameter (defaults to 1)¹⁵. The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to

¹⁵ The original AdaBoost algorithm does not use a learning rate hyperparameter.

zero. However, if it is most often wrong (ie. less accurate than random guessing), then its weight will be negative.

Equation 7-2. Predictor weight

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next the instance weights are updated using [Equation 7-3](#): the misclassified instances are boosted.

Equation 7-3. Weight update rule

for $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized (ie. divided by $\sum_{i=1}^m w^{(i)}$).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated (the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on). The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j . The predicted class is the one that receives the majority of weighted votes (see [Equation 7-4](#)).

Equation 7-4. AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{j=1}^N \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

$$\hat{y}_j(\mathbf{x}) = k$$

Scikit-Learn actually uses a multi-class version of AdaBoost called *SAMME*¹⁶ (which stands for *Stagewise Additive Modeling using a Multi-class Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost. Moreover, if the predictors can estimate class probabilities (ie. if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called *SAMME.R* (the “R” stands

¹⁶ For more details, see “Multi-Class AdaBoost”, Zhu *et al.* (2006): <http://goo.gl/Eji2vR>

for “Real”) which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn’s `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A Decision Stump is a Decision Tree with `max_depth=1`, in other words a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class.

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5
)
ada_clf.fit(X_train, y_train)
```



If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

Gradient Boosting

Another very popular Boosting algorithm is *Gradient Boosting*¹⁷. Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let’s go through a simple regression example using Decision Trees as the base predictors (of course Gradient Boosting also works great with regression tasks). This is called *Gradient Tree Boosting*, or *Gradient Boosted Regression Trees (GBRT)*. First, let’s fit a `DecisionTreeRegressor` to the training set (for example, a noisy quadratic training set):

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Now train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

¹⁷ First introduced in “Arcing the Edge”, L. Breiman (1997): <http://goo.gl/Ezw4jL>

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Then a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing 3 trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Figure 7-9 represents the predictions of these 3 trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just 1 tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

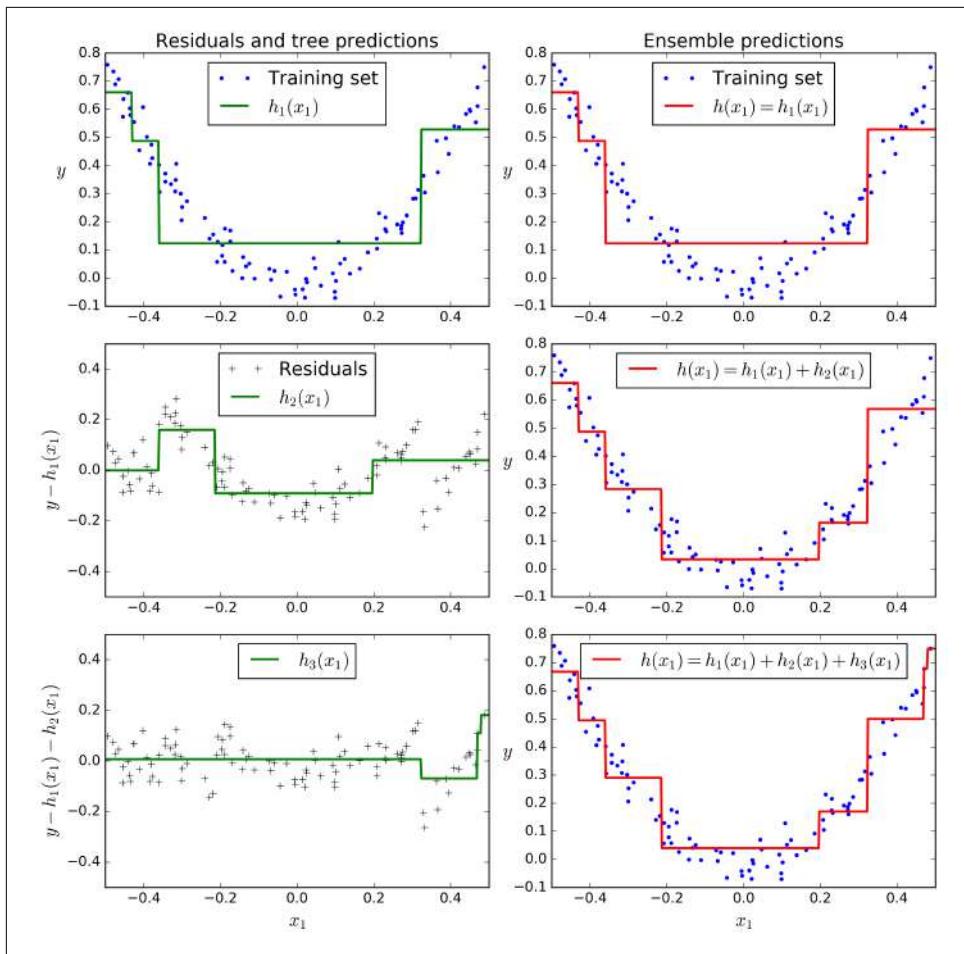


Figure 7-9. Gradient Boosting

A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class. Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of Decision Trees (eg. `max_depth`, `min_samples_leaf`, and so on), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the one above:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as 0.1, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 7-10 shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on the right has too many trees and it overfits the training set.

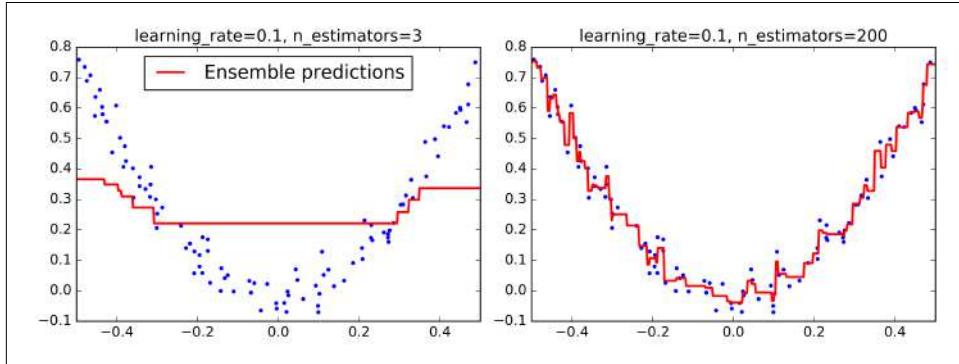


Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

In order to find the optimal number of trees, you can use early stopping (see Chapter 4). A simple way to implement this is to use the `staged_predict()` method: it returns an iterator over the predictions made by the ensemble at each stage of training (with 1 tree, 2 trees, etc.). The following code trains a GBRT ensemble with 120 trees, then measures the validation error at each stage of training to find the optimal number of trees, and finally trains another GBRT ensemble using the optimal number of trees.

```
from sklearn.cross_validation import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
best_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=best_n_estimators)
gbrt_best.fit(X_train, y_train)
```

The validation errors are represented on the left of Figure 7-11, and the best model's predictions are represented on the right.

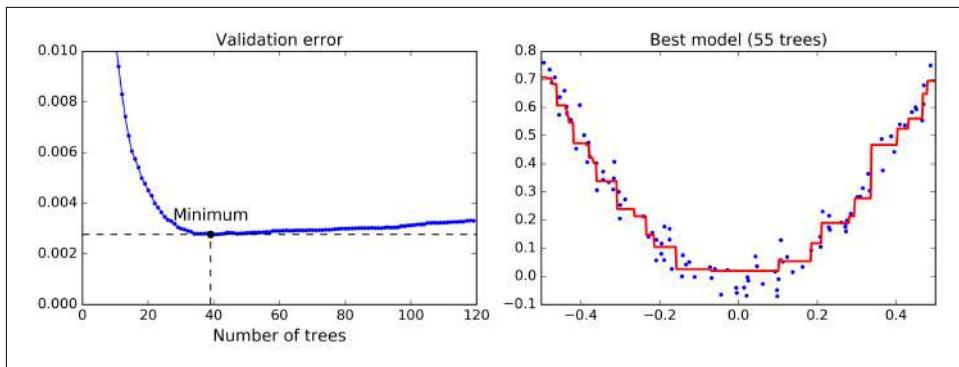


Figure 7-11. Tuning the number of trees using early stopping

It is also possible to implement early stopping by actually stopping training early (instead of training a large number of trees first then looking back to find the optimal number). This can be done by setting `warm_start=True`, which makes Scikit-Learn keep existing trees when the `fit()` method is called, allowing incremental training. The following code stops training when the validation error does not improve for 5 iterations in a row:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
    if error_going_up == 5:
        break # early stopping
```

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25` then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this trades a higher bias for a lower variance. It also speeds up training considerably. This technique is called *Stochastic Gradient Boosting*.



It is possible to use Gradient Boosting with other cost functions. This is controlled by the `loss` hyperparameter (see Scikit-Learn's documentation for more details).

Stacking

The last Ensemble method we will discuss in this chapter is called *Stacking* (short for *Stacked Generalization*)¹⁸. It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? Figure 7-12 shows such an ensemble performing a regression task on a new instance: each of the bottom 3 predictors predicts a different value (3.1, 2.7 and 2.9), then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).

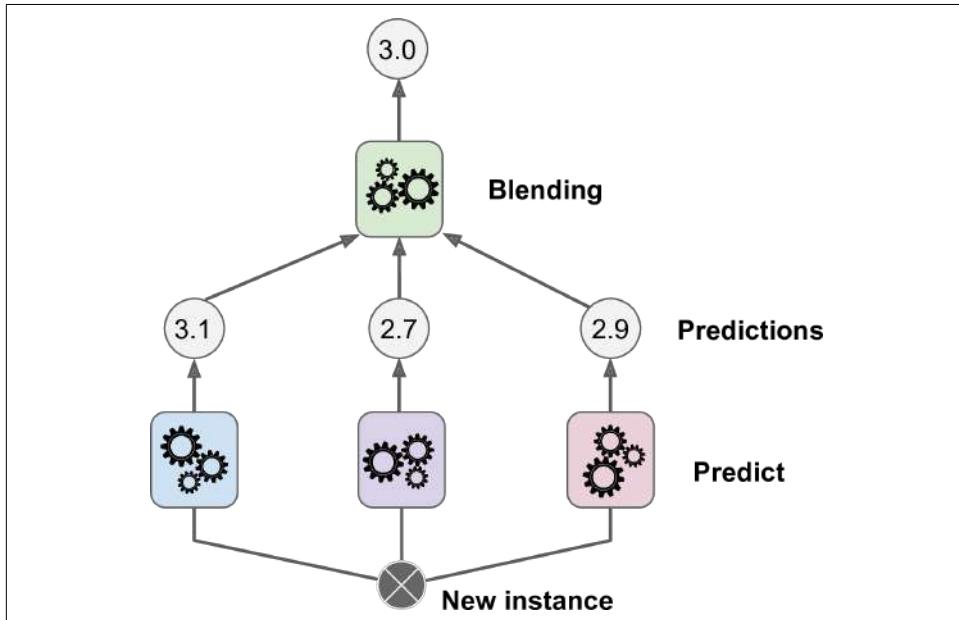


Figure 7-12. Aggregating predictions using a blending predictor

¹⁸ “Stacked Generalization”, D. Wolpert (1992): <http://goo.gl/9I2NBw>

To train the blender, a common approach is to use a hold-out set¹⁹. Let's see how it works. First, the training set is split in two subsets. The first subset is used to train the predictors in the first layer (see [Figure 7-13](#)).

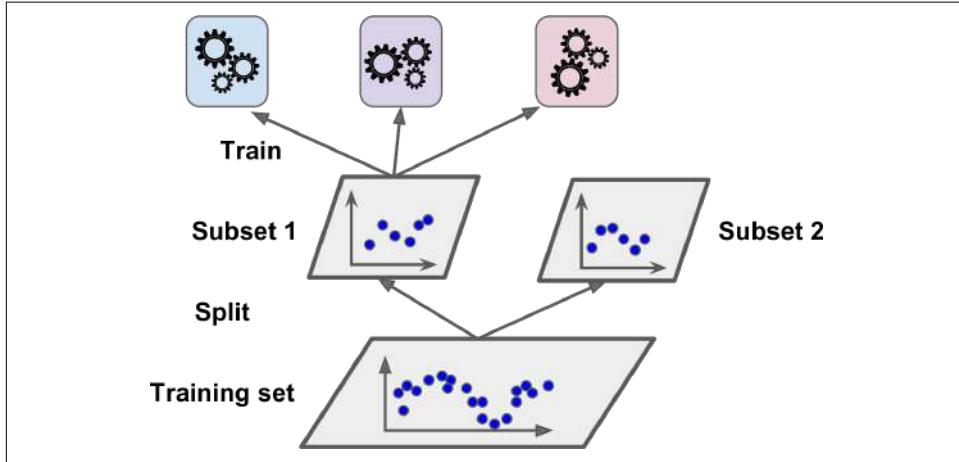


Figure 7-13. Training the first layer

Next, the first layer predictors are used to make predictions on the second (held-out) set (see [Figure 7-14](#)). This ensures that the predictions are “clean”, since the predictors never saw these instances during training. Now for each instance in the hold-out set there are 3 predicted values. We can create a new training set using these predicted values as input features (which makes this new training set 3-dimensional), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value given the first layer’s predictions.

¹⁹ Alternatively, it is possible to use out-of-fold predictions. In some contexts this is called *Stacking* while using a hold-out set is called *Blending*. However, for many people these terms are synonymous.

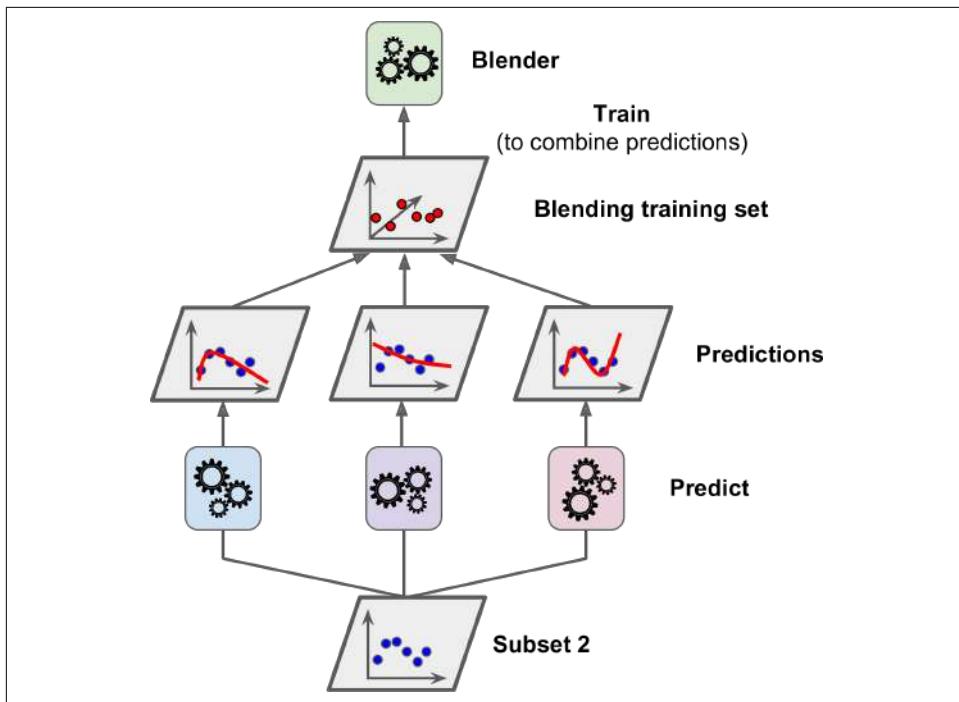


Figure 7-14. Training the blender

It is actually possible to train several different blenders this way (eg. one using Linear Regression, another using Random Forest regression, and so on): we get a whole layer of blenders. The trick is to split the training set into 3 subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer). Once this is done, a prediction can be made for a new instance by going through each layer sequentially, as shown in Figure 7-15.

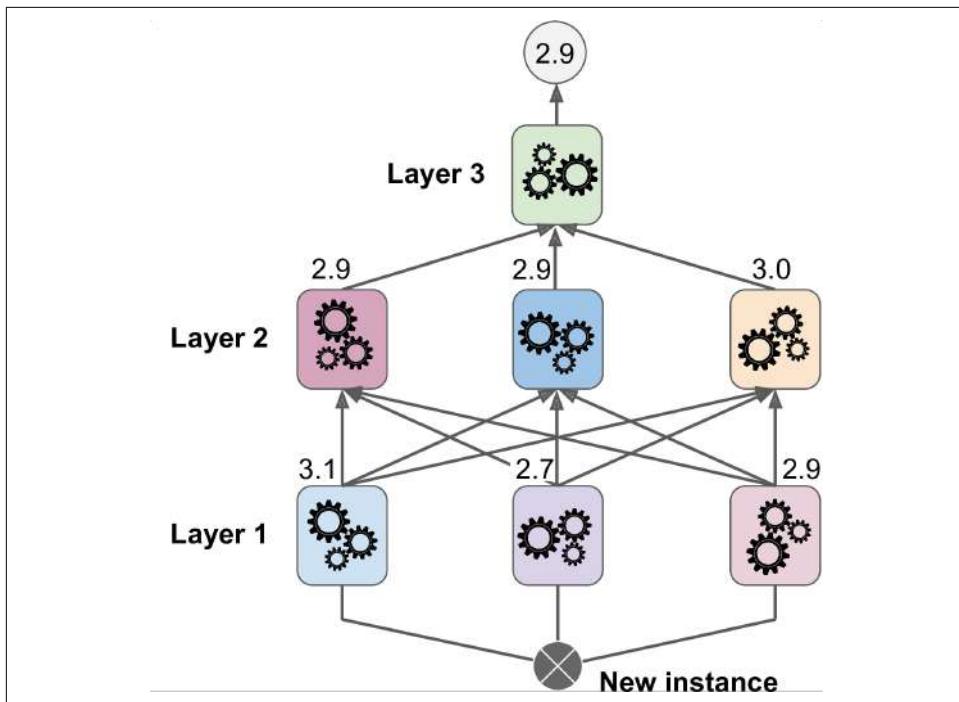


Figure 7-15. Predictions in a multi-layer Stacking ensemble

Unfortunately, Scikit-Learn does not support Stacking directly, but it is not too hard to roll out your own implementation (see the exercises below). Alternatively, you can use an open source implementation such as `brew` (available at <https://github.com/viisar/brew>).

Exercises

1. If you have trained 5 different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft Voting Classifiers?
3. Is it possible to speed up training of a Bagging ensemble by distributing it across multiple servers? What about Pasting ensembles, Boosting ensembles, Random Forests or Stacking ensembles?
4. What is the benefit of Out-of-Bag evaluation?
5. What makes Extra-Trees more random than regular Random Forests? How can this extra randomness help? Are Extra-Trees slower or faster than regular Random Forests?

6. If your AdaBoost ensemble underfits the training data, what hyperparameters should you tweak and how?
7. If your Gradient Boosting ensemble overfits the training set, should you increase or decrease the learning rate?
8. Load the MNIST data (introduced in [Chapter 3](#)), and split it into a training set, a validation set and a test set (eg. use the first 40,000 instances for training, the next 10,000 for validation and the last 10,000 for testing). Then train various classifiers, such as a Random Forest classifier, an Extra-Trees classifier, and an SVM, then try to combine them into an ensemble that outperforms them all on the validation set, using a soft or hard Voting classifier. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Congratulations, you have just trained a blender, and together with the classifiers they form a Stacking ensemble! Now let's evaluate the ensemble on the test set: for each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the Voting Classifier you trained earlier?

Solutions to these exercises are available in [Appendix A](#).

Dimensionality Reduction

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only does this make training extremely slow, it can also make it much harder to find a good solution, as we will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in [Chapter 3](#)): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. [Figure 7-6](#) confirms that these pixels are utterly unimportant for the classification task. Moreover, two neighboring pixels are often highly correlated: if you merge them into a single pixel (eg. by taking the mean of the two pixel intensities), you will not lose much information.



Reducing dimensionality does lose some information (just like compressing an image to JPEG can degrade its quality) so even though it will speed up training, it may also make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. So you should first try to train your system with the original data before considering using reducing dimensionality if training is too slow. In some cases however, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance (but in general it won't, it will just speed up training).

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or *DataViz*): by reducing the number of dimensions down to 2

(or 3), it is possible to plot a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters.

In this chapter we will discuss the curse of dimensionality and get an intuition of what goes on in high-dimensional space, then we will present the two main approaches to dimensionality reduction (projection and manifold learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, Kernel PCA and LLE.

The curse of dimensionality

We are so used to living in 3 dimensions¹ that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our mind (see [Figure 8-1²](#)), let alone a 200-dimensional ellipsoid bended in a 1,000-dimensional space.

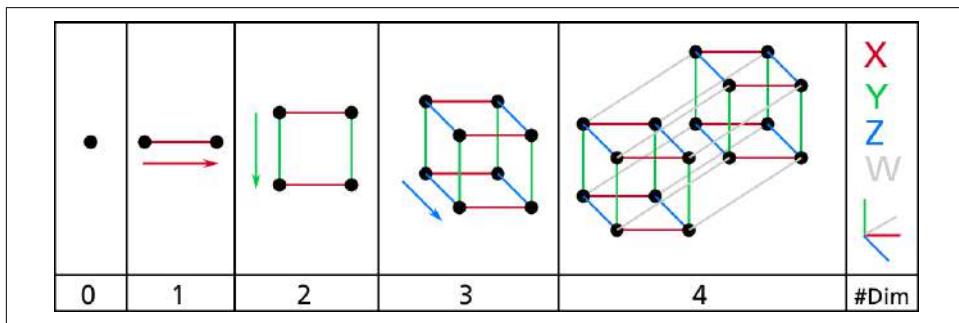


Figure 8-1. Point, segment, square, cube and tesseract (0D to 4D hypercubes)

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square ($a 1 \times 1$ square), it will have only about 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be “extreme” along any dimension). But in a 10,000-dimensional unit hypercube ($a 1 \times 1 \times \dots \times 1$ cube, with ten thousand 1s), this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border³. This also implies that most feature vectors in a high-dimensional dataset are likely to be almost orthogonal.

¹ Well, 4 dimensions if you count time, and a few more if you are a string theorist.

² Watch a rotating tesseract projected into 3D space at <http://goo.gl/OM7ktJ>. Image by Wikipedia user NerdBoy1392 (Creative Commons BY-SA 3.0). Reproduced from <https://en.wikipedia.org/wiki/Tesseract>.

³ Fun fact: anyone you know is probably an extremist in at least one dimension (eg. how much sugar they put in their coffee), if you consider enough dimensions.

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66. But what about two points picked randomly in a one-million-dimensional hypercube? Well the average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1,000,000/6}$)! This is quite counter-intuitive: how can two points be so far apart when they both lie within the same unit hypercube? This fact implies that high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. Of course this also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations.

In theory one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately in practice the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features (much less than in the MNIST problem) you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

Main approaches for dimensionality reduction

Before we dive into specific dimensionality reduction algorithms, let's take a look at the two main approaches: projection and manifold learning.

Projection

In most real world problems training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, others are highly correlated (as discussed above for MNIST). As a result all training instances actually lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract so let's look at an example. In [Figure 8-2](#) you can see a 3D dataset represented by the circles.

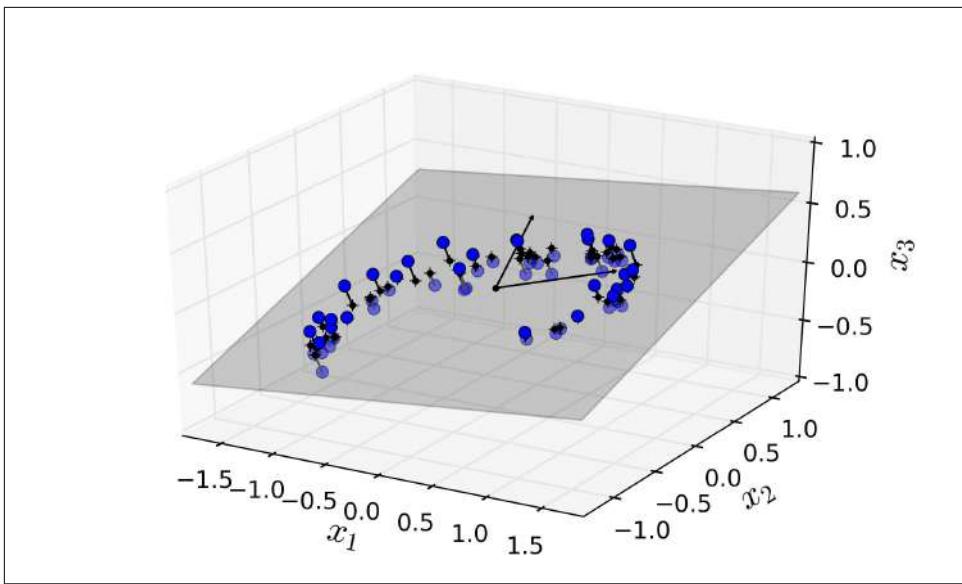


Figure 8-2. A 3D dataset lying close to a 2D subspace

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space. Now if we project every training instance perpendicularly onto this subspace (as represented by the short lines connecting the instances to the plane), we get a new 2D dataset shown in [Figure 8-3](#). Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2 (the coordinates of the projections on the plane).

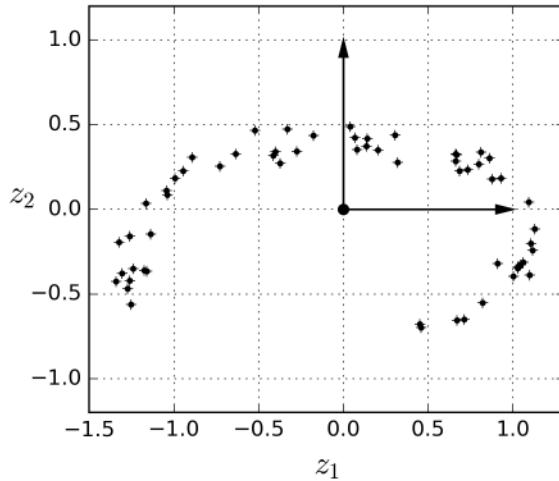


Figure 8-3. The new 2D dataset after projection

However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous *swiss roll* toy dataset represented in Figure 8-4.

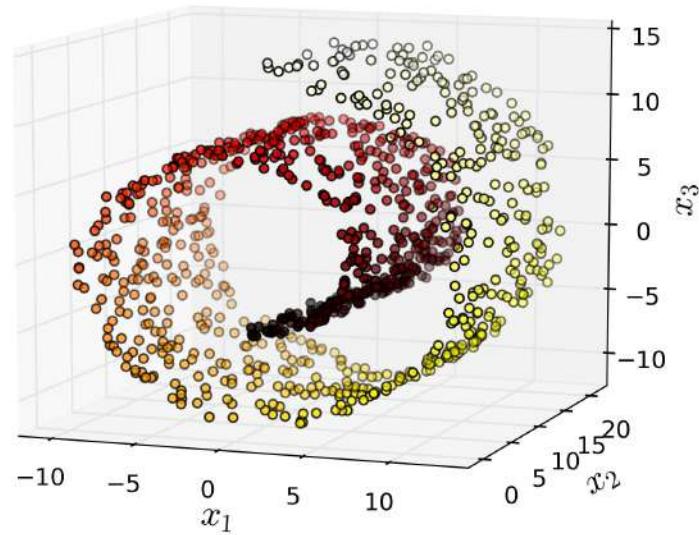


Figure 8-4. Swiss roll dataset

A simple projection onto a plane (eg. by dropping x_3) would squash different layers of the swiss roll together, as shown on the left of [Figure 8-5](#). However what you really want is to unroll the swiss roll to obtain the 2D dataset on the right of [Figure 8-5](#).

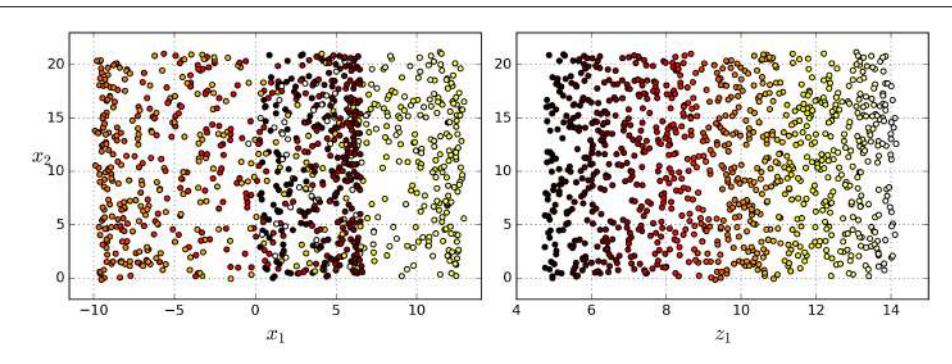


Figure 8-5. Squashing by projecting onto a plane (left) vs unrolling the swiss roll (right)

Manifold learning

The swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bended and twisted in a higher-dimensional space. More generally, a d -dimensional manifold is a part of an n -dimensional space (where $d < n$) that locally resembles a d -dimensional hyperplane. In the case of the swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the 3rd dimension.

Many dimensionality reduction algorithms work by modeling the *manifold* on which the training instances lie: this is called *manifold learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*: this assumption holds that most real world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, they are more or less centered, and so on. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you would have if you were allowed to generate any image you wanted. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (eg. classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of [Figure 8-6](#) the swiss roll is split into two classes: in the 3D space (on the left), the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right), the decision boundary is a simple straight line.

However, this assumption does not always hold. For example, in the bottom row of [Figure 8-6](#), the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex on the unrolled manifold (a collection of 4 independent line segments).

In short, if you reduce the dimensionality of your training set before training a model, it will definitely speed up training, but it may not always lead to a better or simpler solution: it all depends on the dataset.

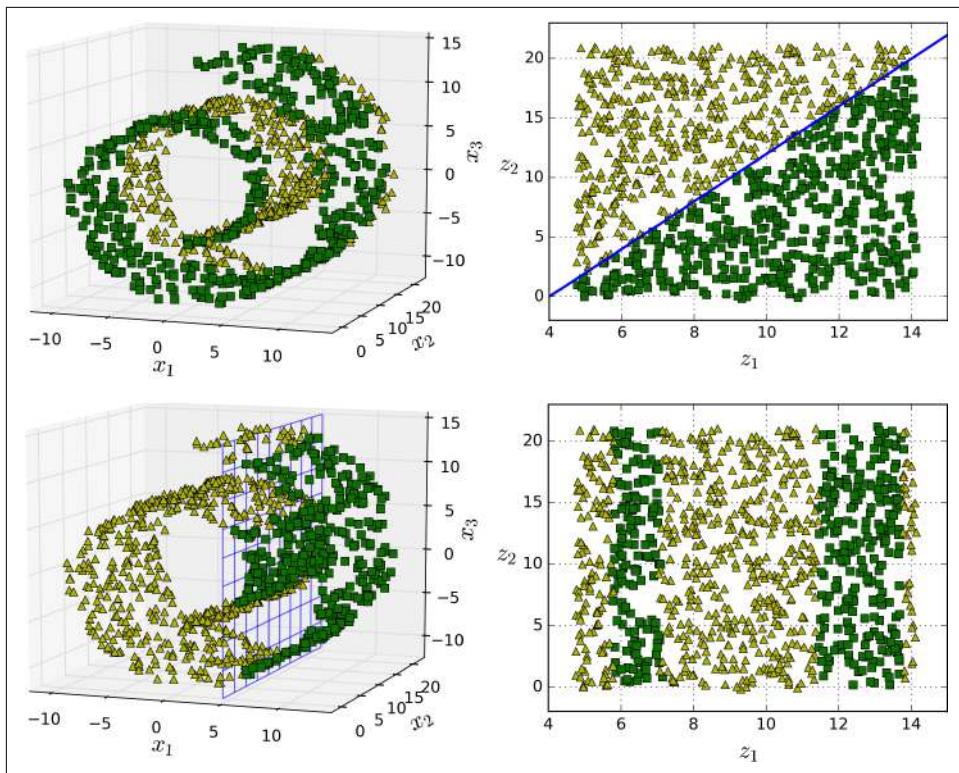


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

Hopefully you now have a good intuition of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms.

PCA

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, then it projects the data onto it.

Preserving the variance

Before you can project the training set onto a lower dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left of Figure 8-7, along with 3 different axes (ie. one-dimensional hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum var-

iance, while the projection onto the dotted line preserves very little variance, and the projection onto the dashed line preserves an intermediate amount of variance.

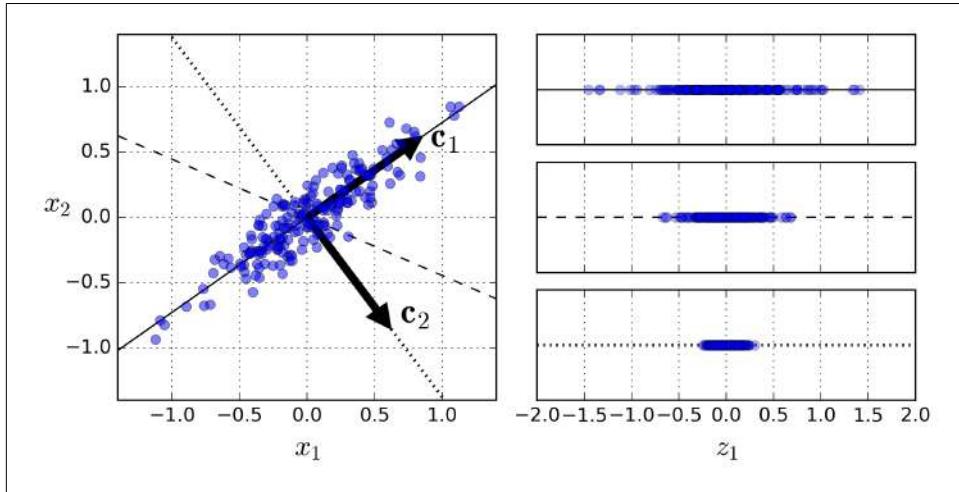


Figure 8-7. Selecting the subspace onto which to project

It seems reasonable to select the axis that preserves the maximum amount of variance as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind *PCA* (short for *Principal Component Analysis*)⁴.

Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In Figure 8-7, it is the solid line. It also finds a second axis, orthogonal to the first one, which accounts for the largest amount of remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on: as many axes as the number of dimensions in the dataset.

The unit vector that defines the i^{th} axis is called the i^{th} *Principal Component* (PC). In Figure 8-7, the 1st PC is c_1 and the 2nd PC is c_2 . In Figure 8-2 the first two PCs are represented by the orthogonal arrows in the plane, and the 3rd PC would be orthogonal to the plane (pointing up or down).

⁴ “On Lines and Planes of Closest Fit to Systems of Points in Space”, K. Pearson (1901): <http://goo.gl/gbNo1D>



The direction of the principal components is not stable: if you perturb the training set slightly and run PCA again, some of the new PCs may point in the opposing direction to the original PCs. However they will generally still lie on the same axes. In some cases, a pair of PCs may even rotate or swap, but the plane they define will generally remain the same.

So how can you find the Principal Components of a training set? Luckily, there is a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix \mathbf{X} into the dot product of three matrices $\mathbf{U} \cdot \Sigma \cdot \mathbf{V}^T$, where \mathbf{V}^T contains all the Principal Components that we are looking for, as shown in [Equation 8-1](#).

Equation 8-1. Principal Components matrix

$$\mathbf{V}^T = \begin{pmatrix} | & | & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & | \end{pmatrix}$$

The following python code uses NumPy's `svd()` function to obtain all the Principal Components of the training set, then it extracts the first two PCs.

```
X_centered = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
```



PCA assumes that the dataset is centered around the origin. As we will see, Scikit-Learn's PCA classes take care of centering the data for you. However, if you implement PCA yourself (as above), or if you use other libraries, don't forget to center the data first.

Projecting down to d dimensions

Once you have identified all the Principal Components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d Principal Components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example in [Figure 8-2](#) the 3D dataset is projected down to the 2D plane defined by the first two principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix \mathbf{X} by the matrix \mathbf{W}_d defined as the matrix contain-

ing the first d Principal Components (ie. the matrix composed of the first d columns of \mathbf{V}^T).

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X} \cdot \mathbf{W}_d$$

The following python code projects the training set onto the plane defined by the first 2 principal components:

```
W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
```

There you have it! You now know how to reduce the dimensionality of any dataset down to any number of dimensions, while preserving as much variance as possible.

Using Scikit-Learn

Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did above. The following code applies PCA to reduce the dimensionality of the dataset down to 2 dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

After fitting the PCA transformer to the dataset, you can access the Principal Components using the `components_` variable (note that it contains the PCs as horizontal vectors, so for example the 1st Principal Component is equal to `pca.components_.T[:, 0]`).

Explained variance ratio

Another very useful piece of information is the *explained variance ratio* of each Principal Component, available *via* the `explained_variance_ratio_` variable. It indicates the proportion of the dataset's variance that lies along the axis of each Principal Component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in Figure 8-2:

```
>> print(pca.explained_variance_ratio_)
array([ 0.84248607,  0.14631839])
```

This tells you that 84.2% of the dataset's variance lies along the first axis, and 14.6% lies along the second axis. This leaves less than 1.2% for the 3rd axis: it is reasonable to assume that it probably carries little information.

Choosing the right number of dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (eg. 95%). Unless of course you are reducing dimensionality for data visualization, in which case you will generally want to reduce the dimensionality down to 2 or 3.

The following code computes PCA without reducing dimensionality, then it computes the minimum number of dimensions required to preserve 95% of the training set's variance.

```
pca = PCA()  
pca.fit(X)  
d = np.argmax(np.cumsum(pca.explained_variance_ratio_) >= 0.95) + 1
```

You could then set `n_components=d` and run PCA again. However, there is a much better option: instead of specifying the number of Principal Components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve.

```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X)
```

PCA for compression

Obviously after dimensionality reduction, the training set takes up much less space. For example, try applying PCA to the MNIST dataset while preserving 95% of its variance: you should find that each instance will have just over 150 features, instead of the original 784 features. So while most of the variance is preserved, the dataset is now less than 20% of its original size! This is a reasonable compression ratio, and you can see how this can speed up a classification algorithm (such as an SVM classifier) tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. Of course this won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be quite close to the original data. The mean squared distance between the original data and the reconstructed data (compressed then decompressed) is called the *reconstruction error*. For example, the following code compresses the MNIST dataset down to 154 dimensions, then uses the `inverse_transform()` method to decompresses it back to 784 dimensions. [Figure 8-8](#) shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

```

pca = PCA(n_components = 154)
X_mnist_reduced = pca.fit_transform(X_mnist)
X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)

```

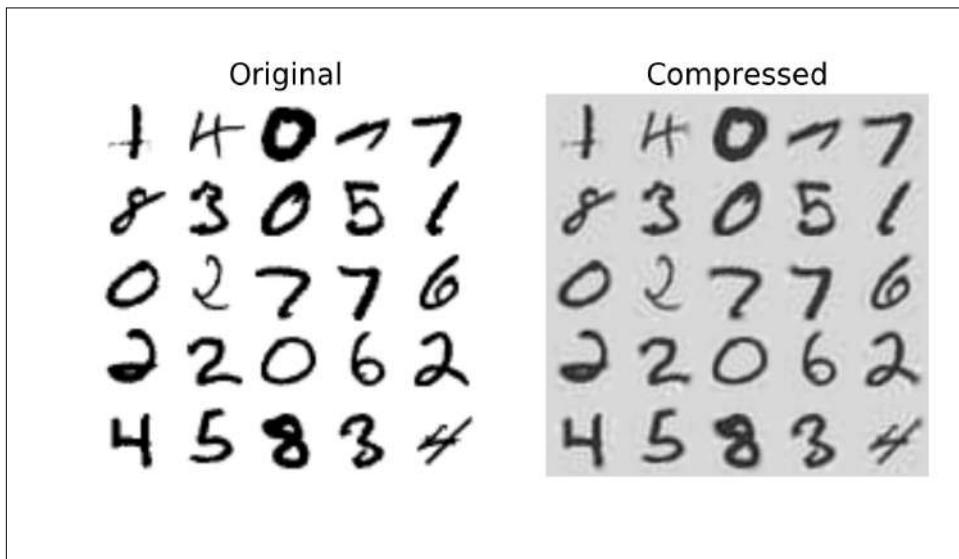


Figure 8-8. MNIST compression preserving 95% of the variance

The equation of the inverse transformation is shown in [Equation 8-3](#).

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d-\text{proj}} \cdot \mathbf{W}_d^T$$

Incremental PCA

One problem with the above implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, *Incremental PCA* (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time. This is useful for large training sets, and also to apply PCA online (ie. on the fly, as new instances arrive).

The following code splits the MNIST dataset into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class⁵ to

⁵ Scikit-Learn uses the algorithm described in “Incremental Learning for Robust Visual Tracking”, D. Ross *et al.* (2007): <http://goo.gl/FmdhUP>

reduce the dimensionality of the MNIST dataset down to 154 dimensions (just like above). Note that you must call the `partial_fit()` method with each mini-batch rather than the `fit()` method with the whole training set.

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_mnist, n_batches):
    inc_pca.partial_fit(X_batch)

X_mnist_reduced = inc_pca.transform(X_mnist)
```

Alternatively, you can use NumPy's `memmap` class which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory: the class loads only the data it needs in memory, when it needs it. Since the `IncrementalPCA` class only uses a small part of the array at any given time, the memory usage remains under control. This makes it possible to call the usual `fit()` method, as you can see in the following code:

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

RandomizedPCA

Scikit-Learn offers yet another class to perform PCA, called `RandomizedPCA`. This class relies on a stochastic algorithm that quickly finds an approximation of the first d Principal Components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n .

```
from sklearn.decomposition import RandomizedPCA

rnd_pca = RandomizedPCA(n_components=154)
X_reduced = rnd_pca.fit_transform(X_mnist)
```

Kernel PCA

In [Chapter 5](#) we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the *feature space*), enabling non-linear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex non-linear decision boundary in the *original space*.

It turns out that the same trick can be applied to PCA, making it possible to perform complex non-linear projections for dimensionality reduction. This is called *Kernel PCA* (kPCA)⁶. It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

For example, the following code uses Scikit-Learn's KernelPCA class to perform kPCA with an RBF kernel (see [Chapter 5](#) for more details about the RBF kernel and the other kernels):

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

[Figure 8-9](#) shows the swiss roll, reduced to 2 dimensions using a linear kernel (equivalent to simply using the PCA class), an RBF kernel and a sigmoid kernel.

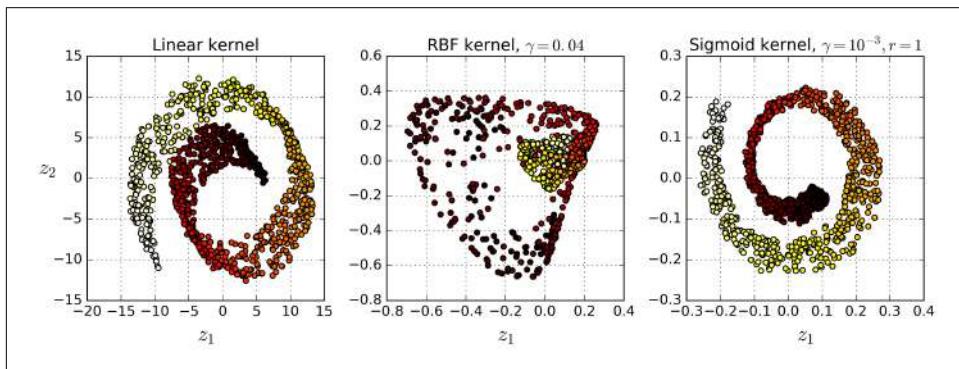


Figure 8-9. Swiss roll reduced to 2D using kPCA with various kernels

Selecting a kernel and tuning hyperparameters

As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values. However, dimensionality reduction is often a preparation step for a supervised learning task (eg. classification), so you can simply use grid search to select the kernel and hyperparameters that lead to the best performance on that task. For example, the following code creates a 2-step pipeline, first reducing dimensionality to 2 dimensions using kPCA, then applying Logistic Regression for classification. Then it uses GridSearchCV to find the best kernel and gamma value for kPCA in order to get the best classification accuracy at the end of the pipeline.

⁶ “Kernel Principal Component Analysis”, B. Schölkopf, A. Smola, K. Müller (1999): <http://goo.gl/5lQT5Q>

```

from sklearn.grid_search import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
    ("kpca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())
])

param_grid = [{ "kpca_gamma": np.linspace(0.03, 0.05, 10),
    "kpca_kernel": ["rbf", "sigmoid"]}]
])

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)

```

The best kernel and hyperparameters are then available through the `best_params_` variable:

```

>>> print(grid_search.best_params_)
{'kpca_gamma': 0.04333333333333335, 'kpca_kernel': 'rbf'}

```

Another approach, this time entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error. However, reconstruction is not as easy as with linear PCA. Here's why. [Figure 8-10](#) shows the original swiss roll 3D dataset (top left), and the resulting 2D dataset after kPCA is applied using an RBF kernel (top right). Thanks to the kernel trick, this is mathematically equivalent to mapping the training set to an infinite-dimensional feature space (bottom right) using the *feature map* φ , then projecting the transformed training set down to 2D using linear PCA. Notice that if we could inverse the linear PCA step for a given instance in the reduced space, the reconstructed point would lie in feature space, not in the original space (eg. like the one represented by an x in the diagram). Since the feature space is infinite-dimensional, we cannot compute the reconstructed point, therefore we cannot compute the true reconstruction error. Fortunately, it is possible to find a point in the original space that would map close to the reconstructed point. This is called the reconstruction *pre-image*. Once you have this pre-image, you can measure its squared distance to the original instance. You can then select the kernel and hyperparameters that minimize this reconstruction pre-image error.

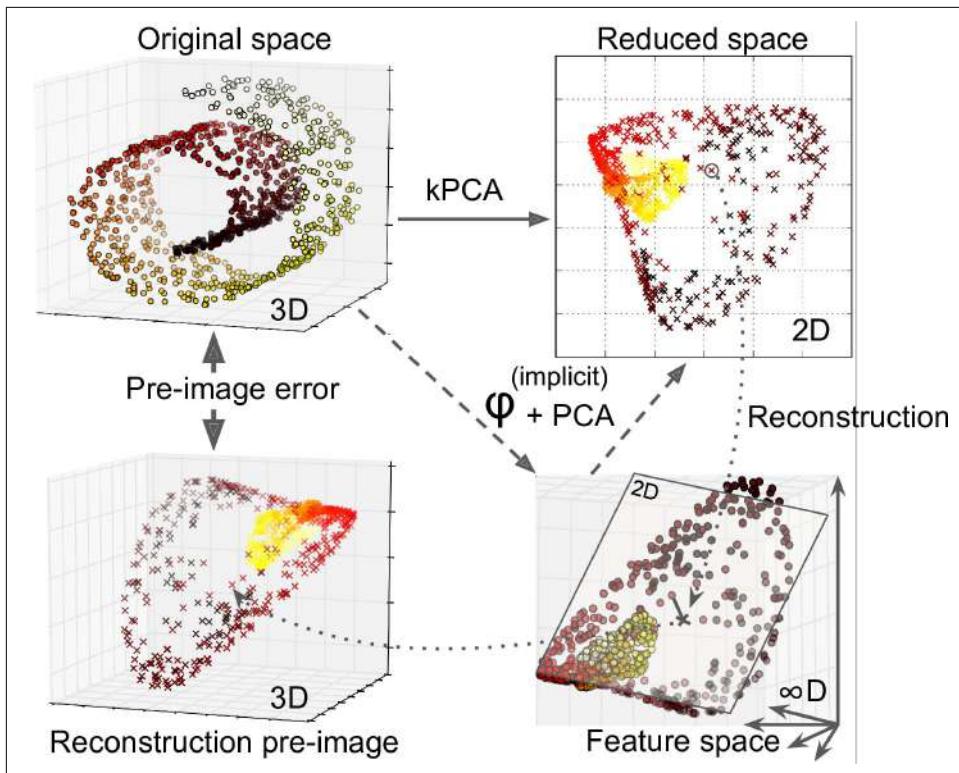


Figure 8-10. Kernel PCA and the reconstruction pre-image error

You may wonder how to perform this reconstruction? One solution is to train a supervised regression model, with the projected instances as the training set and the original instances as the targets. Scikit-Learn will do this automatically if you set `fit_inverse_transform=True`, as shown in the following code⁷:

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                     fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```



By default, `fit_inverse_transform=False` and `KernelPCA` has no `inverse_transform()` method. This method only gets created when you set `fit_inverse_transform=True`.

⁷ Scikit-Learn uses the algorithm based on kernel ridge regression described in “Learning to Find Pre-Images”, G. Bakir *et al.* (2004): <http://goo.gl/d0ydY6>

You can then compute the reconstruction pre-image error:

```
>>> from sklearn.metrics import mean_squared_error  
>>> mean_squared_error(X, X_preimage)  
32.786308795766132
```

Now you can use grid-search with cross-validation to find the kernel and hyperparameters that minimize this pre-image reconstruction error.

LLE

Locally-Linear Embedding (LLE)⁸ is another very powerful *non-linear dimensionality reduction* (NLDR) technique. It is a manifold learning technique that does not rely on projections like the previous algorithms. In a nutshell LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), then it looks for a low-dimensional representation of the training set where these local relationships are best preserved (more details below). This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

For example, the following code uses Scikit-Learn's LocallyLinearEmbedding class to unroll the swiss roll. The resulting 2D dataset is shown in Figure 8-11. As you can see, the swiss roll is completely unrolled and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the left part of the unrolled swiss roll is squeezed while the right part is stretched. Nevertheless, LLE did a pretty good job at modeling the manifold.

```
from sklearn.manifold import LocallyLinearEmbedding  
  
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)  
X_reduced = lle.fit_transform(X)
```

⁸ “Nonlinear Dimensionality Reduction by Locally Linear Embedding”, S. Roweis, L. Saul (2000): <http://goo.gl/5nawF0>

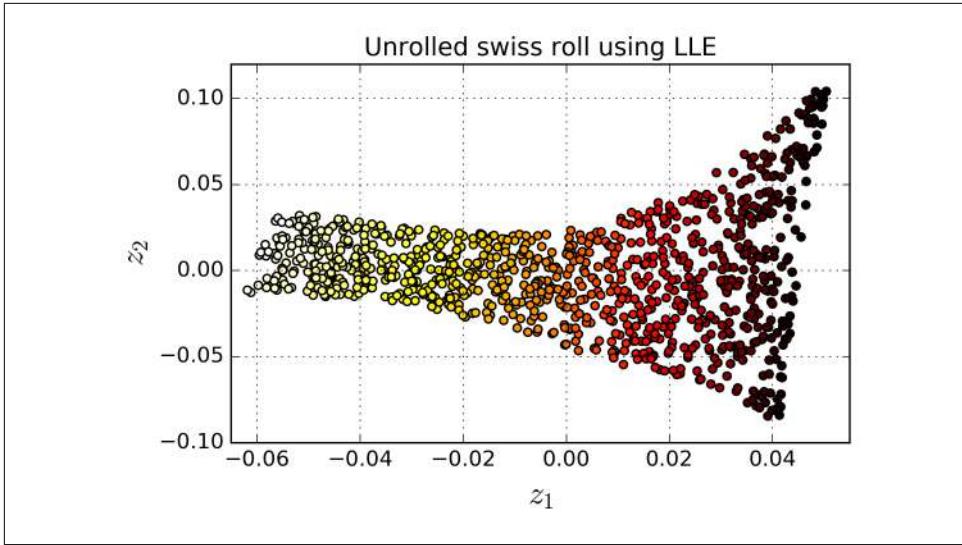


Figure 8-11. Unrolled swiss roll using LLE

Here's how LLE works: first, for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k closest neighbors (in the code above $k = 10$), then it tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it finds the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k closest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in [Equation 8-4](#), where \mathbf{W} is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

[Equation 8-4. LLE step 1: linearly modeling local relationships](#)

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left\| \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right\|^2$$

subject to
$$\begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{for } i = 1, 2, \dots, m \end{cases}$$

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\widehat{w}_{i,j}$) encodes the local linear relationships between the training instances. Now the second step is to map the training instances into a d -dimensional space (where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d -dimensional

space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in [Equation 8-5](#). It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that \mathbf{Z} is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left\| \mathbf{z}^{(i)} - \sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)} \right\|^2$$

Scikit-Learn's LLE implementation has the following computational complexity: $O(m \log(m)n \log(k))$ for finding the k nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

Other dimensionality reduction techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn. Here are some of the most popular:

- *Multi-dimensional Scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances (see [Figure 8-12](#)).
- *Isomap* creates a graph by connecting each instance to its nearest neighbors, then it reduces dimensionality while trying to preserve the *geodesic distances*⁹ between the instances.
- *t-Distributed Stochastic Neighbor Embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high dimensional space (eg. to visualize the MNIST images in 2D).
- *Linear Discriminant Analysis* (LDA) is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a

⁹ The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

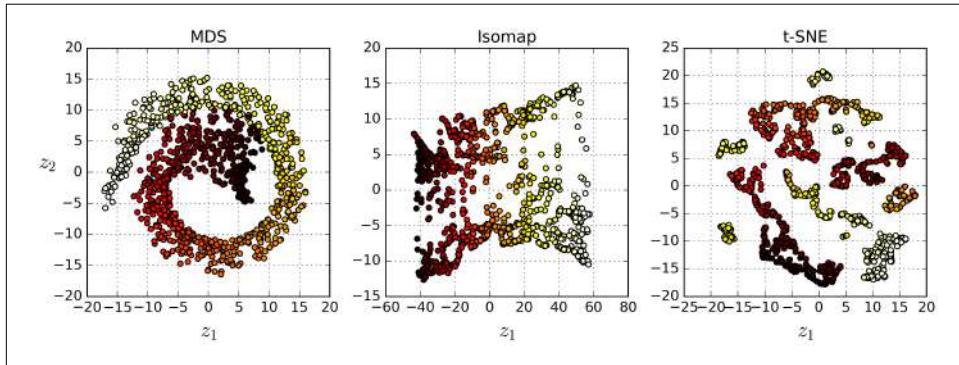


Figure 8-12. Reducing the swiss roll to 2D using various techniques

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly non-linear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%, how many dimensions will the resulting dataset have?
6. In what cases would you use vanilla PCA, Incremental PCA, Randomized PCA or Kernel PCA?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in [Chapter 3](#)) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new Random Forest classifier on the reduced dataset and see how long it

takes. Was training much faster? Next evaluate the classifier on the test set: how does it compare to the previous classifier?

10. Use t-SNE to reduce the MNIST dataset down to 2 dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can write colored digits at the location of each instance, or even plot scaled down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or only plot an instance if no other instance has already been plotted at a close distance). You should get a nice visualization with well separated clusters of digits. Try using other dimensionality reduction algorithms such as PCA, LLE or MDS and compare the resulting visualizations.

Solutions to these exercises are available in [Appendix A](#).

PART II

Neural Networks and Deep Learning

Up and running with TensorFlow

TensorFlow is a powerful open source software library for numerical computation, particularly well suited and fine-tuned for large scale Machine Learning. Its basic principle is simple: you first define in python a graph of computations to perform (for example the one in [Figure 9-1](#)), then TensorFlow takes that graph and runs it efficiently using optimized C++ code.

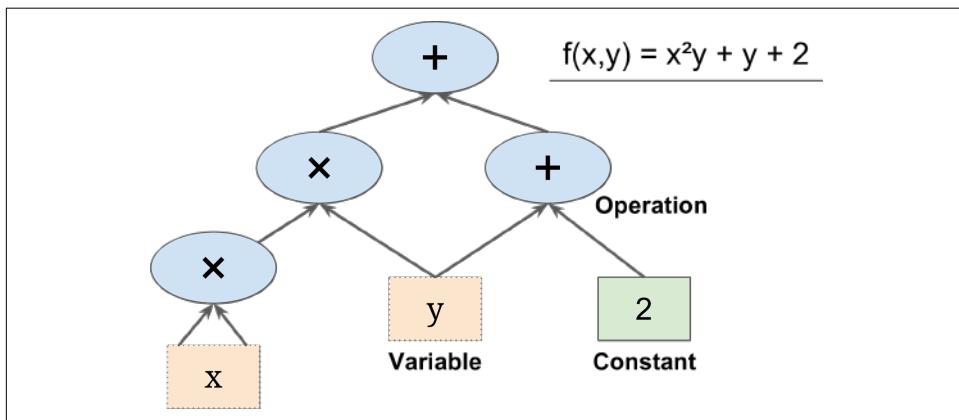


Figure 9-1. A simple computation graph

Most importantly, it is possible to break up the graph into several chunks and run them in parallel across multiple CPUs or GPUs (as shown on [Figure 9-2](#)). TensorFlow also supports distributed computing, so you can train colossal neural networks on humongous training sets in a reasonable amount of time by splitting the computations across hundreds of servers (see [Chapter 12](#)). TensorFlow can train a network with millions of parameters on a training set composed of billions of instances with millions of features each. This should come as no surprise, since TensorFlow was

developed by the Google Brain team and it powers many of Google's large scale services, such as Google Cloud Speech, Google Photos, and Google Search.

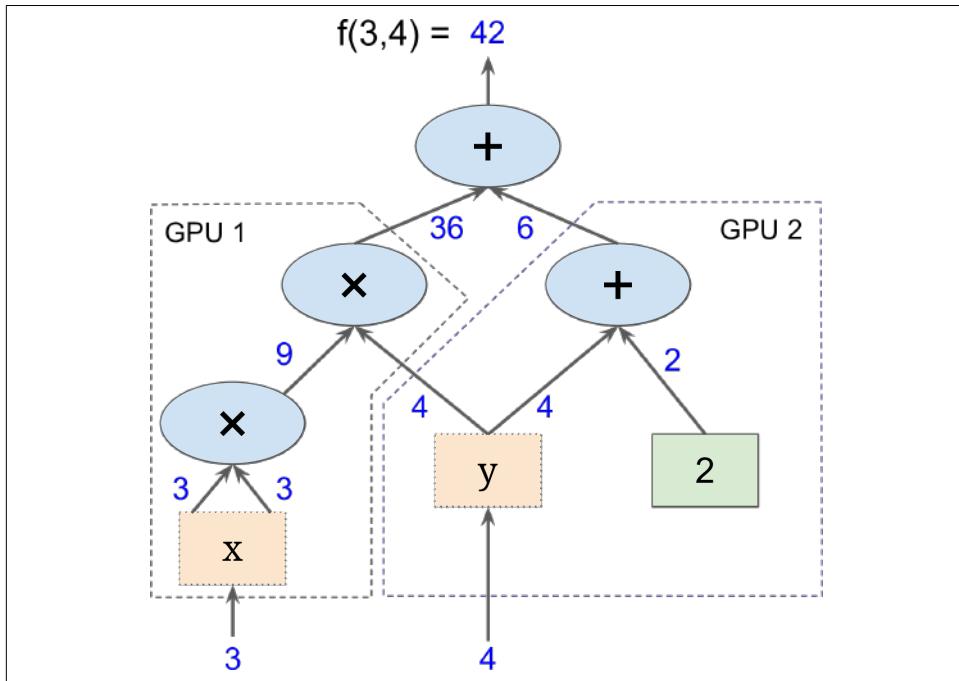


Figure 9-2. Parallel computation on multiple CPUs/GPUs/servers

When TensorFlow was open sourced in November 2015 there were already many popular open source libraries for Deep Learning (Table 9-1 lists a few), and to be fair most of TF's features already existed in one library or another. Nevertheless, TensorFlow's clean design, scalability, flexibility¹ and great documentation (not to mention Google's name) quickly boosted it to the top of the list. Here are some of TF's highlights:

- It runs not only on Linux and MacOSX, but also on mobile devices, including both iOS and Android.
- It provides a very simple python API called *TFLearn*² (tensorflow.com/tflearn), compatible with Scikit-Learn: as you will see, you can use it to train

¹ TF is not limited to neural networks or even Machine Learning, you could run quantum physics simulations if you wanted.

² Not to be confused with the TF Learn library which is an independent project.

various types of neural networks in just a few lines of code. It was previously an independent project called *Scikit Flow* (or *skflow*).

- It also provides another simple API called *TF-slim* (`tensorflow.contrib.slim`) to simplify building, training and evaluating neural networks.
- Its main python API offers much more flexibility (at the cost of higher complexity) to create all sorts of computations, including any neural network architecture you can think of.
- It includes highly efficient C++ implementations of many ML operations, particularly those needed to build neural networks. There is also a C++ API to define your own high-performance operations.
- It provides several advanced optimization nodes to search for the parameters that minimize a cost function. These are very easy to use since TensorFlow automatically takes care of computing the gradients of the functions you define. This is called *automatic differentiating* (or *autodiff*).
- It also comes with a great visualization tool called *TensorBoard* that allows you to browse through the computation graph, view learning curves, and more.
- Google also launched a cloud service to run TF graphs (<https://cloud.google.com/ml/>).
- Last but not least, it has a dedicated team of passionate and helpful developers, and a growing community contributing to improving it. It is one of the most popular open source projects on GitHub, and more and more great projects are being built on top of it (eg. for examples, check out the resources page on <https://www.tensorflow.org/>, or <https://github.com/jtoy/awesome-tensorflow>). To ask technical questions, you should use <http://stackoverflow.com/> and tag your question with "tensorflow". You can file bugs and feature requests through github. For general discussions, join the google group (<http://goo.gl/N7kRF9>).

In this chapter, we will go through the basics of TensorFlow, from installation to creating, running, saving and visualizing simple computational graphs. In the following chapters we will explain how neural networks work and implement various neural network architectures using TensorFlow.

Table 9-1. Open source Deep Learning libraries (not an exhaustive list)

Library	API	Platforms	Started by	Year
Caffe	Python, C++, Matlab	Linux, OS X, Windows	Y. Jia, UC Berkeley (BVLC)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, OS X, Windows, Android	A. Gibson, Skymind	2014
H2O	Python, R	Linux, OS X, Windows	H2O.ai	2014
MXNet	Python, C++, others	Linux, OS X, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, OS X, iOS, Android	Google	2015

Library	API	Platforms	Started by	Year
Theano	Python	Linux, OS X, iOS	University of Montreal	2010
Torch	C++, Lua	Linux, OS X, iOS, Android	R. Collobert, K. Kavukcuoglu, C. Farabet	2002

Installation

Let's get started! Assuming you installed Jupyter and Scikit-Learn by following the installation instructions in [Chapter 2](#), you can simply use pip to install TensorFlow. If you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH          # Your ML working directory (eg. $HOME/ml)
$ source env/bin/activate
```

Next, find the version of TensorFlow that you want to install. There are different builds for Linux and MacOSX, for various versions of python, and builds with GPU support for Ubuntu (check out the installation page on <https://www.tensorflow.org/> for the full list). The following commands install TensorFlow, CPU-Only for MacOSX and python 3.5.

```
$ export TF_BASE_URL=https://storage.googleapis.com/tensorflow
$ export TF_BINARY_URL=$TF_BASE_URL/mac/cpu/tensorflow-0.10.0-py3-none-any.whl
$ pip3 install --upgrade $TF_BINARY_URL
```

At the time of writing (September 2016), TensorFlow does not yet support Windows. This is due to the fact that TensorFlow's build system is Bazel, which is scheduled to support Windows only by Q4 2016. In the meantime your only option is to run TensorFlow within a Linux Virtual Machine (or within a Docker container, which will run within a Linux VM).



If you installed the GPU version of TensorFlow, you also need to install the appropriate version of the Cuda Toolkit and cuDNN (available on <https://developer.nvidia.com/>) and set a couple environment variables. See the installation page on <https://www.tensorflow.org/> for detailed instructions.

To test your installation, type the following command. It should output the version of TensorFlow you installed.

```
$ python3 -c 'import tensorflow; print(tensorflow.__version__)
0.10.0'
```

Creating your first graph and running it in a session

The following code creates the graph represented in [Figure 9-1](#):

```
import tensorflow as tf

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

That's all there is to it! The most important thing to understand is that this code does not actually perform any computation, even though it looks like it does (especially the last line). It just creates a computation graph. In fact even the variables are not initialized yet. To evaluate this graph you need to open a TensorFlow *session* and use it to initialize the variables and evaluate f. A TensorFlow session takes care of placing the operations onto *devices* such as CPUs and GPUs and running them, and it holds all the variable values³. The following code creates a session, initializes the variables, evaluates f then closes the session (which frees up resources).

```
>>> sess = tf.Session()
>>> sess.run(x.initializer)
>>> sess.run(y.initializer)
>>> result = sess.run(f)
>>> print(result)
42
>>> sess.close()
```

Having to repeat `sess.run()` all the time is a bit cumbersome, but fortunately there is a better way:

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

Inside the `with` block, the session is set as the default session. Calling `x.initializer.run()` is equivalent to calling `tf.get_default_session().run(x.initializer)`, and similarly `f.eval()` is equivalent to calling `tf.get_default_session().run(f)`. This makes the code easier to read. Moreover, the session is automatically closed at the end of the block.

Instead of manually running the initializer for every single variable, you can use the `initialize_all_variables()` function. Note that it does not actually perform the initialization immediately, it creates a node in the graph that will initialize all variables when it is run:

```
init = tf.initialize_all_variables() # prepare an init node

with tf.Session() as sess:
```

³ In distributed TensorFlow, variable values are stored on the servers instead of the session, as we will see in [Chapter 12](#).

```
init.run() # actually initialize all the variables
result = f.eval()
```

Inside Jupyter or within a python shell you may prefer to create an `InteractiveSession`. The only difference with a regular `Session` is that when it is created it automatically sets itself as the default session, so you don't need a `with` block (but you do need to close the session manually when you are done with it):

```
>>> sess = tf.InteractiveSession()
>>> init.run()
>>> result = f.eval()
>>> print(result)
42
>>> sess.close()
```

A TensorFlow program is typically split into two parts: the first part builds a computation graph (this is called the *construction phase*), and the second part runs it (this is the *execution phase*). The construction phase typically builds a computation graph representing the ML model and the computations required to train it. The execution phase generally runs a loop that evaluates a training step repeatedly (for example one step per mini-batch) gradually improving the model parameters. We will go through an example below.

Managing graphs

Any node you create is automatically added to the default graph:

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

In most cases this is fine, but sometimes you may want to manage multiple independent graphs. You can do this by creating a new `Graph` and temporarily making it the default graph inside a `with` block, like so:

```
>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```



In Jupyter (or in a python shell), it is common to run the same commands more than once while you are experimenting. As a result, you may end up with a default graph containing many duplicate nodes. One solution is to restart the Jupyter kernel (or the python shell), but a more convenient solution is to just reset the default graph by running `tf.reset_default_graph()`.

Lifecycle of a node value

When you evaluate a node, TensorFlow automatically determines the set of nodes that it depends on and it evaluates these nodes first. For example, consider the following code:

```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval()) # 10
    print(z.eval()) # 15
```

First, this code defines a very simple graph. Then it starts a session and runs the graph to evaluate `y`: TensorFlow automatically detects that `y` depends on `w`, which depends on `x`, so it first evaluates `w`, then `x`, then `y`, and it returns the value of `y`. Finally, the code runs the graph to evaluate `z`. Once again, TensorFlow detects that it must first evaluate `w` and `x`. It is important to note that it will *not* reuse the result of the previous evaluation of `w` and `x`. In short, the code above evaluates `w` and `x` twice.

All node values are dropped between graph runs, except variable values which are maintained by the session across graph runs (queues and readers also maintain some state as we will see in [Chapter 12](#)). A variable starts its life when its initializer is run, and it ends when the session is closed.

If you want to evaluate `y` and `z` efficiently, without evaluating `w` and `x` twice as in the code above, you must ask TensorFlow to evaluate both `y` and `z` in just one graph run, as shown in the following code:

```
with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
    print(y_val) # 10
    print(z_val) # 15
```



In single-process TensorFlow, multiple sessions do not share any state, even if they reuse the same graph (each session would have its own copy of every variable). In Distributed TensorFlow (see [Chapter 12](#)), variable state is stored on the servers, not in the sessions, so multiple sessions can share the same variables.

Linear Regression with TensorFlow

TensorFlow operations (also called *ops* for short) can take any number of inputs and produce any number of outputs. For example, the addition and multiplication ops each take 2 inputs and produce 1 output. Constants and variables take no input (they are called *source ops*). The inputs and outputs are multidimensional arrays, called *tensors* (hence the name “tensor flow”). Just like NumPy arrays, tensors have a type and a shape. In fact in the python API tensors are simply represented by NumPy ndarrays. They typically contain floats, but you can also use them to carry strings (arbitrary byte arrays).

In the examples above, the tensors just contained a single scalar value, but you can of course perform computations on arrays of any shape. For example the following code manipulates 2D arrays to perform Linear Regression on the California housing dataset (introduced in [Chapter 2](#)). It starts by fetching the dataset, then it adds an extra bias input feature ($x_0 = 1$) to all training instances (this is done using NumPy so it runs immediately), then it creates two TensorFlow constant nodes `X` and `y` to hold this data and the targets, and it uses some of the matrix operations provided by TensorFlow to define `theta`. These matrix functions (`transpose()`, `matmul()` and `matrix_inverse()`) are self-explanatory, but as usual they do not perform any computations immediately, instead they create nodes in the graph that will perform them when the graph is run. You may recognize that the definition of `theta` corresponds to the Normal Equation ($\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$, see [Chapter 4](#)). Finally, the code creates a session and uses it to evaluate `theta`.

```
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

The main benefit of this code versus computing the Normal Equation directly using NumPy is that TensorFlow will automatically run this on your GPU card if you have one (provided you installed TensorFlow with GPU support of course, see [Chapter 12](#) for more details).

Implementing Gradient Descent

Let's try using Batch Gradient Descent (see [Chapter 4](#)) instead of the Normal Equation. First we will do this by manually computing the gradients, then we will use TensorFlow's autodiff feature to let TF compute the gradients automatically, and finally we will use a couple of TensorFlow's out of the box optimizers.



When using Gradient Descent, recall that it is important to first normalize the input feature vectors, or else training may be much slower. This can be done using TensorFlow, or NumPy, or Scikit-Learn's `StandardScaler`, or any other solution you prefer. The following code assumes that this has already been done.

Manually computing the gradients

The following code should be fairly self-explanatory, except for a few new elements:

- The `random_uniform()` function creates a node in the graph that will generate a tensor containing random values, given its shape and value range, much like NumPy's `rand()` function.
- The `assign()` function creates a node that will assign a new value to a variable. In this case, it implements the Batch Gradient Descent step $\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$.
- The main loop executes the training step over and over again (`n_epochs` times), and every 100 iterations it prints out the current Mean Squared Error (`mse`). You should see the MSE go down at every iteration.

```
n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

```

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

best_theta = theta.eval()

```

Using autodiff

The code above works fine, but it requires mathematically deriving the gradients from the cost function (MSE). In the case of Linear Regression, it is reasonably easy, but if you had to do this with deep neural networks you would get quite a headache: it would be tedious and error-prone. You could use *symbolic differentiation* to automatically find the equations for the partial derivatives for you, but the resulting code would not necessarily be very efficient.

To understand why, consider the function $f(x) = \exp(\exp(\exp(x)))$. If you know calculus, you can figure out its derivative $f'(x) = \exp(x) \times \exp(\exp(x)) \times \exp(\exp(\exp(x)))$. If you code $f(x)$ and $f'(x)$ separately and exactly as they appear, your code will not be as efficient as it could be. A more efficient solution would be to write a function that first computes $\exp(x)$, then $\exp(\exp(x))$, then $\exp(\exp(\exp(x)))$ and returns all three. This gives you $f(x)$ directly (the 3rd term), and if you need the derivative you can just multiply all three terms and you are done. With the naïve approach you would have had to call the `exp` function 9 times to compute both $f(x)$ and $f'(x)$. With this approach you just need to call it 3 times.

It gets worse when your function is defined by some arbitrary code. Can you find the equation (or the code) to compute the partial derivatives of the following function? Hint: don't even try.

```

def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(a - i)
    return z

```

Fortunately, TF's autodiff feature comes to the rescue: it can automatically and efficiently compute the gradients for you. Simply replace the `gradients = ...` line in the Gradient Descent code in the previous section with the following line, and the code will continue to work just fine:

```
gradients = tf.gradients(mse, [theta])[0]
```

The `gradients()` function takes an op (in this case `mse`) and a list of variables (in this case just `theta`) and it creates a list of ops (one per variable) to compute the gradients of the op with regards to each variable. So the `gradients` node will compute the gradient vector of the MSE with regards to `theta`.

There are mainly 4 approaches to computing gradients automatically. They are summarized in [Table 9-2](#). TensorFlow uses *reverse-mode autodiff*, which is perfect (efficient and accurate) when there are many inputs and few outputs, as is often the case in neural networks. It computes all the partial derivatives of the outputs with regards to all the inputs in just $n_{\text{outputs}} + 1$ graph traversals.

Table 9-2. Main solutions to compute gradients automatically

Technique	Nb of graph traversals to compute all gradients	Accuracy	Supports arbitrary code	Comment
Numerical differentiation	$n_{\text{inputs}} + 1$	Low	Yes	Trivial to implement
Symbolic differentiation	N/A (builds a different graph)	High	No	Implemented by Theano
Forward-mode autodiff	n_{inputs}	High	Yes	Uses <i>dual numbers</i>
Reverse-mode autodiff	$n_{\text{outputs}} + 1$	High	Yes	Implemented by TensorFlow

If you are interested in how this magic works, check out [Appendix C](#).

Using an optimizer

So TensorFlow computes the gradients for you. But it gets even easier: TensorFlow provides a number of optimizers out of the box, including a Gradient Descent optimizer. You can simply replace the `gradients = ...` and `training_op = ...` lines above with the following code, and once again everything will just work fine:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

If you want to use a different type of optimizer, you just need to change one line. For example, you can use a momentum optimizer (which often converges much faster than Gradient Descent, see [Chapter 11](#)) by defining the optimizer like this:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

Feeding data to the training algorithm

Let's try to modify the code above to implement Mini-batch Gradient Descent. For this, we need a way to replace `X` and `y` at every iteration with the next mini-batch. The simplest way to do this is to use placeholder nodes. These nodes are special because they don't actually perform any computation, they just output the data you tell them

to output at runtime. If you don't specify a value at runtime for a placeholder, you get an exception.

To create a placeholder node, you must call the `placeholder()` function and specify the output tensor's data type. Optionally, you can also specify its shape, if you want to enforce it. If you specify `None` for a dimension, it means "any size". For example, the following code creates a placeholder node `A`, and also a node `B = A + 5`. When we evaluate `B`, we pass a `feed_dict` to the `eval()` method that specifies the value of `A`. Note that `A` must have rank 2 (ie. it must be 2-dimensional) and there must be 3 columns (or else an exception is raised), but it can have any number of rows.

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
...     B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
...     B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6.  7.  8.]]
>>> print(B_val_2)
[[ 9. 10. 11.]
 [12. 13. 14.]]
```



You can actually feed the output of *any* operations, not just placeholders. In this case TensorFlow does not try to evaluate these operations, it uses the values you feed it.

To implement Mini-batch Gradient Descent, we only need to tweak the existing code slightly. First change the definition of `X` and `y` in the construction phase to make them placeholder nodes:

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

Then define the batch size and compute the total number of batches:

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

Finally, in the execution phase, fetch the mini-batches one by one, then provide the value of `X` and `y` via the `feed_dict` parameter when evaluating a node that depends on either of them.

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # load the data from disk
    return X_batch, y_batch

with tf.Session() as sess:
```

```

sess.run(init)

for epoch in range(n_epochs):
    for batch_index in range(n_batches):
        X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

best_theta = theta.eval()

```



We don't need to pass the value of `X` and `y` when evaluating `theta` since it does not depend on either of them.

Saving and restoring models

Once you have trained your model, you should save its parameters to disk so you can come back to it whenever you want, use it in another program, compare it to other models, and so on. Moreover, you probably want to save checkpoints at regular intervals during training so in case your computer crashes during training you can continue from the last checkpoint rather than start over from scratch.

TensorFlow makes saving and restoring a model very easy. Just create a `Saver` node at the end of the construction phase (after all variable nodes are created), then in the execution phase just call its `save()` method whenever you want to save the model, passing it the session and path of the checkpoint file:

```

[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.initialize_all_variables()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "my_model_final.ckpt")

```

Restoring a model is just as easy: you create a `Saver` at the end of the construction phase just like above, but then at the beginning of the execution phase, instead of ini-

tializing the variables using the `init` node, you call the `restore()` method of the `Saver` object:

```
with tf.Session() as sess:  
    saver.restore(sess, "/tmp/my_model_final.ckpt")  
    [...]
```

By default a `Saver` saves and restores all variables under their own name, but if you need more control, you can specify which variables to save or restore, and what names to use. For example the following `Saver` will save or restore only the `theta` variable under the name `weights`:

```
saver = tf.train.Saver({"weights": theta})
```

Visualizing the graph and training curves using TensorBoard

So now we have a computation graph that trains a Linear Regression model using Mini-batch Gradient Descent, and we are saving checkpoints at regular intervals. Sounds sophisticated, doesn't it? However we are still relying on the `print()` function to visualize progress during training. There is a better way: enter TensorBoard. If you feed it some training stats it will display nice interactive visualizations of these stats in your web browser (eg. learning curves). You can also provide it the graph's definition and it will give you a great interface to browse through it. This is very useful to identify errors in the graph, to find bottlenecks, and so on.

The first step is to tweak your program a bit so it writes the graph definition and some training stats, for example the training error (MSE), to a log directory that TensorBoard will read from. You need to use a different log directory every time you run your program, or else TensorBoard will merge stats from different runs, which will mess up the visualizations. The simplest solution for this is to include a timestamp in the log directory name. Add the following code at the beginning of the program:

```
from datetime import datetime  
  
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")  
root_logdir = "tf_logs"  
logdir = "{}-run-{}".format(root_logdir, now)
```

Next, add the following code at the very end of the construction phase:

```
mse_summary = tf.scalar_summary('MSE', mse)  
summary_writer = tf.train.SummaryWriter(logdir, tf.get_default_graph())
```

The first line creates a node in the graph that will evaluate the MSE value and write it to a TensorBoard-compatible binary log string called a *summary*. The second line creates a `SummaryWriter` that you will use to write summaries to log files in the log directory. The first parameter indicates the path of the log directory (in this case

something like `tf_logs/run-20160906091959/`, relative to the current directory). The second (optional) parameter is the graph you want to visualize. Upon creation, the `SummaryWriter` creates the log directory if it does not already exist (and its parent directories if needed), and writes the graph definition in a binary log file called an *events file*.

Next you need to update the execution phase to evaluate the `mse_summary` node regularly during training (eg. every 10 mini-batches). This will output a summary that you can then write to the events file using the `summary_writer`. Here is the updated code:

```
[...]
for batch_index in range(n_batches):
    x_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        summary_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```



Avoid logging training stats at every single training step, as this would significantly slow down training.

Finally, you want to close the `SummaryWriter` at the end of the program:

```
summary_writer.close()
```

Now run this program: it will create the log directory and write an events file in this directory, containing both the graph definition and the MSE values. Open up a shell and go to your working directory, then type `ls -l tf_logs/run*` to list the contents of the log directory:

```
$ cd $ML_PATH          # Your ML working directory (eg. $HOME/ml)
$ ls -l tf_logs/run*
total 40
-rw-r--r-- 1 ageron staff 18620 Sep  6 11:10 events.out.tfevents.1472553182.mymac
```

If you run the program a second time, you should see a second directory in the the `tf_logs/` directory:

```
$ ls -l tf_logs/
total 0
drwxr-xr-x 3 ageron  staff  102 Sep  6 10:07 run-20160906091959
drwxr-xr-x 3 ageron  staff  102 Sep  6 10:22 run-20160906092202
```

Great! Now it's time to fire up the TensorBoard server. You need to activate your virtualenv environment if you created one, then start the server by running the `tensorboard` command, pointing it to the root log directory. This starts the TensorBoard web server, listening on port 6006 (which is "goog" written upside down):

```
$ source env/bin/activate  
$ tensorboard --logdir tf_logs/  
Starting TensorBoard on port 6006  
(You can navigate to http://0.0.0.0:6006)
```

Next open a browser and go to `http://0.0.0.0:6006/` (or `http://localhost:6006/`). Welcome to TensorBoard! In the "Events" tab you should see "MSE" on the right. If you click on it, you will see a plot of the MSE during training, for both runs. You can check or uncheck the runs you want to see, zoom in or out, hover over the curve to get details, and so on.



Figure 9-3. Visualizing training stats using TensorBoard

Now click on the "Graphs" tab. You should see the following graph:

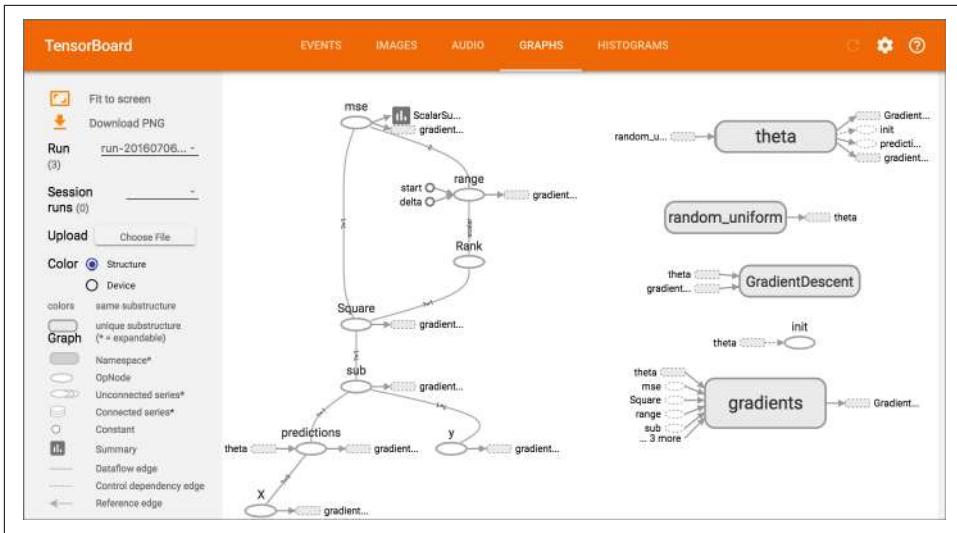


Figure 9-4. Visualizing the graph using TensorBoard

To reduce clutter, the nodes that have many *edges* (ie. connections to other nodes) are separated out to an auxiliary area on the right (you can move a node back and forth between the main graph and the auxiliary area by right clicking on it). Some parts of the graph are also collapsed by default. For example, try hovering over the `gradients` node, then click on the \oplus icon to expand this subgraph. Next, in this subgraph, try expanding the `mse_grad` subgraph.



If you want a quick way to take a peek at the graph directly within Jupyter, you can use the `show_graph()` function available in the notebook for this chapter. It was originally written by A. Mordvintsev in his great deepdream tutorial notebook (<http://goo.gl/EtCWUc>). Another option is to install E. Jang's TensorFlow debugger tool (<https://github.com/ericjang/tdb>) which includes a Jupyter extension for graph visualization (and more).

Name scopes

When dealing with more complex models such as neural networks, the graph can easily become cluttered with thousands of nodes. To avoid this, you can create *name scopes* to group related nodes. For example, let's modify the code above to define the `error` and `mse` ops within a name scope called "loss":

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

The name of each op defined within the scope is now prefixed with "loss/":

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

In TensorBoard, the `mse` and `error` nodes now appear inside the `loss` namespace, which appears collapsed by default:

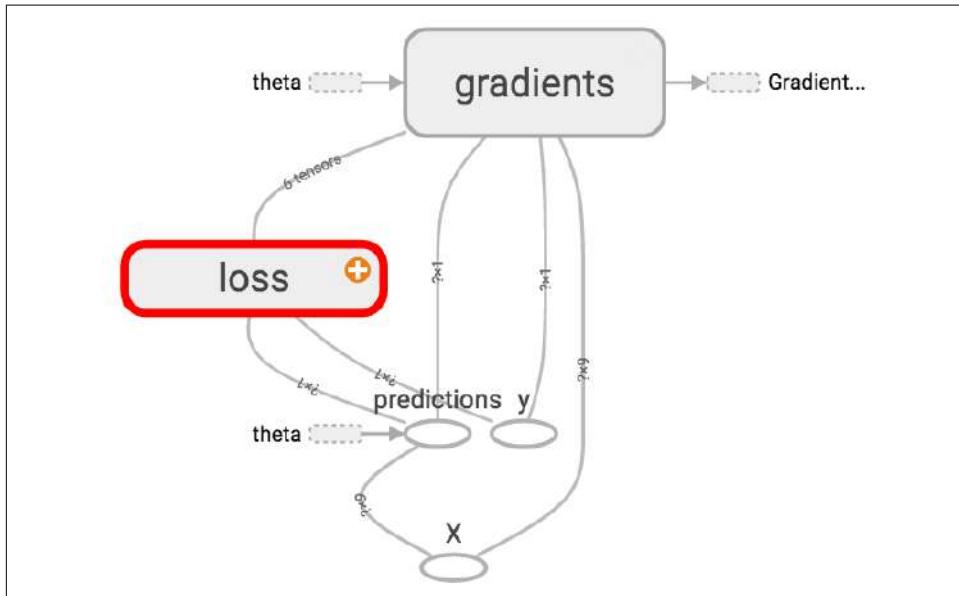


Figure 9-5. A collapsed namescope in TensorBoard

Modularity

Suppose you want to create a graph that adds the output of two *Rectified Linear Units* (ReLU). A ReLU computes a linear function of the inputs, and outputs the result if it is positive, or else it outputs 0 as shown in [Equation 9-1](#):

Equation 9-1. Rectified Linear Unit (ReLU)

$$h_{\mathbf{w}, b}(\mathbf{X}) = \max (\mathbf{X} \cdot \mathbf{w} + b, 0)$$

The following code does the job, but it's quite repetitive:

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
```

```

w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")

output = tf.add_n([relu1, relu2], name="output")

```

Such repetitive code is hard to maintain and error-prone (in fact, the code above contains a cut&paste error, did you spot it?). It would become even worse if you wanted to add a few more ReLUs. Fortunately, TensorFlow lets you stay DRY (Don't Repeat Yourself): simply create a function to build a ReLU. The following code creates 5 ReLUs and outputs their sum:

```

def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")

```

Note that when you create a node, TensorFlow checks whether its name already exists, and if it does it appends an underscore followed by an index to make the name unique. So the first ReLU contains nodes named "weights", "bias", "z" and "relu" (plus many more nodes with their default name, such as "MatMul"), but the second ReLU contains nodes named "weights_1", "bias_1", and so on, the third ReLU contains nodes named "weights_2", "bias_2", etc. TensorBoard identifies such series and collapses them together to reduce clutter (as you can see in [Figure 9-6](#)).

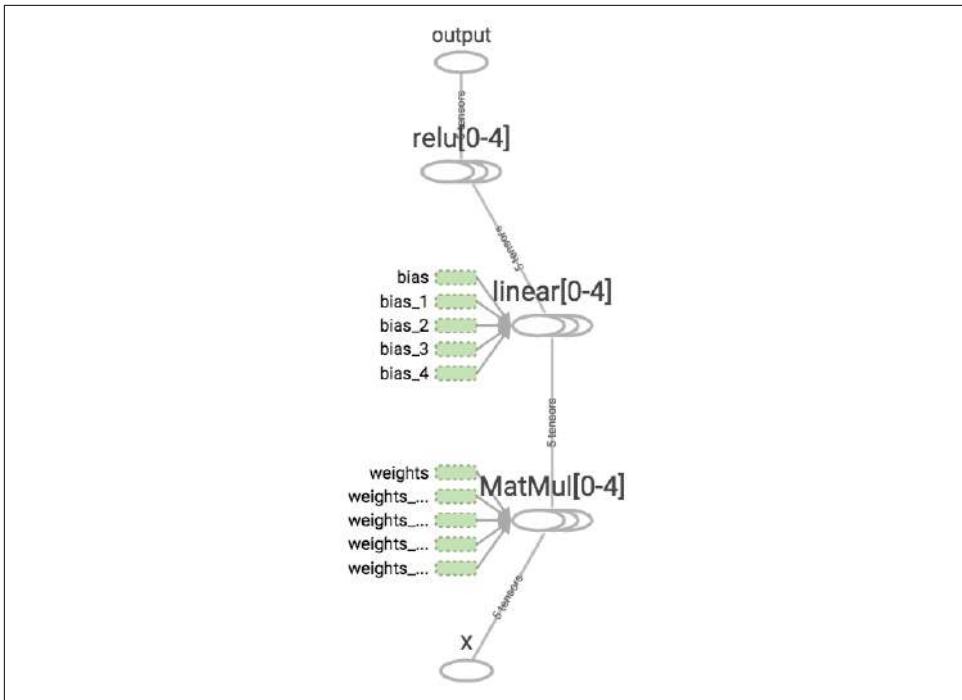


Figure 9-6. Collapsed node series

Using name scopes, you can make the graph much clearer. Simply move all the content of the `relu()` function inside a name scope. Figure 9-7 shows the resulting graph. Notice that TensorFlow also takes care of making the name of name scopes unique by appending `_1`, `_2`, etc.

```
def relu(X):
    with tf.name_scope("relu"):
        [...]
```

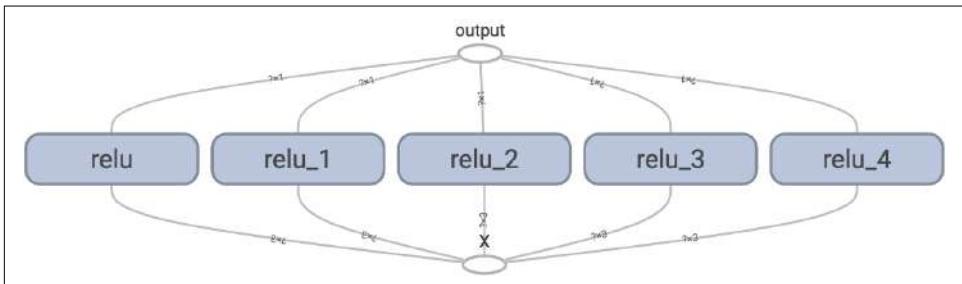


Figure 9-7. A clearer graph using name scoped units

Sharing variables

If you want to share a variable between various components of your graph, one simple option is to create it first, then pass it as a parameter to the functions that need it. For example, suppose you want to control the ReLU threshold (currently hard-coded to 0) using a shared `threshold` variable for all ReLUs. You could just create that variable first, and then pass it to the `relu()` function:

```
def relu(X, threshold):
    with tf.name_scope("relu"):
        [...]
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

This works fine: now you can control the threshold for all ReLUs using the `threshold` variable. However, if there are many shared parameters such as this one, it is going to be painful to have to pass them around as parameters all the time. Many people create a python dictionary containing all the variables in their model, and pass it around to every function. Others create a class for each module (eg. a ReLU class using class variables to handle the shared parameter). Yet another option is to set the shared variable as an attribute of the `relu()` function upon the first call, like so:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
        return tf.maximum(z, relu.threshold, name="max")
```

TensorFlow offers another option, which may lead to slightly cleaner and more modular code than the solutions above⁴. This solution is a bit tricky to understand at first, but since it is used a lot in TensorFlow it is worth going into a bit of details. The idea is to use the `get_variable()` function to create the shared variable if it does not exist yet, or reuse it if it already exists. The desired behavior (creating or reusing) is controlled by an attribute of the current `variable_scope()`. For example, the following code will create a variable named "`relu/threshold`" (as a scalar, since `shape=()`, and using `0.0` as the initial value):

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
```

⁴ Creating a ReLU class is arguably the cleanest option, but it is rather heavy-weight.

Note that if the variable has already been created by an earlier call to `get_variable()`, then this code will raise an exception. This behavior prevents reusing variables by mistake. If you want to reuse a variable, you need to explicitly say so by setting the variable scope's `reuse` attribute to `True` (in which case you don't have to specify the shape or the initializer):

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

This code will fetch the existing "relu/threshold" variable (or raise an exception if it does not exist or if it was not created using `get_variable()`). Alternatively, you can set the `reuse` attribute to `True` inside the block by calling the scope's `reuse_variables()` method:

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```



Once `reuse` is set to `True`, it cannot be set back to `False` within the block. Moreover, if you define other variable scopes inside this one, they will automatically inherit `reuse=True`. Lastly, only variables created by `get_variable()` can be reused this way.

Now you have all the pieces you need to make the `relu()` function access the `threshold` variable without having to pass it as a parameter:

```
def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold") # reuse existing variable
        [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"): # create the variable
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    relus = [relu(X) for relu_index in range(5)]
    output = tf.add_n(relus, name="output")
```

This code first defines the `relu()` function, then it creates the `relu/threshold` variable (as a scalar that will later be initialized to 0.0) and builds 5 ReLUs by calling the `relu()` function. The `relu()` function reuses the `relu/threshold` variable, and creates the other ReLU nodes.



Variables created using `get_variable()` are always named using the name of their `variable_scope` as a prefix (eg. "relu/threshold"), but for all other nodes (including variables created with `tf.Variable()`) the variable scope acts like a new name scope. In particular, if a name scope with an identical name was already created, then a suffix is added to make the name unique. For example, all nodes created in the code above (except the `threshold` variable) have a name prefixed with "relu_1/" to "relu_5/", as shown in Figure 9-8.

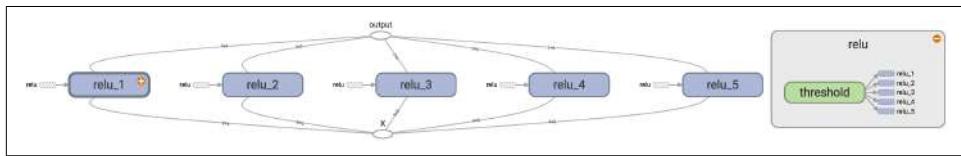


Figure 9-8. Five ReLUs sharing the `threshold` variable

It is somewhat unfortunate that the `threshold` variable must be defined outside of the `relu()` function, where all the rest of the ReLU code resides. To fix this, the following code creates the `threshold` variable within the `relu()` function upon the first call, then reuses it in subsequent calls. Now the `relu()` function does not have to worry about name scopes or variable sharing: it just calls `get_variable()`, which will create or reuse the `threshold` variable (it does not need to know which is the case). The rest of the code calls `relu()` 5 times, making sure to set `reuse=False` on the first call, and `reuse=True` for the other calls.

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
```

The resulting graph is slightly different than before, since the shared variable lives within the first ReLU (see Figure 9-9).

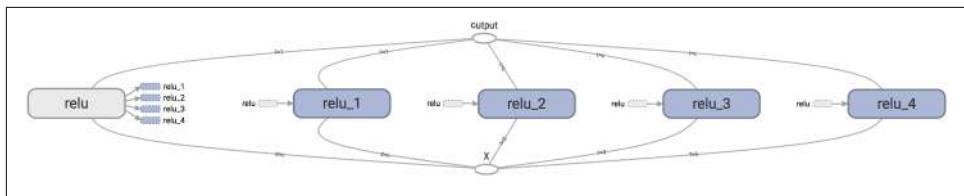


Figure 9-9. Five ReLUs sharing the *threshold* variable

This concludes this introduction to TensorFlow. We will discuss more advanced topics as we go through the following chapters, in particular many operations related to deep neural networks, convolutional neural networks, recurrent neural networks and how to scale up with TensorFlow using multithreading, queues, multiple GPUs and multiple servers.

Exercises

1. What are the main benefits of creating a computation graph rather than directly executing the computations? What are the main drawbacks?
2. Is the statement `a_val = a.eval(session=sess)` equivalent to `a_val = sess.run(a)`?
3. Is the statement `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` equivalent to `a_val, b_val = sess.run([a, b])`?
4. Can you run two graphs in the same session?
5. If you create a graph `g` containing a variable `w`, then start two threads and open a session in each thread, both using the same graph `g`, will each session have its own copy of the variable `w` or will it be shared?
6. When is a variable initialized? When is it destroyed?
7. What is the difference between a placeholder and a variable?
8. What happens when you run the graph to evaluate an operation that depends on a placeholder but you don't feed its value? What happens if the operation does not depend on the placeholder?
9. When you run a graph, can you feed the output value of any operation, or just the value of placeholders?
10. How can you set a variable to any value you want (during the execution phase)?
11. How many times does reverse-mode autodiff need to traverse the graph in order to compute the gradients of the cost function with regards to 10 variables? What about forward-mode autodiff? And symbolic differentiation?

12. Implement Logistic Regression with Mini-batch Stochastic Gradient Descent using TensorFlow. Train it and evaluate it on the moons dataset (introduced in [Chapter 5](#)). Try adding all the bells and whistles:

- define the graph within a `logistic_regression()` function that can be reused easily,
- save checkpoints using a `Saver` at regular intervals during training, save the final model at the end of training,
- restore the last checkpoint upon startup if training was interrupted,
- define the graph using nice scopes so the graph looks good in TensorBoard,
- add summaries to visualize the learning curves in TensorBoard,
- try tweaking some hyperparameters such as the learning rate or the mini-batch size and look at the shape of the learning curve.

Solutions to these exercises are available in [Appendix A](#).

Introduction to Artificial Neural Networks

Birds inspired us to fly, burdock plants inspired velcro, and many other inventions were inspired by Nature. It seems only logical to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the key idea that inspired *Artificial Neural Networks* (ANNs). However, although planes were inspired by birds, they don't have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (eg. by saying "units" rather than "neurons"), lest we restrict our creativity to biologically plausible systems¹.

ANNs are at the very core of Deep Learning. They are versatile, powerful and scalable, making them ideal to tackle large and highly complex Machine Learning tasks, such as classifying billions of images (eg. Google Images), powering speech recognition services (eg. Apple Siri), recommending the best videos to watch to hundreds of millions of users every day (eg. YouTube), or learning to beat the world champion at the game of Go by examining millions of past games then playing against itself (DeepMind's AlphaGo).

In this chapter, we will introduce Artificial Neural Networks, starting with a quick tour of the very first ANN architectures, then we will present *Feedforward Neural Networks* (FNN) and implement one using TensorFlow to tackle the MNIST digit classification problem (introduced in [Chapter 3](#)). Finally, we will take a quick look at some of the other popular ANN architectures.

¹ You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.

In the following chapters, we will discuss techniques to train Deep Neural Networks on huge datasets using TensorFlow across hundreds of servers and GPUs, and we will go through a few other important ANN architectures.

From biological to artificial neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper², they presented a simplified computational model of how *biological neurons* might work together in animal brains to perform complex computations using *propositional logic*. This was the first Artificial Neural Network architecture. Since then many other architectures have been invented (we will discuss some of the most famous below).

The early successes of ANNs until the 1960s led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear that this promise would go unfulfilled (at least for quite a while) funding flew elsewhere and ANNs entered a long dark era. In the early 1980s there was a revival of interest in ANNs as new network architectures were invented and better training techniques were developed. But by the 1990s, powerful alternative Machine Learning techniques such as Support Vector Machines (see [Chapter 5](#)) were favored by most researchers, as they seemed to offer better results and stronger theoretical foundations. Finally, we are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? There are a few good reasons to believe that this one is different and will have a much more profound impact on our lives:

1. There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
2. The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time.
3. The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.
4. Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is

² “A Logical Calculus of Ideas Immanent in Nervous Activity”, W. McCulloch and W. Pitts (1943): <http://goo.gl/9BBRYR>

rather rare in practice (or when it is the case, the local optimum is usually fairly close to the global optimum).

5. ANNs seem to have entered a virtuous circle of funding and progress: amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding towards ANNs, resulting in more and more progress, and even more amazing products.

Biological neurons

Before we discuss *artificial neurons*, let's take a quick look at a *biological neuron* (represented in [Figure 10-1³](#)): it is an unusual looking cell mostly found in animal cerebral cortices (eg. your brain), composed of a *cell body* containing the nucleus and most of the cell's complex components, and many branching extensions called *dendrites*, plus one very long extension called the *axon*. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called *synaptic terminals* (or simply *synapses*) which are connected to the dendrites (or directly to the cell body) of other neurons. Biological neurons receive short electrical impulses called *signals* from other neurons *via* these synapses. When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.

³ Image by Bruce Blaus ([Creative Commons 3.0](#)). Reproduced from <https://en.wikipedia.org/wiki/Neuron>.

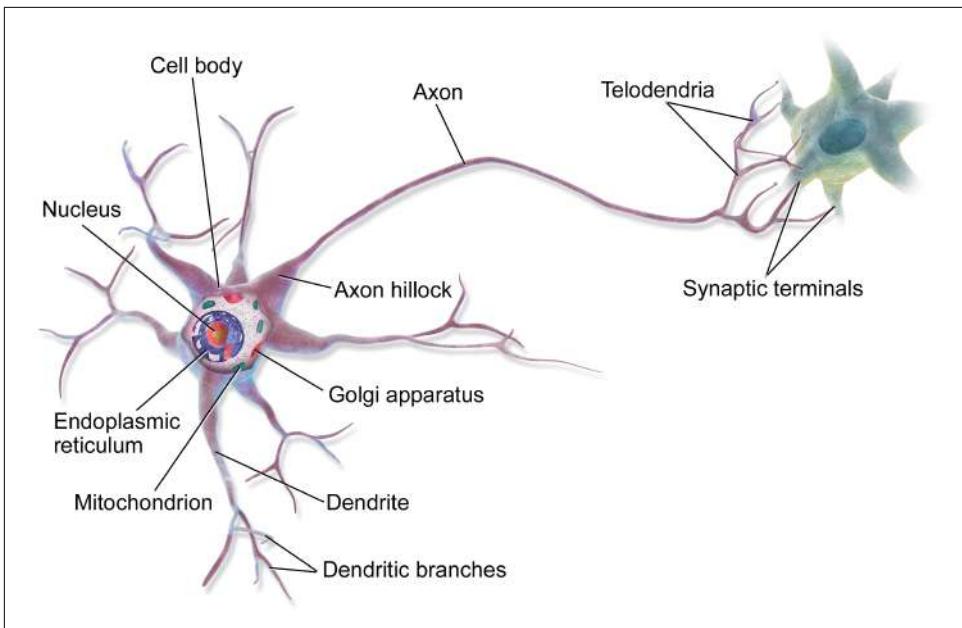


Figure 10-1. Biological neuron

Thus individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions of neurons, each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a vast network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of Biological Neural Networks (BNN⁴) is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, as shown on [Figure 10-2](#).

⁴ In the context of Machine Learning, the phrase “neural networks” generally refers to ANNs, not BNNs.

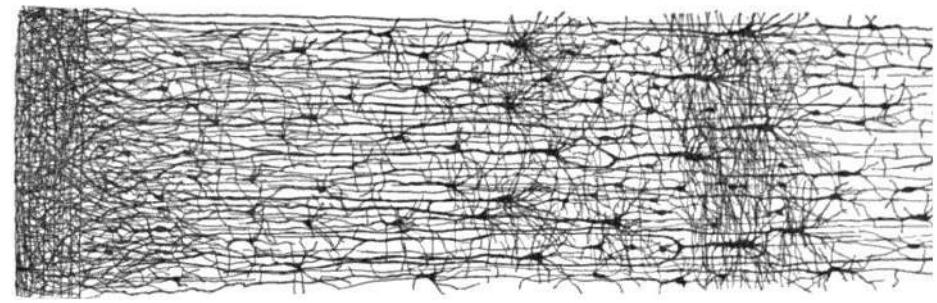


Figure 10-2. Multiple layers in a biological neural network (human cortex)⁵

Logical computations with neurons

Warren McCulloch and Walter Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*: it has one or more binary (on/off) inputs and one binary output. The artificial neuron simply activates its output when more than a certain number of its inputs are active. McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. For example, let's build a few Artificial Neural Networks that perform various logical computations (see Figure 10-3), assuming that a neuron is activated when at least two of its inputs are active.

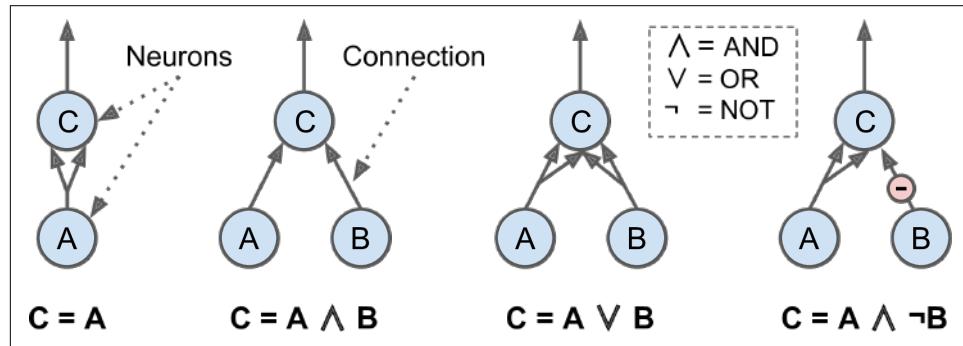


Figure 10-3. Artificial Neural Networks performing simple logical computations

- The first network on the left is simply the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A), but if neuron A is off, then neuron C is off as well.

⁵ Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from https://en.wikipedia.org/wiki/Cerebral_cortex.

- The second network performs a logical AND: neuron C is only activated when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is only activated if neuron A is active and if neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and *vice versa*.

You can easily imagine how the above networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter).

The Perceptron

The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by F. Rosenblatt. It is based on a slightly different artificial neuron (see [Figure 10-4](#)) called a *Linear Threshold Unit* (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight. The LTU computes a weighted sum of its inputs ($z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{w}^T \cdot \mathbf{x}$) then it applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{w}^T \cdot \mathbf{x})$.

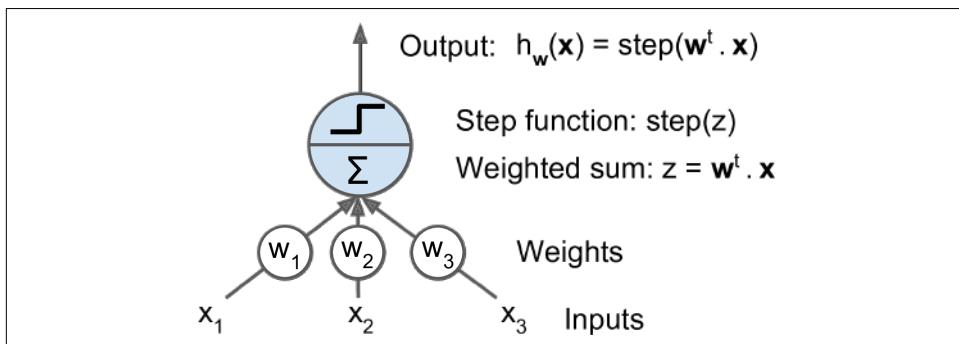


Figure 10-4. Linear Threshold Unit

The most common step function used in Perceptrons is the *Heaviside step function* (see [Equation 10-1](#)). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single LTU can be used for simple linear binary classification: it computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else it outputs the negative class (just like a Logistic Regression classifier or a linear SVM). For example, you could use a single LTU to classify Iris flowers based on the petal length and width (also adding an extra bias feature $x_0 = 1$, just like we did in previous chapters). Training an LTU means finding the right values for w_0 , w_1 and w_2 (the training algorithm is discussed below).

A Perceptron is simply composed of a single layer of LTUs⁶, with each neuron connected to all the inputs. These connections are often represented using special pass-through neurons called *input neurons*: they just output whatever input they are fed. Moreover, an extra bias feature is generally added ($x_0 = 1$). This bias feature is typically represented using a special type of neuron called a *bias neuron* which just outputs 1 all the time.

A Perceptron with 2 inputs and 3 outputs is represented on [Figure 10-5](#). This Perceptron can classify instances simultaneously into 3 different binary classes, which makes it a multioutput classifier.

⁶ The name “Perceptron” is sometimes used to mean a tiny network with a single LTU.

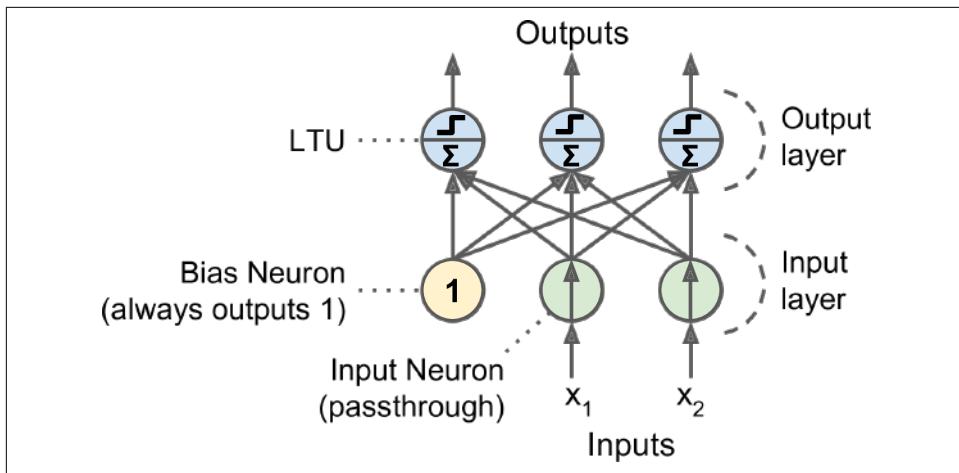


Figure 10-5. Perceptron diagram

So how is a Perceptron trained? The Perceptron training algorithm proposed by F. Rosenblatt was largely inspired by *Hebb's rule*. In his book “The organization of behavior” published in 1949, Donald Hebb suggested that when a biological neuron often triggers another neuron, then the connection between these two neurons grows stronger. This idea was later summarized by S. Löwel's catchy phrase: “Cells that fire together, wire together”. This rule later became known as Hebb's rule (or *Hebbian learning*): the connection weight between two neurons is increased whenever they have the same output. Perceptrons are trained using a variant of this rule that takes into account the error made by the network: it does not reinforce connections that lead to the wrong output. More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in [Equation 10-2](#).

[Equation 10-2. Perceptron learning rule \(weight update\)](#)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- \hat{y}_j is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.

- η is the learning rate.

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, F. Rosenblatt demonstrated that this algorithm would converge to a solution. This is called the *Perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class that implements a single LTU network. It can be used pretty much as you would expect, for example on the Iris dataset (introduced in [Chapter 4](#)):

```
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

You may have recognized that the Perceptron learning algorithm strongly resembles Stochastic Gradient Descent. In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (the learning rate), and `penalty=None` (no regularization).

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability, they just make predictions based on a hard threshold. This is one of the good reasons to prefer Logistic Regression over Perceptrons.

In their 1969 monograph titled “Perceptrons”, M. Minsky and S. Papert highlighted a number of serious weaknesses of Perceptrons, in particular the fact that they are incapable of solving some trivial problems (eg. the *Exclusive OR* (XOR) classification problem, see the left of [Figure 10-6](#)). Of course this is true of any other linear classification model as well (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons, and their disappointment was great: as a result, many researchers dropped *connectionism* altogether (ie. the study of neural networks) in favor of higher-level problems such as logic, problem solving and search.

However, it turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons: the resulting ANN is called a *Multi-Layer Perceptron* (MLP). In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right of [Figure 10-6](#), for each com-

bination of inputs: with inputs (0, 0) or (1, 1) the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.

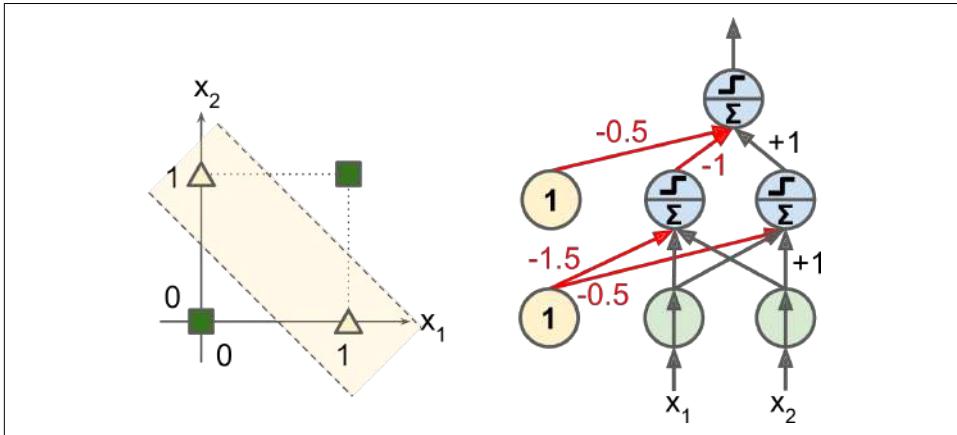


Figure 10-6. XOR classification problem and an MLP that solves it

Multi-Layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) input layer, one or more layers of LTUs, called *hidden layers*, and one final layer of LTUs called the *output layer* (see Figure 10-7). Every layer except the output layer includes a bias neuron and is fully connected to the next layer. When an ANN has two or more hidden layers, it is called a *Deep Neural Network* (DNN).

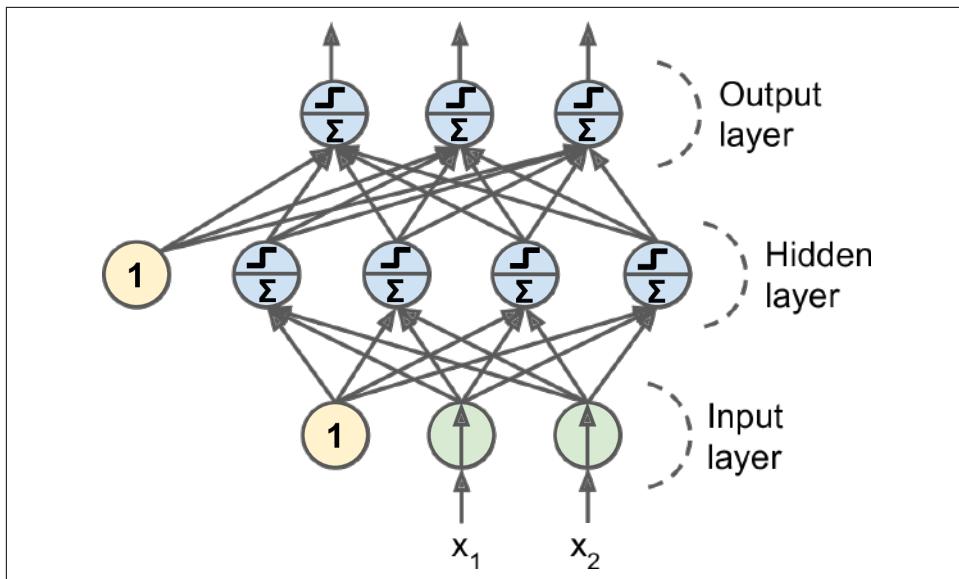


Figure 10-7. Multi-Layer Perceptron

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, D. Rumelhart *et al.* published a groundbreaking article⁷ introducing the *backpropagation* training algorithm⁸. Today we would describe it as Gradient Descent using reverse-mode autodiff (Gradient Descent was introduced in [Chapter 4](#), and autodiff is discussed in [Chapter 9](#)).

For each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer (this is the forward pass, just like when making predictions). Then it measures the network's output error (ie. the difference between the desired output and the actual output of the network), and it computes how much each neuron in the last hidden layer contributed to each output neuron's error. It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer. And so on until the algorithm reaches the input layer. This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backwards in the network (hence the name of the algorithm). If you check out the reverse-mode autodiff algorithm in [Appendix C](#), you will find that the forward and reverse passes of backpropagation simply perform reverse-mode autodiff. The last

⁷ “Learning Internal Representations by Error Propagation”, D. Rumelhart, G. Hinton, R. Williams (1986): <http://goo.gl/NrBzLN>

⁸ This algorithm was actually invented several times by various researchers in different fields, starting with P. Werbos in 1974.

step of the backpropagation algorithm is a Gradient Descent step on all the connection weights in the network, using the error gradients measured earlier.

Let's make this even shorter: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally it slightly tweaks the connection weights to reduce the error (Gradient Descent step).

In order for this algorithm to work properly, the authors made a key change to the MLP's architecture: they replaced the step function by the logistic function, $\sigma(z) = 1/(1 + \exp(-z))$. This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent is stuck), while the logistic function has a well-defined non-zero derivative everywhere, allowing Gradient Descent to make some progress at every step. The backpropagation algorithm may be used with other *activation functions*, instead of the logistic function. Two other popular activation functions are:

- The *hyperbolic tangent* function $\tanh(z) = 2\sigma(2z) - 1$: just like the logistic function it is "S" shaped, continuous and differentiable, but its output value ranges from -1 to 1 (instead of 0 to 1 in the case of the logistic function) which tends to make each layer's output more or less normalized (ie. centered around zero). This often helps speed up convergence.
- The ReLU function (introduced in [Chapter 9](#)): $\text{ReLU}(z) = \max(0, z)$. It is continuous but unfortunately not differentiable at $z = 0$. However in practice it works very well and has the advantage of being extremely fast to compute. Moreover, the fact that it does not have a maximum output value also helps reduce some issues during Gradient Descent (namely gradient saturation).

These popular activation functions and their derivatives are represented in [Figure 10-8](#).

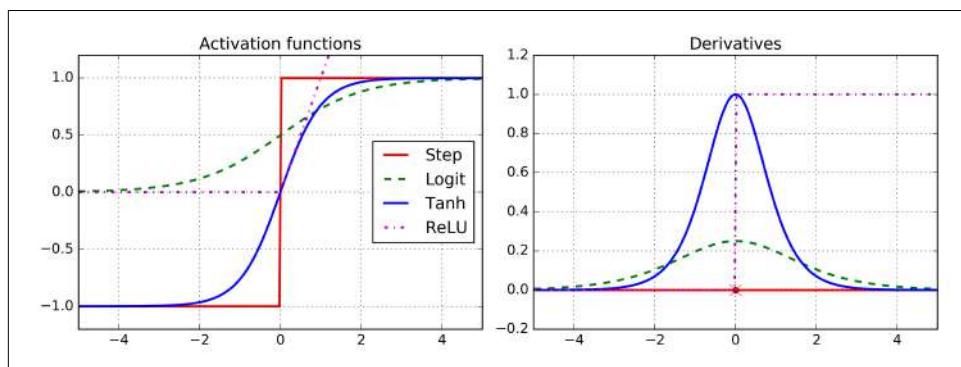


Figure 10-8. Activation functions and their derivatives

An MLP is often used for classification, with each output corresponding to a different binary class (eg. spam/ham, urgent/not-urgent, and so on). When the classes are exclusive (eg. classes 0 through 9 for digit image classification), the output layer is typically modified by replacing the individual activation functions by a shared *softmax* function (see [Figure 10-9](#)). The softmax function was introduced in [Chapter 3](#). The output of each neuron corresponds to the estimated probability of the corresponding class. This architecture is starting to look fairly different from the standard Multi-Layer Perceptron, so it is often referred to simply as a Feedforward Neural Network (FNN), since the signal flows only in one direction (from inputs to outputs).

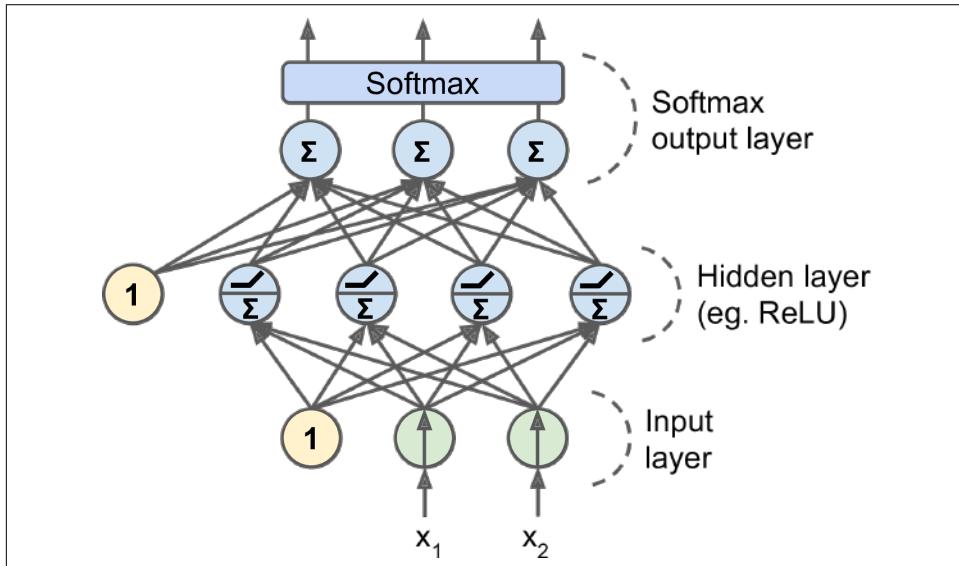


Figure 10-9. A Feedforward Neural Network for classification



Biological neurons seem to implement a roughly sigmoid activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that the ReLU activation function generally works better in ANNs. This is one of the cases where the biological analogy was misleading.

Training an FNN with TensorFlow's high level API

The simplest way to train an FNN with TensorFlow is to use the high level API TF.Learn, which is quite similar to Scikit-Learn's API. The `DNNClassifier` class makes it trivial to train a Deep Neural Network with any number of hidden layers, and a softmax output layer to output estimated class probabilities. For example, the following code trains a DNN for classification with two hidden layers (one with 300

neurons, and the other with 100 neurons) and a softmax output layer with 10 neurons:

```
import tensorflow as tf

feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10,
                                         feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)
```

If you run this code on the MNIST dataset (after scaling it, eg. using Scikit-Learn's `StandardScaler`), you may actually get a model that achieves over 98.1% accuracy on the test set! That's better than the best model we trained in [Chapter 3](#):

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = dnn_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9818000000000001
```

The TF.Learn library also provides some convenience functions to evaluate models:

```
>>> dnn_clf.evaluate(X_test, y_test)
{'accuracy': 0.98180002, 'global_step': 40000, 'loss': 0.073678359}
```

Under the hood, the `DNNClassifier` class creates all the neuron layers, based on the ReLU activation function (this can be changed by setting the `activation_fn` hyper-parameter). The output layer relies on the softmax function, and the cost function is cross entropy (introduced in [Chapter 4](#)).



The TF.Learn API (and TensorFlow in general) is still quite new, so some of the names and functions used in these examples may evolve a bit by the time you read this book. However, the general ideas should not change.

Training a DNN using plain TensorFlow

If you want more control over the architecture of the network, you may prefer to use TensorFlow's lower level python API (introduced in [Chapter 9](#)). In this section we will build the same model as above using this API, and we will implement Mini-batch Gradient Descent to train it on the MNIST dataset. The first step is to build the TensorFlow graph: that's the construction phase. The second step is the execution phase, where you actually run the graph to train the model.

Construction phase

Let's start. First we need to import the `tensorflow` library. Then we must specify the number of inputs and outputs, and set the number of hidden neurons in each layer:

```

import tensorflow as tf

n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

```

Next, just like you did in [Chapter 9](#), you can use placeholder nodes to represent the training data and targets. The shape of X is only partially defined: we know that it will be a 2D tensor (ie. a matrix), with instances along the first dimension and features along the second dimension, we know that the number of features is going to be 28×28 (one feature per pixel) but we don't know yet how many instances each training batch will contain. So the shape of X is $(\text{None}, n_{\text{inputs}})$. Similarly, we know that y will be a 1D tensor with one entry per instance, but again we don't know the size of the training batch at this point, so the shape is (None) .

```

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

```

Now let's create the actual neural network. The placeholder X will act as the input layer: during the execution phase, it will be replaced with one training batch at a time (note that all the instances in a training batch will be processed simultaneously by the neural network). Now you need to create the two hidden layers and the output layer. The two hidden layers are almost identical: they only differ by the inputs they are connected to and by the number of neurons they contain. The output layer is also very similar, but it uses a softmax activation function instead of a ReLU activation function. So let's create a `neuron_layer()` function that we will use to create one layer at a time. It will need parameters to specify the inputs, the number of neurons, the activation function, and the name of the layer:

```

def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation=="relu":
            return tf.nn.relu(z)
        else:
            return z

```

Let's go through this code line by line:

- First we create a name scope using the name of the layer: it will contain all the computation nodes for this neuron layer. This is optional, but the graph will look much nicer in TensorBoard if its nodes are well organized.

- Next, we get the number of inputs by looking up the input matrix's shape and getting the size of the second dimension (the first dimension is for instances).
- The next three lines create a `W` variable that will hold the weights matrix. It will be a 2D tensor containing all the connection weights between each input and each neuron, hence its shape will be `(n_inputs, n_neurons)`. It will be initialized randomly, using a truncated⁹ normal (Gaussian) distribution with a standard deviation of $2/\sqrt{n_{\text{inputs}}}$. Using this specific standard deviation helps the algorithm converge much faster (we will discuss this further in [Chapter 11](#), it is one of those small tweaks to neural networks that have had a tremendous impact on their efficiency). It is important to initialize connection weights randomly for all hidden layers to avoid any symmetries that the Gradient Descent algorithm would be unable to break¹⁰.
- The next line creates a `b` variable for biases, initialized to zero (no symmetry issue in this case), with one bias parameter per neuron.
- Then we create a subgraph to compute $\mathbf{z} = \mathbf{X} \cdot \mathbf{W} + \mathbf{b}$. This vectorized implementation will efficiently compute the weighted sums of the inputs plus the bias term for each and every neuron in the layer, for all the instances in the batch in just one shot.
- Finally, if the `activation` parameter is set to "relu", the code returns `relu(z)` (ie. `max(0, z)`), or else it just returns `z`.

Ok, so now you have a nice function to create a neuron layer. Let's use it to create the Deep Neural Network! The first hidden layer takes `X` as its input. The second takes the output of the first hidden layer as its input. And finally, the output layer takes the output of the second hidden layer as its input.

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

Notice that once again we used a name scope for clarity. Also note that `logits` is the output of the neural network *before* going through the softmax activation function: for optimization reasons, we will handle the softmax computation later.

⁹ Using a truncated normal distribution rather than a regular normal distribution ensures that there won't be any large weights, which could slow down training.

¹⁰ For example, if you set all the weights to zero, then all neurons will output zero, and the error gradient will be the same for all neurons in a given hidden layer. The Gradient Descent step will then update all the weights in exactly the same way in each layer, so they will all remain equal. In other words, despite having hundreds of neurons per layer, your model will act as if there were only one neuron per layer. It is not going to fly.

As you might expect, TensorFlow comes with many handy functions to create standard neural network layers: there's often no need to define your own `neuron_layer()` function like we just did. For example, TensorFlow's `fully_connected()` function creates a fully connected layer, where all the inputs are connected to all the neurons in the layer. It takes care of creating the `weights` and `biases` variables, with the proper initialization strategy, and it uses the ReLU activation function by default (this can be changed using the `activation_fn` argument). As we will see in [Chapter 11](#), it also supports regularization and normalization parameters. Let's tweak the code above to use the `fully_connected()` function instead of our `neuron_layer()` function. Simply import the function and replace the `dnn` construction section with the following code:

```
from tensorflow.contrib.layers import fully_connected

with tf.name_scope("dnn"):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                            activation_fn=None)
```



The `tensorflow.contrib` package contains many useful functions, but it is a place for experimental code that has not yet graduated to be part of the main TensorFlow API. So the `fully_connected()` function (and any other `contrib` code) may change or move in the future.

Now that we have the neural network model ready to go, we need to define the cost function that we will use to train it. Just like we did for softmax regression in [Chapter 4](#), we will use cross entropy. As we discussed earlier, cross entropy will penalize models that estimate a low probability for the target class. TensorFlow provides several functions to compute cross entropy. We will use `sparse_softmax_cross_entropy_with_logits()`: it computes the cross entropy based on the "logits" (ie. the output of the network *before* going through the softmax activation function), and it expects labels in the form of integers ranging from 0 to the number of classes minus 1 (in our case, from 0 to 9). This will give us a 1D tensor containing the cross entropy for each instance. We can then use TensorFlow's `reduce_mean()` function to compute the mean cross entropy over all instances.

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits, y)
    loss = tf.reduce_mean(xentropy, name="loss")
```



The `sparse_softmax_cross_entropy_with_logits()` function is equivalent to applying the softmax activation function then computing the cross entropy, but it is more efficient, and it properly takes care of corner cases like logits equal to 0. This is why we did not apply the softmax activation function earlier. There is also another function called `softmax_cross_entropy_with_logits()` which takes labels in the form of one-hot vectors (instead of ints from 0 to the number of classes minus 1).

We have the neural network model, we have the cost function, now we need to define a `GradientDescentOptimizer` that will tweak the model parameters to minimize the cost function. Nothing new, it's just like we did in [Chapter 9](#):

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

The last important step in the construction phase is to specify how to evaluate the model. We will simply use accuracy as our performance measure. First, for each instance determine if the neural network's prediction is correct by checking whether or not the highest logit corresponds to the target class. For this you can use the `in_top_k()` function. This returns a 1D tensor full of boolean values: we need to cast these booleans to floats and then compute the average. This will give us the network's overall accuracy.

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

And as usual, we need to create a node to initialize all variables, and we will also create a `Saver` to save our trained model parameters to disk:

```
init = tf.initialize_all_variables()
saver = tf.train.Saver()
```

Phew! This concludes the construction phase. This was less than 40 lines of code, but it was pretty intense: we created placeholders for the inputs and the targets, we created a function to build a neuron layer, we used it to create the DNN, we defined the cost function, we created an optimizer and finally we defined the performance measure. Now on to the execution phase.

Execution phase

This part is much shorter and simpler. First, let's load MNIST. We could use Scikit-Learn for that as we did in previous chapters, but TensorFlow offers its own helper

which fetches the data, scales it (between 0 and 1), shuffles it, and provides a simple function to load one mini-batches at a time. So let's use it instead:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

Now we define the number of epochs that we want to run, as well as the size of the mini-batches:

```
n_epochs = 400
n_batches = 100
```

And now we can train the model:

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(len(mnist.train.labels)//batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
            acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                              y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "my_model_final.ckpt")
```

This code opens a TensorFlow session, and it runs the `init` node which initializes all the variables. Then it runs the main training loop: at each epoch, the code iterates through a number of mini-batches that corresponds to the training set size. Each mini-batch is fetched using the `next_batch()` method, then the code simply runs the training operation, feeding it the current mini-batch input data and targets. Next, at the end of each epoch the code evaluates the model on the last mini-batch and on the full training set, and it prints out the result. Finally, the model parameters are saved to disk.

Using the neural network

Now that the neural network is trained, you can use it to make predictions. To do that, you can reuse the same construction phase, but change the execution phase like this:

```
with tf.Session() as sess:
    saver.restore(sess, "my_model_final.ckpt")
    X_new_scaled = [...] # some new images (scaled from 0 to 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

First the code loads the model parameters from disk. Then it loads some new images that you want to classify. Remember to apply the same feature scaling as for the training data (in this case, scale it from 0 to 1). Then the code evaluates the `logits` node.

If you wanted to know all the estimated class probabilities, you would need to apply the `softmax()` function to the logits, but if you just want to predict a class, you can simply pick the class that has the highest logit value (using the `argmax()` function does the trick).

Fine-tuning neural network hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable *network topology* (how neurons are interconnected), but even in a simple Feedforward Neural Networks you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weights initialization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

Of course, you can use grid search with cross-validation to find the right hyperparameters, like you did in previous chapters, but since training a neural network on a large dataset takes a lot of time, you will only be able to explore a small part of the hyperparameter space in a reasonable amount of time. One way to speed up the search is to start with a very coarse grid search, and then gradually zoom in on the part of the hyperparameter space that works best.

It helps to have an idea of what are reasonable values for each hyperparameter, so you can restrict the search space. Let's start with the number of hidden layers.

Number of hidden layers

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It was actually shown that an FNN with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time these facts convinced researchers that there was no need to investigate any deeper neural networks. But they overlooked the fact that deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially less neurons than shallow nets, making them much faster to train.

To understand why, suppose you are asked to draw a forest using some drawing software, but you are forbidden to use copy/paste. You would have to draw each tree individually, branch per branch, leaf per leaf. If you could instead draw one leaf, copy/paste it to draw a branch, then copy/paste that branch to create a tree, and finally copy/paste this tree to make a forest, you would be finished in no time. Real world data is often structured in such a hierarchical way and DNNs automatically take advantage of this fact: lower hidden layers model low-level structures (eg. line segments of various shapes and orientations), intermediate hidden layers combine

these low-level structures to model intermediate-level structures (eg. squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high level structures (eg. faces).

Not only does this hierarchical architecture help DNNs converge faster to a good solution, it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures, and you now want to train a new neural network to recognize hairstyles, then you can kickstart training by reusing the lower layers of the first network: instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the value of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures, it will only have to learn the higher-level structures (eg. hairstyles).

In summary, for many problems you can start with just one or two hidden layers and it will work just fine (eg. you can easily reach above 97% accuracy on the MNIST dataset using just 1 hidden layer with a few hundred neurons, and above 98% accuracy using 2 hidden layers with the same total amount of neurons, in roughly the same amount of training time). For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks such as large image classification or speech recognition typically require networks with dozens of layers (but not fully connected ones, as we will see in [???](#)), and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will be a lot faster and require much less data (we will discuss this in [Chapter 11](#)).

Number of neurons per hidden layer

A reasonable starting point for the number of neurons per hidden layer is to roughly interpolate the number of neurons between the input layer and the output layer. For example, in the MNIST task, there are 784 input neurons and 10 output neurons, so if you decide to have just one hidden layer, you could try filling it with 400 neurons (roughly the mean of 784 and 10), as a starting point. With 2 hidden layers, putting 300 neurons in the first layer and 100 neurons in the second layer is a reasonable choice as well. Just like for the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a black art.

A simpler approach is to pick a model with more layers and neurons than you actually need, then use early stopping to prevent it from overfitting (and other regularization techniques, such as *Dropout*, as we will see in [Chapter 11](#)). This has been

dubbed the “stretch pants” approach¹¹: instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

Activation functions

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants, as we will see in [Chapter 11](#)): it is much faster to compute than other activation functions, and Gradient Descent does not get stuck as much on plateaus, thanks to the fact that it does not saturate for large input values (as opposed to the logistic function or the hyperbolic tangent function that saturate at 1).

For the output layer, the softmax activation function is generally a good choice for classification tasks. For regression tasks, you can simply use no activation function at all.

Other popular ANN architectures

In the rest of this chapter, we will give a quick overview of a few popular neural network architectures. These architectures are much less used today than FNNs and the other architectures that we will study in the next chapters, but it can be helpful to know about them as they are often mentioned in the literature and some are still used in many applications. Moreover, Deep Belief Nets were the state-of-the-art in Deep Learning until the early 2010s. They are still the subject of very active research, so they may well come back with a vengeance in the near future.

Hopfield Nets

Hopfield nets were first introduced by W. A. Little in 1974, then popularized by J. Hopfield in 1982. They are *associative memory* networks: you first teach them some patterns, then when they see a new pattern they (hopefully) output the closest learned pattern. This has made them useful in particular for character recognition before they were outperformed by other approaches: you first train the network by showing it examples of character images (each binary pixel maps to one neuron), then when you show it a new character image, after a few iterations it outputs the closest learned character.

They are fully connected graphs (see [Figure 10-10](#)): every neuron is connected to every other neuron. Note that on the diagram the images are 6x6 pixels, so the neural network on the left should contain 36 neurons (and 648 connections) but for visual clarity a much smaller network is represented.

¹¹ By Vincent Vanhoucke in his Deep Learning class on Udacity.com.

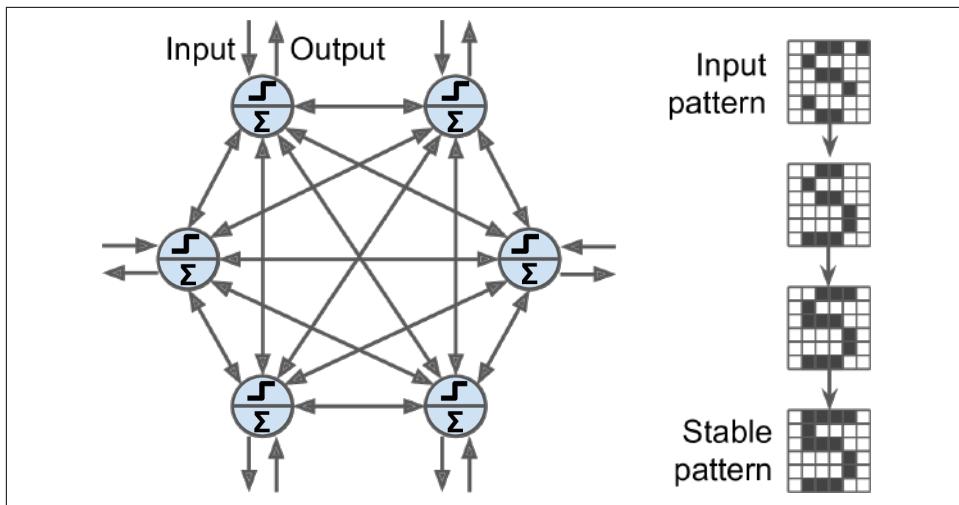


Figure 10-10. Recurrent Neural Network

The training algorithm works by using Hebb's rule: for each training image, the weight between two neurons is increased if the corresponding pixels are both on or both off, but it is decreased if one pixel is on and the other is off.

To show a new image to the network, you just activate the neurons that correspond to active pixels. The network then computes the output of every neuron, and this gives you a new image. You can then take this new image and repeat the whole process. After a while, the network reaches a stable state. Generally, this corresponds to the training image that most resembles the input image.

A so-called *energy function* is associated to Hopfield nets: it can be shown that at each iteration, the energy decreases, so the network is guaranteed to eventually stabilize to a low energy state. The training algorithm tweaks the weights in a way that decreases the energy level of the training patterns, so the network is likely to stabilize in one of these low energy configurations. Unfortunately, some patterns that were not in the training set also end up with low energy, so the network sometimes stabilizes in a configuration that was not learned. These are called *spurious patterns*.

Another major flaw with Hopfield Nets is that they don't scale very well: it was shown that their memory capacity is roughly equal to 14% of the number of neurons. For example, to classify 28x28 images, you would need a Hopfield net with 784 fully connected neurons and 306,936 weights. Such a network would only be able to learn about 110 different characters (14% of 784). That's a lot of parameters for such a small memory.

Boltzmann Machines

Boltzmann Machines were invented in 1985 by G. Hinton and T. Sejnowski. Just like Hopfield nets, they are fully-connected ANNs, but they are based on *stochastic neurons*: instead of using a deterministic step function to decide what value to output, these neurons output 1 with some probability, or 0 otherwise. The probability function that they use is based on the Boltzmann distribution (used in statistical mechanics) hence the name of these ANNs. [Equation 10-3](#) gives the probability that a particular neuron will output a 1.

Equation 10-3. Probability that the i^{th} neuron will output 1

$$P(s_i^{\text{(next step)}} = 1) = \sigma\left(\frac{\sum_{j=1}^N w_{i,j} s_j + b_i}{T}\right)$$

- s_j is the j^{th} neuron's state (0 or 1).
- $w_{i,j}$ is the connection weight between the i^{th} and j^{th} neurons. Note that $w_{i,i} = 0$.
- b_i is the i^{th} neuron's bias term. This term can be implemented by adding a bias neuron to the network.
- N is the number of neurons in the network.
- T is a number called the network's *temperature*: the higher the temperature, the more random the output is (ie. the more the probability approaches 50%).
- σ is the logistic function.

Neurons in Boltzmann Machines are separated into two groups: *visible units* and *hidden units* (see [Figure 10-11](#)). All neurons work in the same stochastic way, but the visible units are the ones that receive the inputs and from which outputs are read.

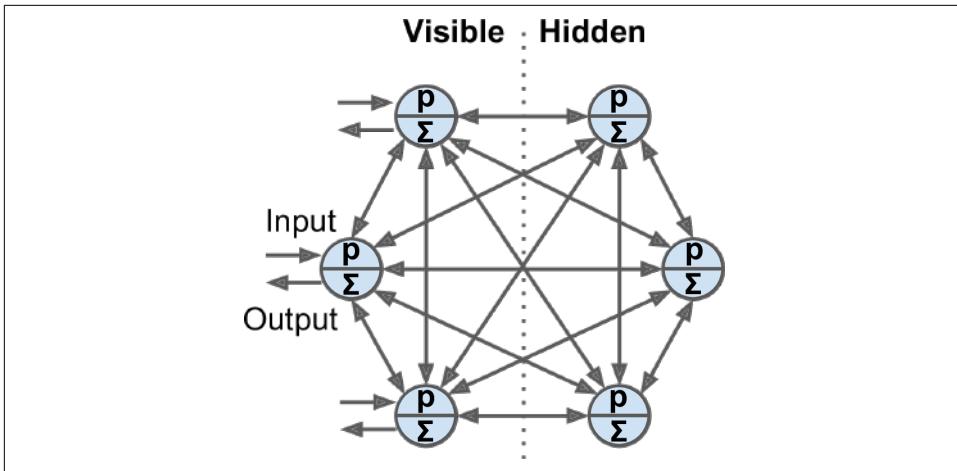


Figure 10-11. Boltzmann Machine

Because of its stochastic nature, a Boltzmann Machine will never stabilize into a fixed configuration, but instead it will keep switching between many configurations. It can be shown that if it is left running for a sufficiently long time, the probability of observing a particular configuration will only be a function of the connection weights and bias terms, not of the original configuration (similarly, after shuffling a deck of cards for long enough, the configuration of the deck does not depend on the initial state). When the network reaches this state where the original configuration is “forgotten”, it is said to be in *thermal equilibrium* (although its configuration keeps changing all the time). By setting the network parameters appropriately, letting the network reach thermal equilibrium then observing its state, it is possible to simulate a wide range of probability distributions. This is called a *generative model*.

Training a Boltzmann Machine means finding the parameters that will make the network approximate the training set’s probability distribution. For example, if there are 3 visible neurons and the training set contains 75% of (0, 1, 1) triplets, 10% of (0, 0, 1) triplets, and 15% of (1, 1, 1) triplets, then after training a Boltzmann Machine, you could use it to generate random binary triplets with about the same probability distribution. For example, about 75% of the time it would output the (0, 1, 1) triplet.

Such a generative model can be used in a variety of ways. For example, if it is trained on images, and you provide an incomplete or noisy image to the network, it will automatically “repair” the image in a reasonable way. You can also use a generative model for classification: just add a few visible neurons to encode the training image’s class (eg. add 10 visible neurons and turn on only the 5th neuron when the training image represents a 5), then when given a new image, the network will automatically turn on the appropriate visible neurons, indicating the image’s class (eg. it will turn on the 5th visible neuron if the image represents a 5).

Unfortunately, there is no efficient technique to train Boltzmann Machines. However, fairly efficient algorithms have been developed to train *Restricted Boltzmann Machines* (RBM).

Restricted Boltzmann Machines

An RBM is simply a Boltzmann Machine in which there are no connections between visible units or between hidden units, only between visible and hidden units. For example, Figure 10-12 represents an RBM with 3 visible units and 4 hidden units.

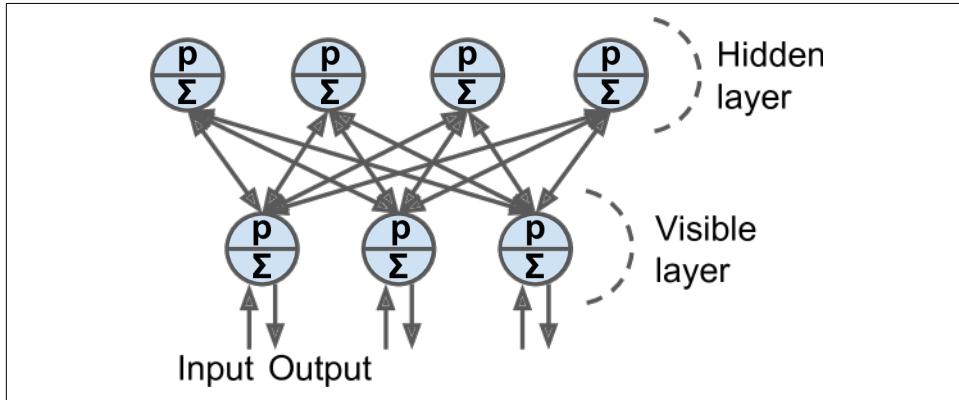


Figure 10-12. Restricted Boltzmann Machine

A very efficient training algorithm, called *Contrastive Divergence*, was introduced in 2005 by M. Á. Carreira-Perpiñán and G. Hinton¹². Here is how it works: for each training instance \mathbf{x} , the algorithm starts by feeding it to the network by setting the state of the visible units to x_1, x_2, \dots, x_n . Then you compute the state of the hidden units, by applying the stochastic equation described above (Equation 10-3). This gives you a hidden vector \mathbf{h} (where h_i is equal to the state of the i^{th} unit). Next you compute the state of the visible units, by applying the same stochastic equation. This gives you a vector $\dot{\mathbf{x}}$. Then once again you compute the state of the hidden units, which gives you a vector $\dot{\mathbf{h}}$. Now you can update each connection weight by applying the following rule:

Equation 10-4. Contrastive divergence weight update

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta (\mathbf{x}\mathbf{h}^T - \dot{\mathbf{x}}\dot{\mathbf{h}}^T)$$

¹² "On Contrastive Divergence Learning", M. Á. Carreira-Perpiñán and G. Hinton (2005): <http://goo.gl/ZCP6Ir>

The great benefit of this algorithm is that it does not require waiting for the network to reach thermal equilibrium: it just goes forward, backward and forward again, and that's it. This makes it incomparably more efficient than previous algorithms, and it was a key ingredient to the first success of Deep Learning based on multiple stacked RBMs.

Deep Belief Nets

Several layers of RBMs can be stacked: the hidden units of the first level RBM serve as the visible units for the second layer RBM, and so on. Such an RBM stack is called a *Deep Belief Net* (DBN).

Yee-Whye Teh, one of Geoffrey Hinton's students, observed that it was possible to train DBNs one layer at a time using Contrastive Divergence, starting with the lower layers then gradually moving up to the top layers. This led to the groundbreaking article that kickstarted the Deep Learning tsunami in 2006: "A fast learning algorithm for deep belief nets"¹³.

Just like RBMs, DBNs learn to reproduce the probability distribution of their inputs, without any supervision. However, they are much better at it, for the same reason that deep FNNs are more powerful than shallow ones: real world data is often organized in hierarchical patterns, and DBNs take advantage of that: their lower layers learn low-level features in the input data, while higher layers learn high-level features.

Just like RBMs, DBNs are fundamentally unsupervised, but they can also be trained in a supervised manner by adding some visible units to represent the labels. Moreover, one great feature of DBNs is that they can be trained in a semi-supervised fashion. [Figure 10-13](#) represents such a DBN configured for semi-supervised learning.

¹³ G. Hinton, S. Osindero, Y. Teh (2006): <http://goo.gl/BcZQrH>

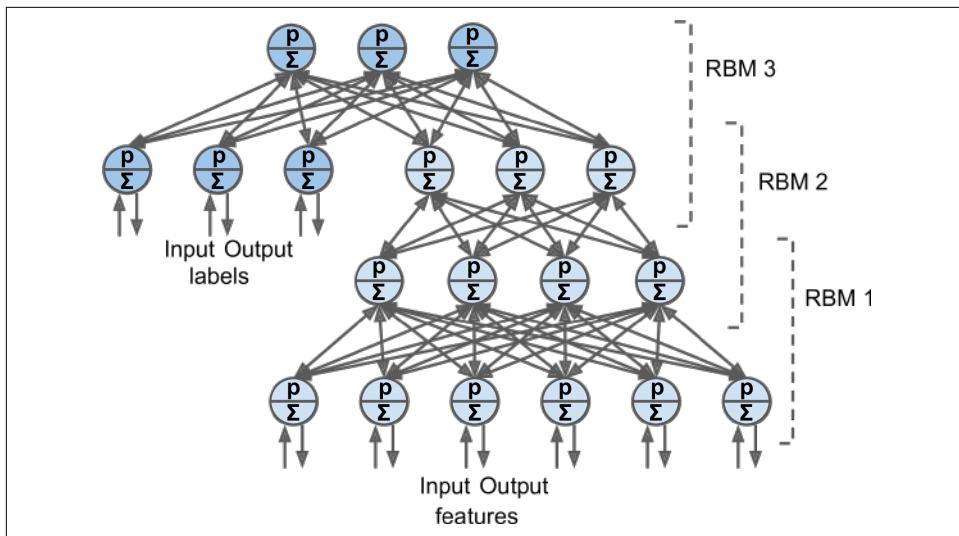


Figure 10-13. A Deep Belief Network configured for semi-supervised learning

First, the RBM 1 is trained without supervision. It learns low-level features in the training data. Then RBM 2 is trained with RBM 1's hidden units as inputs, again without supervision: it learns higher-level features (note that RBM 2's hidden units only include the three right most units, not the label units). Several more RBMs could be stacked this way, but you get the idea. So far, training was 100% unsupervised. Lastly, RBM 3 is trained using both RBM 2's hidden units as inputs, as well as extra visible units used to represent the target labels (eg. a one-hot vector representing the instance class). It learns to associate high-level features with training labels. This is the supervised step.

At the end of training, if you feed RBM 1 a new instance, the signal will propagate up to RBM 2, then up to the top of RBM 3, then back down to the label units, and hopefully the appropriate label will light up. This is how a DBN can be used for classification.

One great benefit of this semi-supervised approach is that you don't need much labeled training data: if the unsupervised RBMs do a good enough job, then only a small amount of labeled training instances per class will be necessary. Similarly, a baby learns to recognize objects without supervision, so when you point to a chair and say "chair", the baby can associate the word "chair" to the class of objects it has already learned to recognize on its own. You don't need to point to every single chair and say "chair", only a few examples will suffice (just enough so the baby can be sure that you are indeed referring to the chair, not to its color or one of its parts).

Quite amazingly, DBNs can also work in reverse: if you activate one of the label units, the signal will propagate up to the hidden units of RBM 3, then down to RBM 2, then

RBM 1, and a new instance will be output by the visible units of RBM 1. This new instance will usually look like a regular instance of the class whose label unit you activated. This generative capability of DBNs is quite powerful. For example, it has been used to automatically generate captions for images, and *vice versa*: first a DBN is trained (without supervision) to learn features in images, and another DBN is trained (again without supervision) to learn features in sets of captions (eg. “car” often comes with “automobile”). Then an RBM is stacked on top of both DBNs and it is trained with a set of images along with their captions: it learns to associate high-level features in images with high-level features in captions. Next, if you feed the image DBN an image of a car, the signal will propagate through the network, up to the top-level RBM and back down to the bottom of the caption DBN, producing a caption. Due to the stochastic nature of RBMs and DBNs, the caption will keep changing randomly, but it will generally be appropriate for the image. If you generate a few hundred captions, the most frequently generated ones will likely be a good description of the image.¹⁴

Self-Organizing Maps

Self-Organizing Maps (SOM) are quite different from all the other types of neural networks we have discussed so far. They are used to produce a low-dimensional representation of a high-dimensional dataset, generally for visualization, clustering or classification. The neurons are spread across a map (typically 2D for visualization, but it can be any number of dimensions you want), as shown on [Figure 10-14](#), and each neuron has a weighted connection to every input (note that the diagram shows just 2 inputs, but there are typically a very large number, since the whole point of SOMs is to reduce dimensionality).

¹⁴ See this video by G. Hinton for more details and a demo: <http://goo.gl/7Z5QiS>

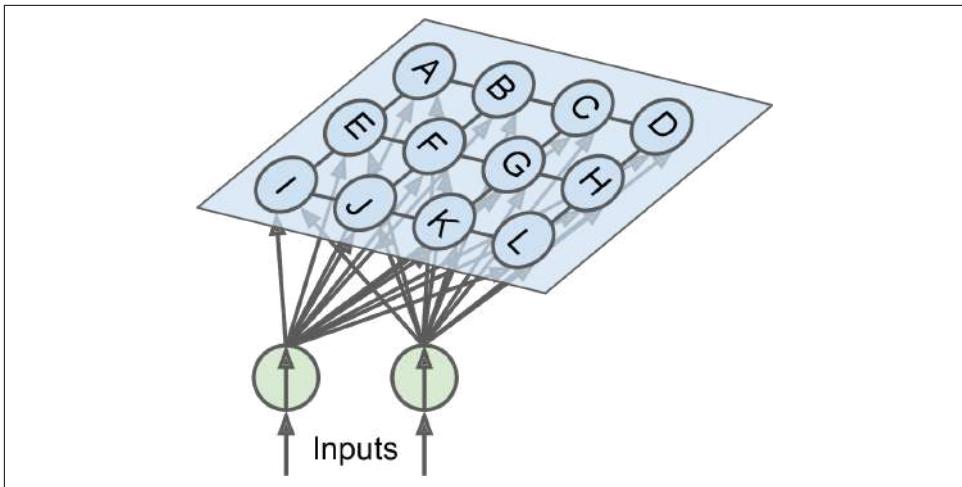


Figure 10-14. Self-Organizing Maps

Once the network is trained, you can feed it a new instance and this will activate only one neuron (ie. hence one point on the map): it is the neuron whose weight vector is closest to the input vector. In general, instances that are nearby in the original input space will activate neurons that are nearby on the map. This makes SOMs useful for visualization (in particular you can easily identify clusters on the map), but also for applications like speech recognition: for example, if each instance represents the audio recording of a person pronouncing a vowel, then different pronunciations of the vowel “a” will activate neurons in the same area of the map, while instances of the vowel “e” will activate neurons in another area, and intermediate sounds will generally activate intermediate neurons on the map.



One important difference with the other dimensionality reduction techniques discussed in [Chapter 8](#) is that all instances get mapped to a discrete number of points in the low-dimensional space (one point per neuron). When there are very few neurons, then this technique is better described as clustering rather than dimensionality reduction.

The training algorithm is unsupervised. It works by having all the neurons compete against each other. First, all the weights are initialized randomly. Then a training instance is picked randomly and fed to the network. All neurons compute the distance between their weight vector and the input vector (this is very different from the artificial neurons we have seen so far). The neuron that measures the smallest distance wins and it tweaks its weight vector to be even slightly closer to the input vector, making it more likely to win future competitions for other inputs similar to this one. It also recruits its neighboring neurons and they too update their weight vector

to be slightly closer to the input vector (but their don't update their weights as much as the winner neuron). Then the algorithm picks another training instance and repeats the process, again and again. This algorithm tends to make nearby neurons gradually specialize in similar inputs.¹⁵

This concludes this introduction to Artificial Neural Networks. In the following chapters, we will discuss techniques to train DNNs, distribute training across multiple servers and GPUs, then we will explore a few other popular neural network architectures: Convolutional Neural Networks, Recurrent Neural Networks and Autoencoders.

Exercises

1. Draw an ANN using the original artificial neurons (like the ones in [Figure 10-3](#)) that computes $A \oplus B$ (where \oplus represents the XOR operation). Hint: $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$.
2. Why is it generally preferable to use a Logistic Regression classifier rather than a Perceptron?
3. Why was the sigmoid activation a key ingredient in training the first MLPs?
4. Name 3 popular activation functions? Can you draw them?
5. Suppose you have an FNN composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
 - What is the shape of the input matrix \mathbf{X} ?
 - What about the shape of the hidden layer's weight vector \mathbf{W}_h , and the shape of its bias vector \mathbf{b}_h ?
 - What is the shape of the output layer's weight vector \mathbf{W}_o , and its bias vector \mathbf{b}_o ?
 - What is the shape of the network's output matrix \mathbf{Y} ?
 - Write the equation that computes the network's output matrix \mathbf{Y} as a function of \mathbf{X} , \mathbf{W}_h , \mathbf{b}_h , \mathbf{W}_o and \mathbf{b}_o .
6. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer?

¹⁵ You can imagine a class of young children with roughly similar skills: one child happens to be slightly better at basketball. This motivates her to practice more, especially with her friends. After a while, this group of friends gets so good at basketball that other kids cannot compete. But that's ok, the other kids specialize in other topics. After a while, the class is full of little specialized groups.

If instead you want to tackle MNIST, how many neurons do you need in the output layer, using what activation function? Same questions if you want your network to predict housing prices like in [Chapter 2](#)?

7. How is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
8. Can you list all the hyperparameters you can tweak in a plain DNN? If the DNN overfits the training data, how could you tweak these hyperparameters to try to solve the problem?
9. What is a generative model? Can you name some types of ANNs that are generative models and some that are not?
10. Train a DNN on the MNIST dataset and see if you can get over 98% precision. Just like in the last exercise of [Chapter 9](#), try adding all the bells and whistles (ie. save checkpoints, restore the last checkpoint in case of an interruption, add summaries, plot learning curves using TensorBoard, and so on).

Deep Learning

In [Chapter 10](#) we introduced Artificial Neural Networks and trained our first Deep Neural Network. But it was a very shallow DNN, with only two hidden layers. What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, perhaps with (say) ten layers, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be a walk in the park:

- First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- Second, with such a large network, training would be extremely slow.
- Third, a model with millions of parameters would severely risk overfitting the training set.

In this chapter, we will go through each of these problems in turn and present techniques to solve them. We will start by explaining the vanishing gradients problem and exploring some of the most popular solutions to this problem. Next we will look at various optimizers that can speed up training large models tremendously compared to plain Gradient Descent. Finally, we will go through a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets: welcome to Deep Learning!

Vanishing/exploding gradients problems

As we discussed in [Chapter 10](#), the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each

parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger, so the lower layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem. More generally, deep neural networks suffer from unstable gradients: different layers may learn at widely different speeds.

Although this unfortunate behavior has been empirically observed for quite a while (it is one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it: a paper titled “Understanding the difficulty of training deep feedforward neural networks” by Xavier Glorot and Y. Bengio¹ found a few suspects, including the combination of the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using a Normal distribution with mean 0 and standard deviation of 1. In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

Looking at the logistic activation function (see [Figure 11-1](#)), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

¹ <http://goo.gl/1rhAef>

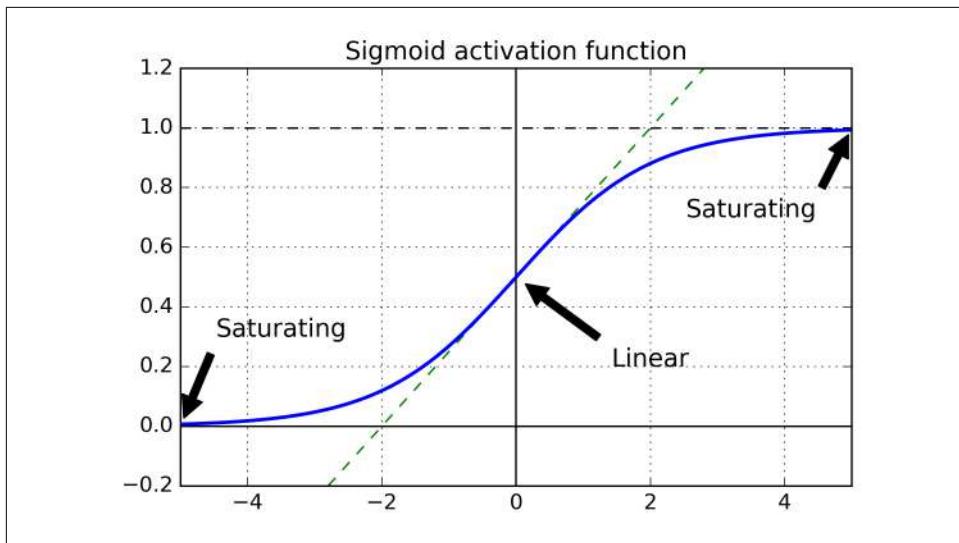


Figure 11-1. Sigmoid activation function saturation

Xavier and He initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. For this to happen, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of input and output connections, but they proposed a good compromise that has proven to work very well in practice: the connection weights must be initialized randomly as described in [Equation 11-1](#), where n_{inputs} and n_{outputs} are the number of input and output connections for the layer whose weights are being initialized (also called *fan-in* and *fan-out*). This initialization strategy is often called *Xavier initialization* (after the author's first name), or sometimes *Glorot initialization*.

Equation 11-1. Xavier initialization (when using the logistic activation function)

$$\text{Normal distribution with mean 0 and standard deviation } \sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

$$\text{Or a uniform distribution between } -r \text{ and } +r, \text{ with } r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

When the number of input connections is roughly equal to the number of output connections, you get simpler equations (eg. $\sigma = 1/\sqrt{n_{\text{inputs}}}$ or $r = \sqrt{3}/\sqrt{n_{\text{inputs}}}$). We used this simplified strategy in [Chapter 10](#)².

Using the Xavier initialization strategy can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning. Some recent papers³ have provided similar strategies for different activation functions, as shown in [Table 11-1](#). The initialization strategy for the ReLU activation function (and its variants, including the ELU activation described below) is sometimes called *He initialization* (after the last name of its author).

Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution $[-r, r]$	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

By default, the `fully_connected()` function (introduced in [Chapter 10](#)) uses Xavier initialization (with a uniform distribution). You can change this to He initialization by using the `variance_scaling_initializer()` function like this:

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = fully_connected(X, n_hidden1, weights_initializer=he_init, scope="h1")
```



He initialization only considers the fan-in, not the average between fan-in and fan-out like in Xavier initialization. This is also the default for the `variance_scaling_initializer()` function, but you can change this by setting the argument `mode="FAN_AVG"`.

Non-saturating activation functions

One of the insights in the 2010 paper by X. Glorot and Y. Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to

² This simplified strategy was actually already proposed much earlier, eg. in the 1998 book “Neural Networks, tricks of the trade” by Genevieve Orr and Klaus-Robert Müller.

³ Such as “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, Kaiming He *et al.*: <http://goo.gl/VHP3pB>

use roughly sigmoid activation functions in biological neurons, it must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is very fast to compute).

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively die, meaning they stop outputting anything else than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated in a way that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happens, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, you may want to use a variant of the ReLU function, such as the *Leaky ReLU*. This function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see [Figure 11-2](#)). The hyperparameter α defines how much the function “leaks”: it is the slope of the function for $z < 0$. It is typically set to 0.01. This small slope ensures that Leaky ReLUs never die: they can go into a long coma, but they have a chance to eventually wake up. A recent paper⁴ compared several variants of the ReLU activation function and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting $\alpha = 0.2$ (huge leak) seemed to result in better performance than $\alpha = 0.01$ (small leak). They also evaluated the *Randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training, and it is fixed to an average value during testing: it also performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set). Finally, they also evaluated the *Parametric leaky ReLU* (PReLU), where α is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter which can be modified by backpropagation like any other parameter). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

⁴ “Empirical Evaluation of Rectified Activations in Convolution Network”, Bing Xu *et al.* (2015): <https://goo.gl/B1xhKn>

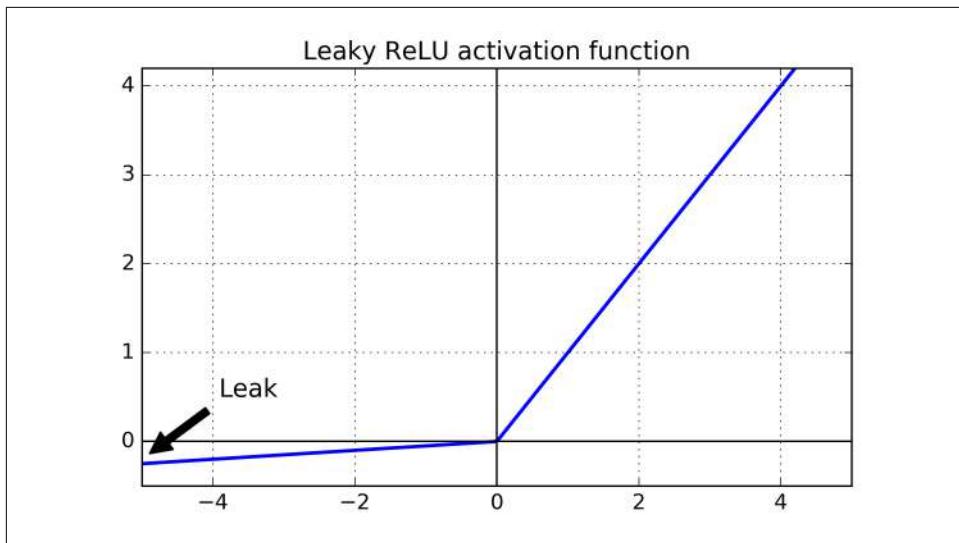


Figure 11-2. Leaky ReLU

Last but not least, a 2015 paper by Djork-Arné Clevert *et al.*⁵ proposed a new activation function called the *Exponential Linear Unit* (ELU) which outperformed all the ReLU variants in their experiments: training time was reduced and the neural network performed better on the test set. It is represented in Figure 11-3 and Equation 11-2 shows its definition.

Equation 11-2. ELU activation function

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

⁵ “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”, D. Clevert, T. Unterthiner and S. Hochreiter (2015): <http://goo.gl/SdI2P7>

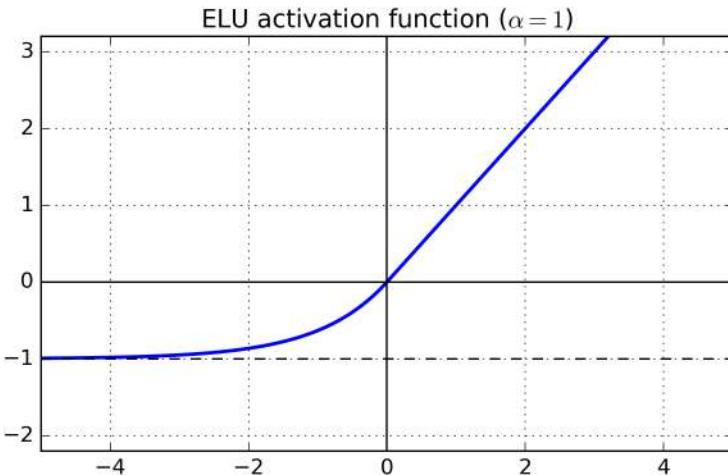


Figure 11-3. ELU activation function

It looks a lot like the ReLU function, with a few major differences:

- First it takes on negative values when $z < 0$ which allows the unit to have an average output closer to 0. This helps alleviate the vanishing gradients problem, as discussed above. The hyperparameter α defines the value that the ELU function approaches when z is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter if you want.
- Second, it has a non-zero gradient for $z < 0$, which avoids the dying units issue.
- Third, the function is smooth everywhere, including around $z = 0$, which helps speed up Gradient Descent, since it does not bounce as much left and right of $z = 0$.

The main drawback of the ELU activation function is that it is slower to compute than the ReLU and its variants (due to the use of the exponential function), but during training this is compensated by the faster convergence rate. However, at test time an ELU network will be slower than a ReLU network.



So which activation function should you use for Deep Neural Networks? Although your mileage will vary, in general ELU > Leaky ReLU (and its variants) > ReLU > tanh > logistic. If you care a lot about runtime performance, then you may prefer Leaky ReLUs over ELUs. If you don't want to tweak yet another hyperparameter, you may just use the default α values suggested above (0.01 for the Leaky ReLU, and 1 for ELU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular RReLU if your network is overfitting, or PReLU if you have a huge training set.

TensorFlow offers an `elu()` function that you can use to build your neural network. Simply set the `activation_fn` argument when calling the `fully_connected()` function, like this:

```
hidden1 = fully_connected(X, n_hidden1, activation_fn=tf.nn.elu)
```

TensorFlow does not have a predefined function for Leaky ReLUs, but it is easy enough to define:

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_relu)
```

Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce the vanishing/exploding gradients problems at the beginning of training, they don't guarantee that they won't come back during training.

In a 2015 paper⁶, Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change (which they call the *Internal Covariate Shift* problem).

The technique consists in adding an operation in the model, just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer.

⁶ "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", S. Ioffe and C. Szegedy (2015): <https://goo.gl/gA4GSP>

In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs over the current mini-batch (hence the name "Batch Normalization"). The whole operation is summarized in [Equation 11-3](#).

Equation 11-3. Batch Normalization algorithm

$$\begin{aligned} 1. \quad \mu_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \\ 2. \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2 \\ 3. \quad \hat{\mathbf{x}}^{(i)} &= \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ 4. \quad \mathbf{z}^{(i)} &= \gamma \hat{\mathbf{x}}^{(i)} + \beta \end{aligned}$$

- μ_B is the empirical mean, evaluated over the whole mini-batch B .
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.
- $\hat{\mathbf{x}}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

At test time, there is no mini-batch to compute the empirical mean and standard deviation, so instead you simply use the whole training set's mean and standard deviation. These are typically efficiently computed during training using a moving average. So in total, four parameters are learned for each batch-normalized layer: γ (scale), β (offset), μ (mean) and σ (standard deviation).

The authors demonstrated that this technique considerably improved all the Deep Neural Networks they experimented with. The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the logistic activation function. The networks were also much less sensitive to the weights initialization. They were able to use much larger learning

rates, significantly speeding up the learning process. Specifically, they note that "*Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. [...] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.*" Finally, like a gift that keeps on giving, Batch Normalization also acts like a regularizer, reducing the need for other regularization techniques (such as Dropout, described below).

Batch Normalization does however add some complexity to the model (although it removes the need for normalizing the input data since the first hidden layer will take care of that, provided it is batch-normalized). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. So if you need predictions to be lightning fast, you may want to check how well plain ELU + He initialization perform before playing with Batch Normalization.



You may observe that training is rather slow at first while Gradient Descent is searching for the optimal scales and offsets for each layer, but then it accelerates once it has found reasonably good values.

Implementing Batch Normalization with TensorFlow

TensorFlow provides a `batch_normalization()` function that simply centers and normalizes the inputs, but you must compute the mean and standard deviation yourself (based on the mini-batch data during training or on the full dataset during testing, as discussed above) and pass them as parameters to this function, and you must also handle the creation of the scaling and offset parameters (and pass them to this function). It is doable, but not the most convenient approach. Instead, you should use the `batch_norm()` function which handles all this for you. You can either call it directly or tell the `fully_connected()` function to use it, such as in the following code:

```
import tensorflow as tf
from tensorflow.contrib.layers import batch_norm

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

is_training = tf.placeholder(tf.bool, shape=(), name='is_training')
bn_params = {
```

```

    'is_training': is_training,
    'decay': 0.99,
    'updates_collections': None
}

hidden1 = fully_connected(X, n_hidden[0], scope="hidden1",
                         normalizer_fn=batch_norm, normalizer_params=bn_params)
hidden2 = fully_connected(hidden1, n_hidden[1], scope="hidden2",
                         normalizer_fn=batch_norm, normalizer_params=bn_params)
logits = fully_connected(hidden2, n_outputs, activation_fn=None, scope="outputs",
                         normalizer_fn=batch_norm, normalizer_params=bn_params)

```

Let's walk through this code. The first lines are fairly self-explanatory, until we define the `is_training` placeholder, which will either be `True` or `False`. This will be used to tell the `batch_norm()` function whether it should use the current mini-batch's mean and standard deviation (during training) or the running averages that it keeps track of (during testing).

Next we define `bn_params` which is a dictionary that defines the parameters that will be passed to the `batch_norm()` function, including `is_training` of course. The algorithm uses *exponential decay* to compute the running averages, which is why it requires the `decay` parameters. Given a new value v , the running average \hat{v} is updated using the equation: $\hat{v} \leftarrow \hat{v} \times \text{decay} + v \times (1 - \text{decay})$. A good decay value is typically close to 1, for example 0.9, 0.99, or 0.999 (you want more 9s for larger datasets and smaller mini-batches). Finally, `updates_collections` should be set to `None` if you want the `batch_norm()` function to update the running averages right before it performs batch normalization during training (ie. when `is_training=True`). If you don't set this parameter, by default TensorFlow will just add the operations that update the running averages to a collection of operations that you must run yourself.

Lastly, we create the layers by calling the `fully_connected()` function, just like we did in [Chapter 10](#), but this time we tell it to use the `batch_norm()` function (with the parameters `nb_params`) to normalize the inputs right before calling the activation function.

Note that by default `batch_norm()` only centers, normalizes and shifts the inputs, it does not scale them (ie. γ is fixed to 1). This makes sense for layers with no activation function or with the ReLU activation function, since the next layer's weights can take care of scaling, but for any other activation function, you should add "scale": `True` to `bn_params`.

You may have noticed that defining the three layers above was fairly repetitive since several parameters were identical. To avoid repeating the same parameters over and over again, you can create an *argument scope* using the `arg_scope()` function: the first parameter is a list of functions, and the other parameters will be passed to these functions automatically). The last 3 lines of the code above can be modified like so:

```
[...]
```

```
with tf.contrib.framework.arg_scope(
    [fully_connected],
    normalizer_fn=batch_norm,
    normalizer_params=bn_params):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

It may not look much better than before in this small example, but if you have 10 layers and you want to set the activation function, the initializers, the normalizers, the regularizers and so on, it will make your code much more readable.

The rest of the construction phase is the same as in [Chapter 10](#): define the cost function, create an optimizer, tell it to minimize the cost function, define the evaluation operations, create a saver, etc.

The execution phase is also pretty much the same, with one exception: whenever you run an operation that depends on the `batch_norm` layer, you need to set the `is_training` placeholder to `True` or `False`:

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op,
                     feed_dict={is_training: True, X: X_batch, y: y_batch})
        accuracy_score = accuracy.eval(
            feed_dict={is_training: False, X: X_test_scaled, y: y_test}))
        print(accuracy_score)
```

That's all! In this tiny example with just two layers, it's unlikely that Batch Normalization will have a very positive impact, but for deeper networks it can make a tremendous difference.

Gradient clipping

A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold. This is called *Gradient clipping*⁷. In general people now prefer Batch Normalization, but it's still useful to know about Gradient clipping and how to implement it.

⁷ “On the difficulty of training recurrent neural networks”, R. Pascanu *et al.* (2013): <http://goo.gl/dRDAaf>

In TensorFlow, the optimizer's `minimize()` function takes care of both computing the gradients and applying them, so you must instead call the optimizer's `compute_gradients()` method first, then create an operation to clip the gradients using the `clip_by_value()` function, and finally create an operation to apply the clipped gradients using the optimizer's `apply_gradients()` method:

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

You would then run this `training_op` at every training step, as usual. It will compute the gradients, clip them between -1.0 and 1.0 and apply them. The threshold is a hyperparameter you can tune.

Reusing pretrained layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle, then just reuse the lower layers of this network: this is called *transfer learning*. It will not only speed up training considerably, it will also require much less training data.

For example, suppose that you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and every day objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, so you should try to reuse parts of the first network (see Figure 11-4).

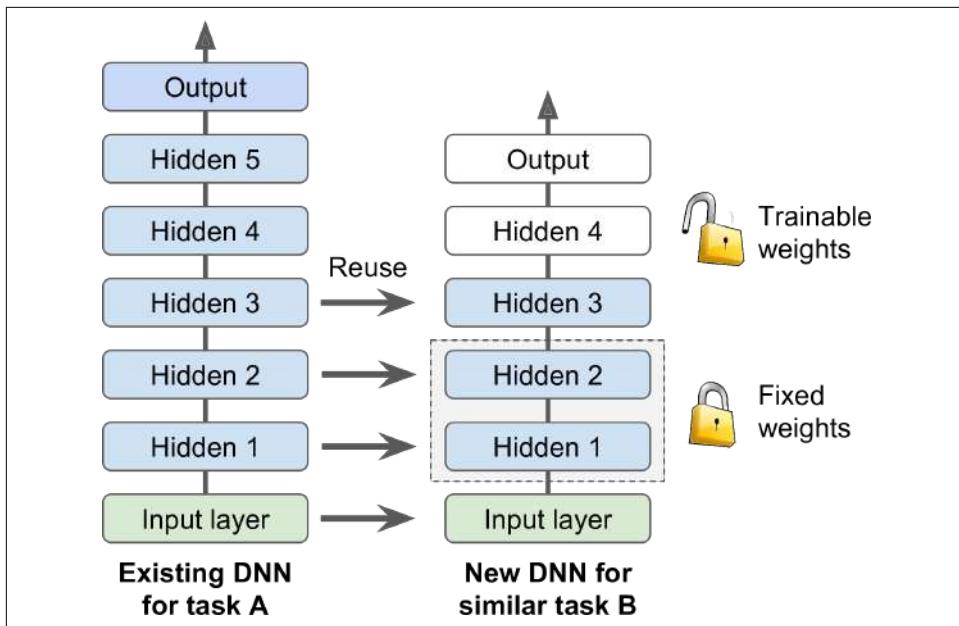


Figure 11-4. Reusing pretrained layers



If the input pictures of your new task don't have the same size as the ones used in the original task, you will have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will only work well if the inputs have similar low level features.

Reusing a TensorFlow model

If the original model was trained using TensorFlow, you can simply restore it and train it on the new task:

```
[...] # construct the original model

with tf.Session() as sess:
    saver.restore(sess, "my_original_model.ckpt")
    [...] # Train it on your new task
```

However, in general you will want to reuse only part of the original model (as we will discuss below). A simple solution is to configure the saver to restore only a subset of the variables from the original model. For example, the following code restores only hidden layers 1, 2 and 3:

```
[...] # build new model with the same definition as before for hidden layers 1-3

init = tf.initialize_all_variables()
```

```

reuse_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[123]")
reuse_vars_dict = dict([(var.name, var.name) for var in reuse_vars])
original_saver = tf.Saver(reuse_vars_dict) # saver to restore the original model

new_saver = tf.Saver() # saver to save the new model

with tf.Session() as sess:
    sess.run(init)
    original_saver.restore("my_original_model.ckpt") # restore layers 1 to 3
    [...] # train the new model
    new_saver.save("my_new_model.ckpt") # save the whole model

```

First we build the new model, making sure to copy the original model's hidden layers 1 to 3. We also create a node to initialize all variables. Then we get the list of all variables that were just created with "trainable=True" (which is the default), and we keep only the ones whose scope matches the regular expression "hidden[123]" (ie. we get all trainable variables in hidden layers 1 to 3). Next we create a dictionary mapping the name of each of these variables in the original model to its name in the new model (generally you want to just keep the exact same names). Then we create a saver that will only restore these variables, and we create another saver to save the entire new model, not just layers 1 to 3. We then start a session and initialize all variables in the model, then we restore the variable values from the original model's layers 1 to 3. Finally, we then train the model on the new task and we save it.



The more similar the tasks are, the more layers you want to reuse (starting with the lower layers). For very similar tasks, you can try keeping all the hidden layers and just replace the output layer.

Reusing models from other frameworks

If the model was trained using another framework, you will need to load the weights manually (eg. using Theano code if it was trained with Theano), then assign them to the appropriate variables. This can be quite tedious. For example, the following code shows how you would copy the weight and biases from the first hidden layer of a model trained using another framework:

```

original_w = [...] # Load the weights from the other framework
original_b = [...] # Load the biases from the other framework

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
[...] # Build the rest of the model

# Get a handle on the variables created by fully_connected()

```

```

with tf.variable_scope("", reuse=True):
    hidden1_weights = tf.get_variable("hidden1/weights")
    hidden1_biases = tf.get_variable("hidden1/biases")

    # Create nodes to assign arbitrary values to the weights and biases
    original_weights = tf.placeholder(tf.float32, shape=(n_inputs, n_hidden1))
    original_biases = tf.placeholder(tf.float32, shape=(n_hidden1))
    assign_hidden1_weights = tf.assign(hidden1_weights, original_weights)
    assign_hidden1_biases = tf.assign(hidden1_biases, original_biases)

init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)
    sess.run(assign_hidden1_weights, feed_dict={original_weights: original_w})
    sess.run(assign_hidden1_biases, feed_dict={original_biases: original_b})
    [...] # Train the model on your new task

```

Freezing the lower layers

It is likely that the lower layers of the first DNN have learned to detect low-level features in pictures that will be useful across both image classification tasks, so you can just reuse these layers as they are. It is generally a good idea to “lock” their weights when training the new DNN: if the lower layer weights are fixed, then the higher layer weights will be easier to train (because they won’t have to learn a moving target). To freeze the lower layers during training, the simplest solution is to give the optimizer the list of variables to train, excluding the variables from the lower layers:

```

train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                             scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)

```

The first line gets the list of all trainable variables in hidden layers 3 and 4 and in the output layer. This leaves out the variables in the hidden layers 1 and 2. Next we provide this restricted list of trainable variables to the optimizer’s `minimize()` function. Ta-da! Layers 1 and 2 are now locked: they will not budge during training (these are often called *frozen layers*).

Caching the frozen layers

Since the frozen layers won’t change, it is possible to cache the output of the top-most frozen layer for each training instance. Since training goes through the whole dataset many times, this will give you a huge speed boost as you will only need to go through the frozen layers once per training instance (instead of once per epoch). For example, you could first run the whole training set through the lower layers (assuming you have enough RAM):

```
hidden2_outputs = sess.run(hidden2, feed_dict={X: X_train})
```

Then during training instead of building batches of training instances, you would build batches of outputs from hidden layer 2 and feed them to the training operation:

```
n_epochs = 100
n_batches = 500

for epoch in range(n_epochs):
    shuffled_idx = rnd.permutation(len(hidden2_outputs))
    hidden2_batches = np.array_split(hidden2_outputs[shuffled_idx], n_batches)
    y_batches = np.array_split(y_train[shuffled_idx], n_batches)
    for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
        sess.run(training_op, feed_dict={hidden2: hidden2_batch, y: y_batch})
```

The last line runs the training operation defined earlier (which freezes layers 1 and 2), and it feeds it a batch of outputs from the second hidden layer (as well as the targets for that batch). Since we give TensorFlow the output of hidden layer 2, it does not try to evaluate it (or any node it depends on).

Tweaking, dropping or replacing the upper layers

The output layer of the original model should usually be replaced since it is mostly likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers since the high level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.

Try freezing all the copied layers first, train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freeze all remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even add more hidden layers.

Model zoos

Where can you find a neural network trained for a task similar to the one you want to tackle? The first place to look is obviously in your own catalog of models: this is one good reason to save all your models and organize them so you can retrieve them later easily. Another option is to search in a *model zoo*: many people train Machine Learning models for various tasks and kindly release their pretrained models to the public.

TensorFlow has its own model zoo available at <https://github.com/tensorflow/models>. In particular, it contains most of the state-of-the-art image classification nets such as VGG, Inception and ResNet (see ???, check out the `models/slim` directory), including the code, the pretrained models and tools to download popular image datasets.

Another popular model zoo is Caffe's Model-Zoo⁸. It also contains many computer vision models (eg. LeNet, AlexNet, ZFNet, GoogleNet, VGGNet, inception) trained on various datasets (eg. ImageNet, Places Database, CIFAR10, etc.). Saumitro Dasgupta wrote a converter available at <https://github.com/ethereon/caffe-tensorflow>.

Unsupervised pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task... don't lose all hope! First, you should of course try to gather more labeled training data, but if this is too hard or too expensive, you may still be able to perform *unsupervised pretraining* (see Figure 11-5): if you have plenty of unlabeled training data, you can try to train the layers one by one, starting with the lowest layer then going up, using an unsupervised feature detector algorithm such as RBMs (see Chapter 10) or Autoencoders (see ???). Each layer is trained on the output of the previously trained layers (all layers except the one being trained is locked). Once all layers have been trained this way, you can fine-tune the network using supervised learning (ie. with backpropagation).

This is a rather long and tedious process, but it often works rather well: in fact it is this technique that Geoffrey Hinton and his team used in 2006 and which led to the revival of neural networks and the success of Deep Learning. Until 2010, unsupervised pretraining (typically using RBMs) was the norm for deep nets, and it is only after the vanishing gradients problem was alleviated that it became much more common to train DNNs purely using backpropagation. However, unsupervised pretraining (today typically using Autoencoders rather than RBMs) is still the only option when you have a complex task to solve, no similar model you can reuse, little labeled training data but plenty of unlabeled training data.

⁸ <https://goo.gl/XI02X3>

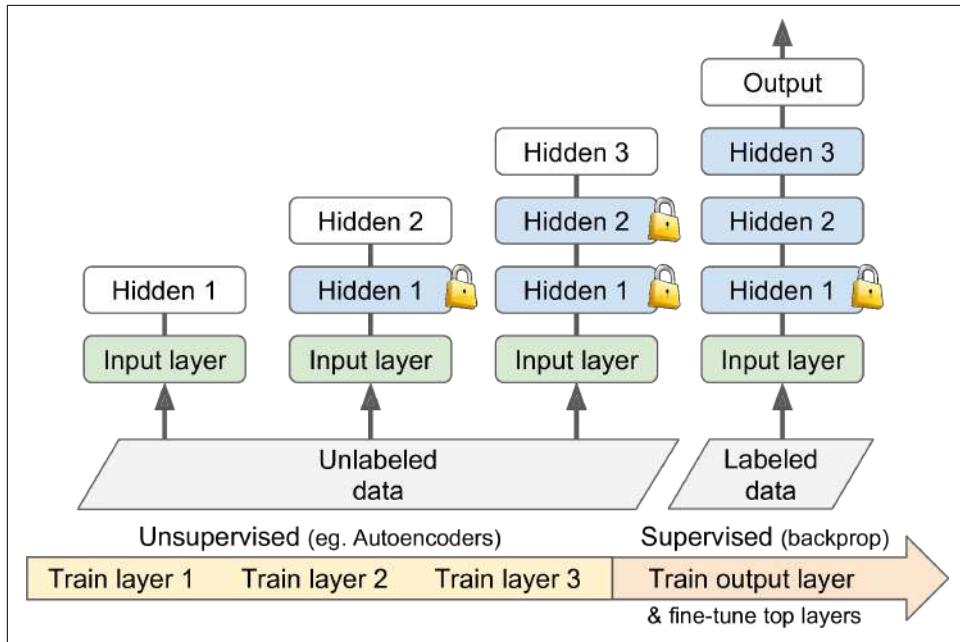


Figure 11-5. Unsupervised pretraining

Pretraining on an auxiliary task

One last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual: clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. However, you could gather a lot of pictures of random people on the Internet and train a first neural network to detect whether or not two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier using little training data.

It is often rather cheap to gather unlabeled training examples, but quite expensive to label them. In this situation, a common technique is to label all your training examples as “good”, then generate many new training instances by corrupting the good ones, and label these corrupted instances as “bad”. Then you can train a first neural network to classify instances as good or bad. For example, you could download millions of sentences, label them as “good”, then randomly change a word in each sentence and label the resulting sentences as “bad”. If a neural network can tell that “The

“dog sleeps” is a good sentence but “The dog they” is bad, it probably knows quite a lot about language. Reusing its lower layers will likely help in many language processing tasks.

Another approach is to train a first network to output a score for each training instance, and use a cost function that ensures that a good instance’s score is greater than a bad instance’s score by at least some margin. This is called *max margin learning*.

Faster optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization and reusing parts of a pretrained network. Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp and finally Adam optimization.

Spoiler alert: the conclusion of this section is that you should almost always use Adam optimization, so if you don’t care about how it works, simply replace your `GradientDescentOptimizer` by an `AdamOptimizer` and skip to the next section! With just this small change, training will typically be several times faster. However, Adam optimization does have three hyperparameters that you can tune (plus the learning rate): the default values usually work fine, but if you ever need to tweak them it may be helpful to know what they do. Adam optimization combines several ideas from other optimization algorithms, so it is useful to look at these algorithms first.

Momentum optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *Momentum optimization*, proposed by Boris Polyak in 1964⁹. In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

Recall that Gradient Descent simply updates the weights θ by directly subtracting the gradient of the cost function $J(\theta)$ with regards to the weights ($\nabla_{\theta}J(\theta)$) multiplied by

⁹ “Some methods of speeding up the convergence of iteration methods”, B. Polyak (1964): <https://goo.gl/F1SE8c>

the learning rate η . The equation is: $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it adds the local gradient to the *momentum vector* \mathbf{m} (multiplied by the learning rate η), and it updates the weights by simply subtracting this momentum vector (see [Equation 11-4](#)). In other words the gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum to grow too large, the algorithm introduces a new hyperparameter β , simply called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta \mathbf{m} + \eta \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

You can easily verify that if the gradient remains constant, the terminal velocity (ie. the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $\frac{1}{1-\beta}$. For example, if $\beta = 0.9$ then the terminal velocity is equal to 10 times the gradient times the learning rate, so Momentum optimization ends up going 10 times faster than Gradient Descent! This allows Momentum optimization to escape from plateaus much faster than Gradient Descent. In particular we saw in [Chapter 4](#) that when the inputs have very different scales the cost function will look like an elongated bowl (see [Figure 4-6](#)): Gradient Descent goes down the steep slope quite fast but then it takes a very long time to go down the valley. In contrast, Momentum optimization will roll down the bottom of the valley faster and faster until it reaches the bottom (the optimum). In Deep Neural Networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using Momentum optimization helps a lot. It can also help roll past local optima.



Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing Momentum optimization in TensorFlow is a no-brainer: just replace the `GradientDescentOptimizer` with the `MomentumOptimizer`, then lie back and profit!

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

The one drawback of Momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than Gradient Descent.

Nesterov Momentum optimization

One small variant to Momentum optimization was proposed by Yurii Nesterov in 1983¹⁰ which makes it almost always faster than vanilla Momentum optimization. The idea of *Nesterov momentum optimization*, also called *Nesterov Accelerated Gradient* (NAG), is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum (see [Equation 11-5](#)). The only difference with vanilla Momentum optimization is that the gradient is measured at $\theta + \beta\mathbf{m}$ rather than at θ .

Equation 11-5. Nesterov momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta - \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (ie. towards the optimum), so it will be slightly more accurate to use the gradient measured a bit further in that direction rather than using the gradient at the original position, as you can see in [Figure 11-6](#) (where ∇_1 represents the gradient of the cost function measured at the starting point θ , and ∇_2 represents the gradient at the point located at $\theta + \beta\mathbf{m}$). As you can see, the Nesterov update ends up slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push further across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus converge faster.

¹⁰ "A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$ ", Y. Nesterov (1983)

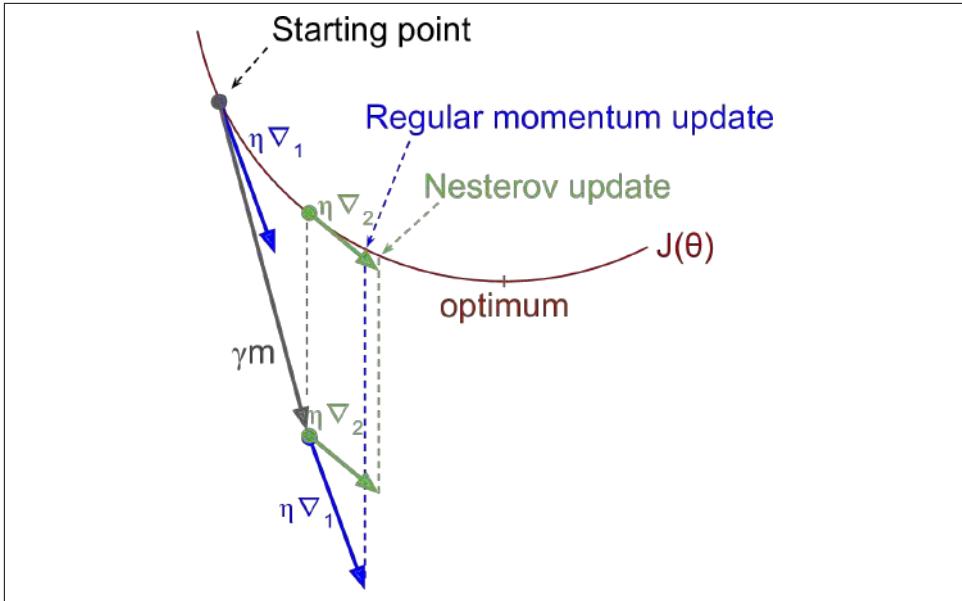


Figure 11-6. Regular vs Nesterov momentum optimization

NAG will almost always speed up training compared to regular momentum optimization. To use it, simply set `use_nesterov=True` when creating the `MomentumOptimizer`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then it slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more towards the global optimum.

The *AdaGrad* algorithm¹¹ achieves this by scaling down the gradient vector along the steepest dimensions (see [Equation 11-6](#)):

Equation 11-6. AdaGrad algorithm

1. $s \leftarrow s + \nabla_{\theta}J(\theta) \otimes \nabla_{\theta}J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta) \oslash \sqrt{s + \epsilon}$

¹¹ “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, J. Duchi *et al.* (2011): <http://goo.gl/4Tyd4j>

The first step accumulates the square of the gradients into the vector \mathbf{s} (the \otimes symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial/\partial\theta_i J(\theta))^2$ for each element s_i of the vector \mathbf{s} : in other words, each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i . If the cost function is steep along the i^{th} dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{s + \epsilon}$ (the \oslash symbol represents the element-wise division, and $\backslash\epsilon$ psilon is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to computing $\theta_i \leftarrow \theta_i - \eta \partial/\partial\theta_i J(\theta)/\sqrt{s_i + \epsilon}$ for all parameters θ_i (simultaneously).

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly towards the global optimum (see [Figure 11-7](#)). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η .

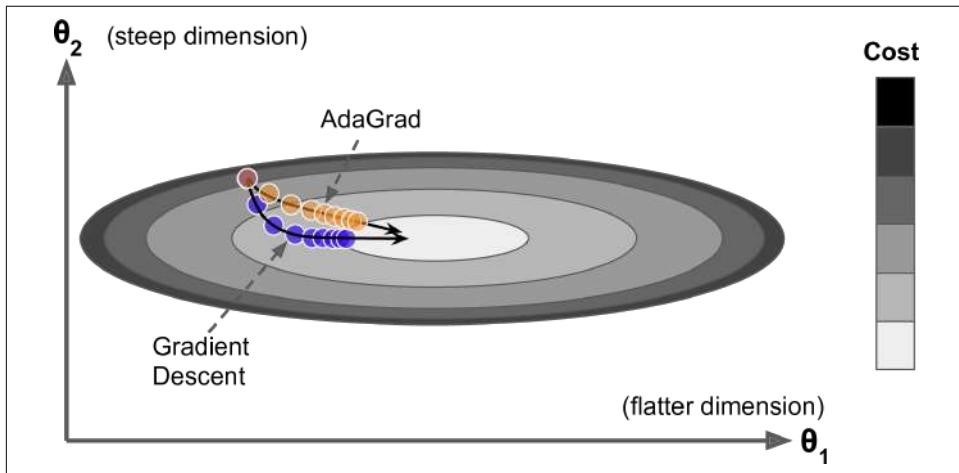


Figure 11-7. AdaGrad vs Gradient Descent

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks: the learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though TensorFlow has an `AdagradOptimizer`, you should not use it to train Deep Neural Networks (it may be efficient for simpler tasks such as Linear Regression, though).

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm¹² fixes this by only accumulating the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step (see [Equation 11-7](#)).

Equation 11-7. RMSProp algorithm

1. $s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well so you may not need to tune it at all.

As you might expect, TensorFlow has an `RMSPropOptimizer` class:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. It also generally performs better than Momentum optimization and Nesterov Accelerated Gradients. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam optimization

*Adam*¹³ (which stands for *Adaptive Moment Estimation*) combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it

¹² This algorithm was created by T. Tieleman and G. Hinton in 2012, and presented by G. Hinton in his Coursera class on Neural Networks (slides: <http://goo.gl/RsQeis>, video: <https://goo.gl/XUblJj>). Amusingly, since the authors have not written a paper to describe it, researchers often cite “slide 29 in lecture 6” in their papers.

¹³ “Adam: A Method for Stochastic Optimization”, D. Kingma, J. Ba (2015): <https://goo.gl/Un8Axa>

keeps track of an exponentially decaying average of past squared gradients (see [Equation 11-8](#)).¹⁴

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^T}$
5. $\theta \leftarrow \theta - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- t represents the iteration number (starting at 1).

If you just look at steps 1, 2 and 5, you will notice the close similarity with both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). The steps 3 and 4 are somewhat of a technical detail: since \mathbf{m} and \mathbf{s} are initialized at zero, they will be biased towards zero at the beginning of training, so these two steps will help boost \mathbf{m} and \mathbf{s} at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ϵ is usually initialized to a tiny number such as 10^{-8} . These are the default values for TensorFlow's `AdamOptimizer` class, so you can simply use:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

In fact, since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η . You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

¹⁴ These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.



All the optimization techniques discussed above only rely on the *first order partial derivatives* (*Jacobians*). The optimization literature contains amazing algorithms based on the *second order partial derivatives* (*Hessians*). Unfortunately, these algorithms are very hard to apply to Deep Neural Networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Training sparse models

All the optimization algorithms presented above produce dense models, meaning that most parameters will be non-zero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One trivial way to achieve this is to train the model as usual then get rid of the tiny weights (set them to zero).

Another option is to apply strong ℓ_1 regularization during training, as it pushes the optimizer to zero-out as many weights as it can (as discussed in [Chapter 4](#) about Lasso regression).

However, in some cases these techniques may remain insufficient. One last option is to apply *Dual Averaging*, often called *Follow The Regularized Leader* (FTRL), a technique proposed by Y. Nesterov¹⁵. When used with ℓ_1 regularization, this technique often leads to very sparse models. TensorFlow implements a variant of FTRL called *FTRL-Proximal*¹⁶ in the `FTRLOptimizer` class (see the paper for more details).

Learning rate scheduling

Finding the optimal learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in [Chapter 4](#)). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never settling down (unless you use an adaptive learning rate optimization algorithm such as AdaGrad, RMSProp or Adam, but even then it may take time to settle). If you have a limited computing budget, you may have to inter-

¹⁵ “Primal-dual subgradient methods for convex problems”, Y. Nesterov (2005)

¹⁶ “Ad Click Prediction: a View from the Trenches”, H. McMahan *et al.* (2013): <https://goo.gl/bxme2B>

rupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

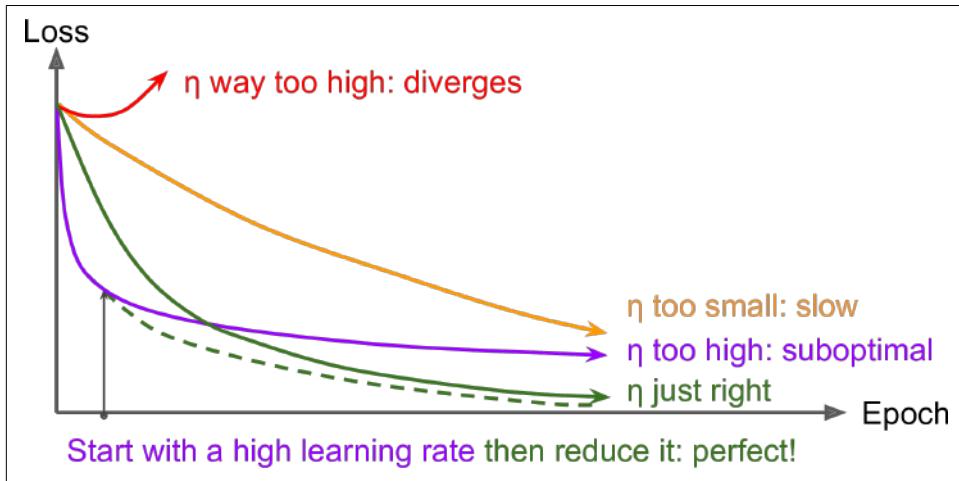


Figure 11-8. Learning curves for various learning rates η

You may be able to find a fairly good learning rate by training your network several times during just a few epochs using various learning rates and comparing the learning curves. The ideal learning rate will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training: these strategies are called *learning schedules* (we briefly introduced this concept in [Chapter 4](#)). The most common strategies are:

- *Predetermined piecewise constant learning rate*: for example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to find the right learning rates and when to use them.
- *Performance scheduling*: measure the validation error every N steps (just like for Early Stopping) and reduce the learning rate by a factor λ when the error stops dropping.
- *Exponential scheduling*: set the learning rate to a function of the iteration number t : $\eta(t) = \eta_0 10^{-t/r}$. This works great, but it requires tuning η_0 and r . The learning rate will drop by a factor of 10 every r steps.

- *Power scheduling*: set the learning rate to $\eta(t) = \eta_0(1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling but the learning rate drops much more slowly.

A 2013 paper¹⁷ by A. Senior *et al.* compared the performance of some of the most popular learning schedules when training deep neural networks for speech recognition using Momentum optimization: it concluded that, in this setting, both performance scheduling and exponential scheduling performed well, but they favored exponential scheduling because it is simpler to implement, easy to tune, and it converged slightly faster to the optimal solution.

Implementing a learning schedule with TensorFlow is fairly straightforward:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False)
learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                            decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

After setting the hyperparameter values, we create a non-trainable variable `global_step` (initialized to 0) to keep track of the current training iteration number. Then we define an exponentially decaying learning rate (with $\eta_0 = 0.1$ and $r = 10,000$) using TensorFlow's `exponential_decay()` function. Then we create an optimizer (in this example a `MomentumOptimizer`) using this decaying learning rate. Finally, we create the training operation by calling the optimizer's `minimize()` method: since we pass it the `global_step` variable, it will kindly take care of incrementing it. That's it!

Since AdaGrad, RMSProp and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule. For other optimization algorithms using exponential decay or performance scheduling can considerably speed up convergence.

Avoiding overfitting through regularization

Deep Neural Networks typically have tens of thousands of connections, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and it can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set.

¹⁷ “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition”, A. Senior *et al.* (2013): <http://goo.gl/Hu6Zya>

With four parameters I can fit an elephant and with five I can make him wiggle his trunk.

—John von Neumann, *Cited by Enrico Fermi in Nature* 427

With millions of parameters you can fit the whole zoo. In this section we will present some of the most popular regularization techniques for neural networks, and how to implement them with TensorFlow: Early Stopping, ℓ_1 and ℓ_2 regularization, Dropout and Data augmentation.

Early Stopping

To avoid overfitting the training set, a great solution is Early Stopping (introduced in [Chapter 4](#)): just interrupt training when its performance on the validation set starts dropping.

One way to implement this with TensorFlow is to evaluate the model on a validation set at regular intervals (eg. every 50 steps), and save a “winner” snapshot if it outperforms previous “winner” snapshots. Count the number of steps since the last “winner” snapshot was saved, and interrupt training when this number reaches some limit (eg. 2,000 steps). Then restore the last “winner” snapshot.

Another option is to use TF.Learn’s `ValidationMonitor` class which manages this for you:

```
feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(
    feature_columns = feature_columns,
    hidden_units=[300, 100],
    n_classes=10,
    model_dir="/tmp/my_model",
    config=tf.contrib.learn.RunConfig(save_checkpoints_secs=60)
)

validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
    X_val,
    y_val,
    every_n_steps=50,
    early_stopping_metric="loss",
    early_stopping_metric_minimize=True,
    early_stopping_rounds=2000
)

dnn_clf.fit(x=X_train,
            y=y_train,
            steps=40000,
            monitors=[validation_monitor]
        )
```

The first section creates a `DNNClassifier` just like in [Chapter 10](#), but we added the instruction to automatically save a checkpoint every 60 seconds to the `/tmp/my_model` directory. This is required because the `ValidationMonitor` evaluates the latest checkpoint.

The next section creates a `ValidationMonitor`: it will evaluate the model every 50 steps on the validation data (`X_val` and `y_val`), and interrupt training if the model's `loss` metric is not improved for 2,000 steps in a row (recall that the `DNNClassifier` uses cross entropy by default as its cost function).

Finally, the last line trains the model for a maximum of 40,000 steps, but since it uses the `ValidationMonitor` we created it will likely stop earlier.

Although Early Stopping works very well in practice, you can usually get much higher performance out of your network by combining it with other regularization techniques.

ℓ_1 and ℓ_2 regularization

Just like you did in [Chapter 4](#) for simple linear models, you can use ℓ_1 and ℓ_2 regularization so constrain a neural network's connection weights (but typically not its biases).

One way to do this using TensorFlow is to simply add the appropriate regularization terms to your cost function. For example, assuming you have just one hidden layer with weights `weights1` and one output layer with weights `weights2`, then you can apply ℓ_1 regularization like this:

```
[...] # construct the neural network
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

However, if there are many layers, this approach is not very convenient. Fortunately, TensorFlow provides a better option: many functions that create variables (such as `get_variable()` or `fully_connected()`) accept a `*_regularizer` argument for each created variable (eg. `weights_regularizer`). You can pass any function that takes weights as an argument and returns the corresponding regularization loss. The `l1_regularizer()`, `l2_regularizer()` and `l1_l2_regularizer()` functions return such functions. The following code puts all this together:

```
with arg_scope(
    [fully_connected],
    weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, activation_fn=None, scope="out")
```

This code creates a neural network with two hidden layers and one output layer, and it also creates nodes in the graph to compute the ℓ_1 regularization loss corresponding to each layer's weights. TensorFlow automatically adds these nodes to a special collection containing all the regularization losses. You just need to add these regularization losses to your overall loss, like this:

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add(base_loss, reg_losses, name="loss")
```



Don't forget to add the regularization losses to your overall loss, or else they will simply be ignored.

Dropout

The most popular regularization technique for deep neural networks is arguably *Dropout*. It was proposed¹⁸ by G. Hinton in 2012 and further detailed in a paper¹⁹ by N. Srivastava *et al.*, and it has proven to be highly successful: even the state-of-the-art neural networks got a 1-2% accuracy boost simply by adding Dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily “dropped out”, meaning it will be entirely ignored during this training step, but it may be active during the next step (see [Figure 11-9](#)). The hyperparameter p is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss below).

¹⁸ “Improving neural networks by preventing co-adaptation of feature detectors”, G. Hinton *et al.* (2012): <https://goo.gl/PMjVnG>

¹⁹ “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, N. Srivastava *et al.* (2014): <http://goo.gl/DNKZo1>

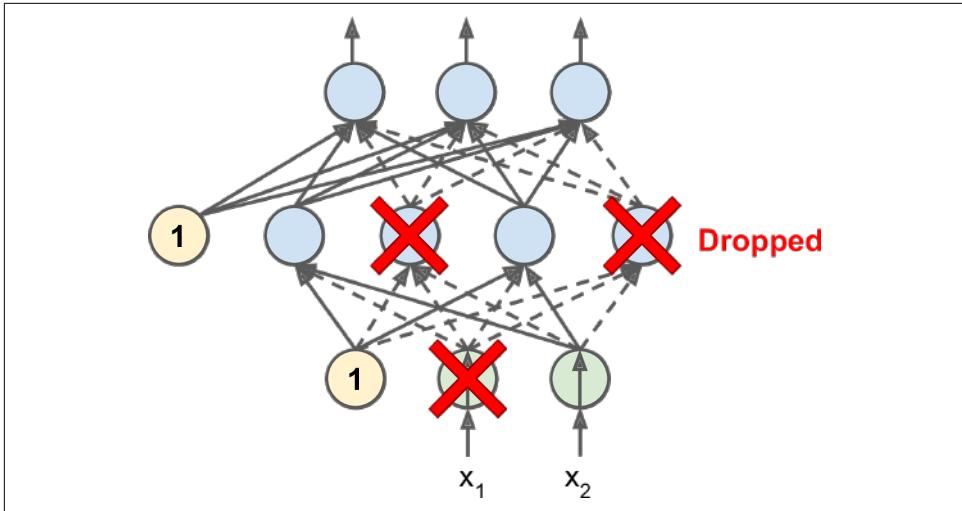


Figure 11-9. Dropout regularization

It is quite surprising at first that this rather brutal technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows, perhaps it would! The company would obviously be forced to adapt its organization: it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their co-workers, not just a handful of them. The company would be much more resilient: if one person quits, it won't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks: neurons trained with Dropout cannot co-adapt with their neighboring neurons: they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons, they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.

Another way to understand the power of Dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there are a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run a 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are obviously not independent since they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

There is one small but important technical detail: suppose $p = 50$, then during testing a neuron will be connected to twice as many input neurons as it was (on average) during training. To compensate for this fact, we need to multiply each neuron's input connection weights by 0.5 after training. If we don't, each neuron will get a total input signal roughly twice as large as what the network was trained on, and it is unlikely to perform well. More generally, we need to multiply each input connection weight by the *keep probability* ($1 - p$) after training. Alternatively, we can divide each neuron's output by the keep probability during training (these alternatives are not perfectly equivalent, but they work equally well).

To implement Dropout using TensorFlow, you can simply apply the `dropout()` function to the input layer and to the output of every hidden layer. During training, this function randomly drops some items (setting them to 0) and divides the remaining items by the keep probability. After training, this function does nothing at all. The following code applies Dropout regularization to our 3 layer neural network:

```
from tensorflow.contrib.layers import dropout

[...]
is_training = tf.placeholder(tf.bool, shape=(), name='is_training')

keep_prob = 0.5
X_drop = dropout(X, keep_prob, is_training=is_training)

hidden1 = fully_connected(X_drop, n_hidden1, scope="hidden1")
hidden1_drop = dropout(hidden1, keep_prob, is_training=is_training)

hidden2 = fully_connected(hidden1_drop, n_hidden2, scope="hidden2")
hidden2_drop = dropout(hidden2, keep_prob, is_training=is_training)

logits = fully_connected(hidden2_drop, n_outputs, activation_fn=None,
                         scope="outputs")
```



You want to use the `dropout()` function in `tensorflow.contrib.layers`, not the one in `tensorflow.nn`. The first one turns off (no-op) when not training, which is what you want, while the second one does not.

Of course, just like you did earlier for Batch Normalization, you need to set `is_training` to `True` when training, and to `False` when testing.

If you observe that the model is overfitting, you can increase the dropout rate (ie. reduce the `keep_prob` hyperparameter). Conversely, you should try decreasing the dropout rate (ie. increasing `keep_prob`) if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly: it is generally well worth the extra time and effort.



Dropconnect is a variant of Dropout where individual connections are dropped randomly rather than whole neurons. In general Dropout performs better.

Max-norm regularization

Another regularization technique that is quite popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights \mathbf{w} of the incoming connections such that $\|\mathbf{w}\|_2 \leq r$, where r is the max norm hyperparameter and $\|\cdot\|_2$ is the ℓ_2 norm.

This constraint is typically implemented by computing $\|\mathbf{w}\|_2$ after each training step and clipping \mathbf{w} if needed ($\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\|\mathbf{w}\|_2}$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients problems (if you are not using Batch Normalization).

TensorFlow does not provide an off-the-shelf Max-norm regularizer, but it is not too hard to implement. The following code creates a node `clip_weights` that will clip the `weights` variable along the second axis so that each row vector has a maximum norm of 1.0.

```
threshold = 1.0
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

You would then apply this operation after each training step, like so:

```
with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            clip_weights.eval()
```

You may wonder how to get access to the `weights` variable of each layer. For this you can simply use a variable scope like this:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
```

```
with tf.variable_scope("hidden1", reuse=True):
    weights1 = tf.get_variable("weights")
```

Alternatively, you can use the root variable scope:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
[...]

with tf.variable_scope("", reuse=True):
    weights1 = tf.get_variable("hidden1/weights")
    weights2 = tf.get_variable("hidden2/weights")
```

If you don't know what the name of a variable is, you can either use TensorBoard to find out or simply use the `all_variables()` function and print out all the variable names:

```
for variable in tf.all_variables():
    print(variable.name)
```

Although the above solution should work fine, it is a bit messy. A cleaner solution is to create a `max_norm_regularizer()` function and use it just like the `l1_regularizer()` function above:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None # there is no regularization loss term
    return max_norm
```

This function returns a parametrized `max_norm()` function that you can use like any other regularizer:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          weights_regularizer=max_norm_reg)
```

Note that Max norm regularization does not require adding a regularization loss term to your overall loss function, so the `max_norm()` function returns `None`. But you still need to be able to run the `clip_weights` operation after each training step, so you need to be able to get a handle on it. This is why the `max_norm()` function adds the `clip_weights` node to a collection of Max norm clipping operations. You need to fetch these clipping operations and run them after each training step:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
```

```
[...]
for X_batch, y_batch in zip(X_batches, y_batches):
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    sess.run(clip_all_weights)
```

Much cleaner code, isn't it?

Data augmentation

One last regularization technique consists in generating new training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting, making this a regularization technique. The trick is to generate realistic training instances: ideally a human should not be able to tell which instances were generated and which ones were not. Moreover, simply adding white noise will not help: the modifications you apply should be learnable (white noise is not).

For example, if your model is meant to classify pictures of mushrooms, you can slightly shift, rotate and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see [Figure 11-10](#)). This forces the model to be more tolerant to the position, orientation and size of the mushrooms on the picture. If you want the model to be more tolerant to lighting conditions, you can similarly generate many images with various contrasts. Assuming the mushrooms are symmetrical, you can also flip the pictures horizontally. By combining these transformations you can greatly increase the size of your training set.

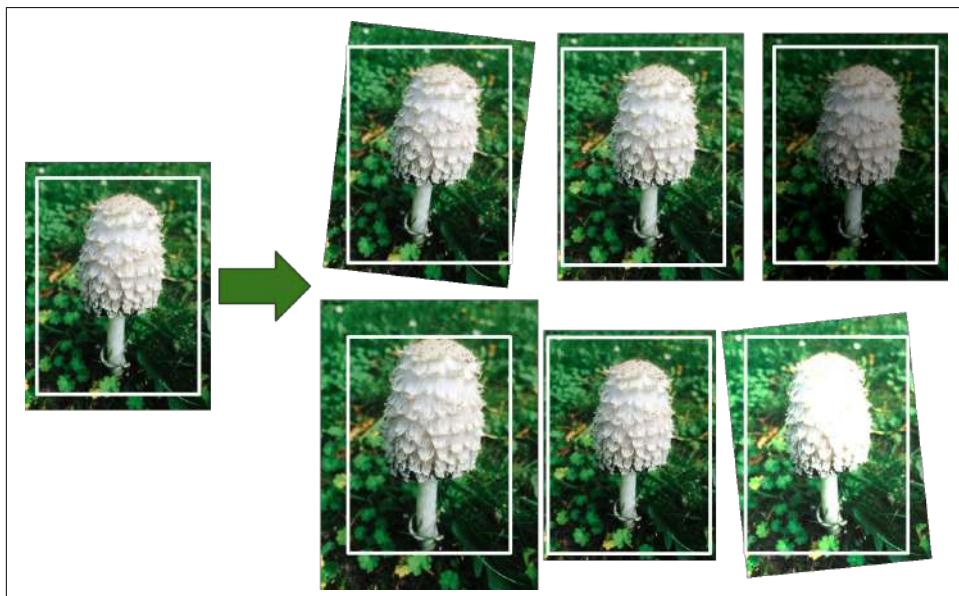


Figure 11-10. Generating new training instances from existing ones

It is often preferable to generate training instances on the fly during training rather than wasting storage space and network bandwidth. TensorFlow offers several image manipulation operations such as transposing (shifting), rotating, resizing, flipping, cropping, adjusting the brightness, contrast, saturation and hue (see the API documentation for more details). This makes it easy to implement data augmentation for image datasets.



Another powerful technique to train very deep neural networks is to add *skip connections* (a skip connection is when you add the input of a layer to the output of a higher layer). We will explore this idea in ??? when we talk about deep Residual Networks.

Practical guidelines

In this chapter, we have covered a wide range of techniques and you may be asking yourself which ones you should use. The following configuration will work fine in most cases:

Table 11-2. Default Deep Neural Network configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

Of course, you should try to reuse parts of a pretrained neural network if you can find one that solves a similar problem.

This default configuration may need to be tweaked:

- If you can't find a good learning rate (convergence was too slow, you increased the training rate and now convergence is fast but the network's accuracy is sub-optimal) then you can try adding a learning schedule such as exponential decay.
- If your training set is a bit too small, you can implement data augmentation.
- If you need a sparse model, you can add some ℓ_1 regularization to the mix (and optionally zero out the tiny weights after training). In you need an even sparser model, you can try using FTRL instead of Adam optimization, along with ℓ_1 regularization.

- If you need a lightning fast model at runtime, you may want to drop Batch Normalization, and possibly replace the ELU activation function by the Leaky ReLU. Having a sparse model will also help.

With these guidelines, you are now ready to train very deep nets. Well, if you are very patient that is! If you use a single machine, you may have to wait for days or even months for training to complete. In the next chapter we will discuss how to use Distributed TensorFlow to train and run models across many servers and GPUs.

Exercises

1. Is it ok to initialize all the weights to the same value as long as this value is selected randomly using He initialization?
2. Is it ok to initialize the bias terms to zero?
3. Can you name 3 advantages of the ELU activation function over ReLU?
4. In which cases would you want to use each of the following activation functions: ELU, Leaky ReLU (and its variants), ReLU, tanh, logistic and softmax?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (eg. 0.99999) when using a `MomentumOptimizer`?
6. Can you name 3 ways you can produce a sparse model?
7. Does dropout slow down training? Does it slow down inference (ie. making predictions on new instances)?
8. Deep Learning:
 - Build a DNN with 5 hidden layers of 100 neurons each, He initialization and the ELU activation function.
 - Using Adam optimization and Early Stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with 5 neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
 - Tune the hyperparameters using cross-validation and see what precision you can achieve.
 - Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
 - Is the model overfitting the training set? Try adding Dropout to every layer and try again. Does it help?
9. Transfer learning:

- Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezing them, and replaces the softmax output layer by a fresh new one.
 - Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?
 - Try caching the frozen layers, and train the model again: how much faster is it now?
 - Try again reusing just 4 hidden layers instead of 5. Can you achieve a higher precision?
 - Now unfreeze the top 2 hidden layers and continue training: can you get the model to perform even better?
10. Pretraining on an auxiliary task.
- In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifiers using very little training data. Start by building two DNNs (let's call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have 5 hidden layers of 100 neurons each, He initialization and ELU activation. Next, add a single output layer on top of both DNNs. You should use TensorFlow's `concat()` function with `concat_dim=1` to concatenate the outputs of both DNNs along the horizontal axis, then feed the result to the output layer. This output layer should contain a single neuron using the logistic activation function.
 - Split the MNIST training set in two sets: split #1 should contain 55,000 images, and split #2 should contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.
 - Train the DNN on this training set: for each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.
 - Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on with 10 neurons. Train this network on

split #2 and see if you can achieve high performance despite having only 500 images per class.

Solutions to these exercises are available in [Appendix A](#).

Distributing TensorFlow across devices and servers

In [Chapter 11](#) we discussed several techniques that can considerably speed up training: better weight initialization, Batch Normalization, sophisticated optimizers, and so on. However, even with all of these techniques, training a large neural network on a single machine with a single CPU can take days or even weeks.

In this chapter we will see how to use TensorFlow to distribute computations across multiple devices (CPUs and GPUs) and run them in parallel (see [Figure 12-1](#)). First we will distribute computations across multiple devices on just one machine, then on multiple devices across multiple machines.

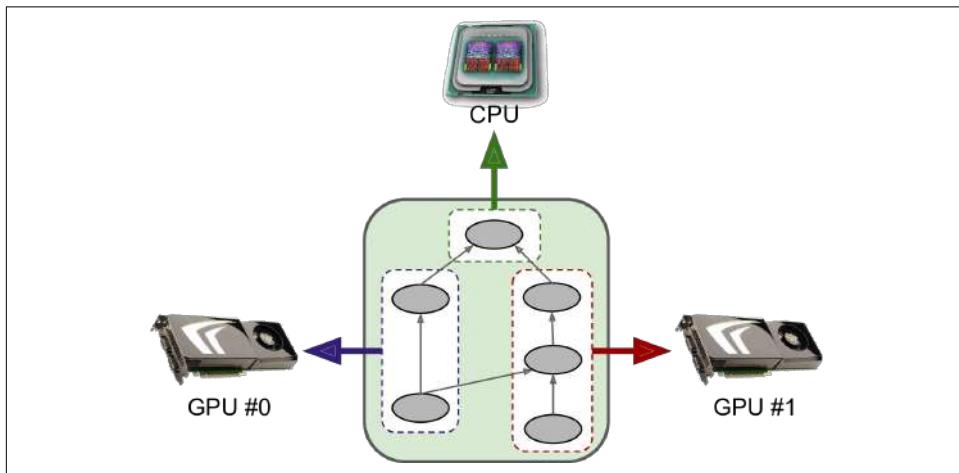


Figure 12-1. Executing a TensorFlow graph across multiple devices in parallel

TensorFlow's support of distributed computing is one of its main highlights compared to other neural network frameworks. It gives you full control over how to split (or replicate) your computation graph across devices and servers, and it lets you parallelize and synchronize operations in flexible ways so you can choose between all sorts of parallelization approaches.

We will look at some of the most popular approaches to parallelizing the execution and training of a neural network. Instead of waiting for weeks for a training algorithm to complete, you may end up waiting for just a few hours. Not only does this save an enormous amount of time, it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

Other great use cases of parallelization include exploring a much larger hyperparameter space when fine-tuning your model, and running large ensembles of neural networks efficiently.

But we must learn to walk before we can run. Let's start by parallelizing simple graphs across several GPUs on a single machine.

Multiple devices on a single machine

You can often get a major performance boost simply by adding GPU cards to a single machine. In fact, in many cases this will suffice, you won't need to use multiple machines at all. For example, you can typically train a neural network just as fast using 8 GPUs on a single machine rather than 16 GPUs across multiple machines (due to the extra delay imposed by network communications in a multi-machine setup).

In this section we will look at how to setup your environment so that TensorFlow can use multiple GPU cards on one machine, then we will look at how you can distribute operations across available devices and execute them in parallel.

Installation

In order to run TensorFlow on multiple GPU cards, you first need to make sure your GPU cards have NVidia Compute Capability greater or equal to 3.0. This includes Nvidia's Titan, Titan X, K20 and K40 cards (if you own another card, you can check its compatibility on <https://developer.nvidia.com/cuda-gpus>).



If you don't own any GPU cards, you can use a hosting service with GPU capability such as Amazon AWS. Detailed instructions to setup TensorFlow 0.9 with Python 3.5 on an Amazon AWS GPU instance are available in this helpful blog post by Žiga Avsec: <http://goo.gl/kbge5b>. Google is also working on a cloud service called *Cloud Machine Learning* (currently available only in limited preview, as of September 2016) to run TensorFlow graphs: <https://cloud.google.com/ml>. Of course, you can also buy a GPU card. Tim Dettmers wrote a [great blog post](#) to help you choose, and he updates it regularly.

You must then download and install the appropriate version of the CUDA and cuDNN libraries (CUDA 7.5 and cuDNN 4 if you are using the binary installation of TensorFlow 0.10.0). Nvidia's *Compute Unified Device Architecture* library (CUDA) allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration). Nvidia's *CUDA Deep Neural Network* library (cuDNN) is a GPU-accelerated library of primitives for DNNs. It provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see ???). It is part of Nvidia's Deep Learning SDK (note that it requires creating an Nvidia developer account in order to download it). TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see [Figure 12-2](#)).

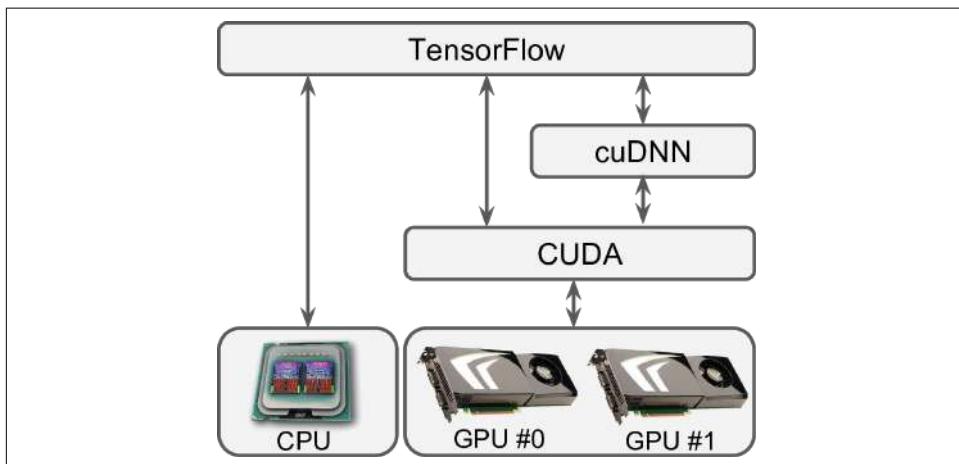


Figure 12-2. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

You can use the `nvidia-smi` command to check that CUDA is properly installed. It lists the available GPU cards, as well as processes running on each card:

```
$ nvidia-smi  
Wed Sep 16 09:50:03 2016
```

```

+-----+
| NVIDIA-SMI 352.63      Driver Version: 352.63      |
+-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====|
|  0  GRID K520          Off | 0000:00:03.0    Off |                  N/A |
| N/A   27C    P8    17W / 125W |      11MiB / 4095MiB |      0%     Default |
+-----+-----+-----+
+-----+
| Processes:                               GPU Memory |
| GPU     PID  Type  Process name          Usage     |
| =====+=====+=====+=====
| No running processes found
+-----+

```

Next you must set a few environment variables so TensorFlow knows where to find CUDA and cuDNN. On MacOSX, you can add the following lines to your `.bash_profile` (on Linux, replace `DYLD_LIBRARY_PATH` with `LD_LIBRARY_PATH`):

```

export CUDA_HOME=/usr/local/cuda
export PATH=$PATH:$CUDA_HOME/bin
export CUDA_LIBS=$CUDA_HOME/lib64:$CUDA_HOME/extras/CUPTI/lib64
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:$CUDA_LIBS

```

Finally, you must install TensorFlow with GPU support. If you created an isolated environment using `virtualenv`, you first need to activate it:

```

$ cd $ML_PATH           # Your ML working directory (eg. $HOME/ml)
$ source env/bin/activate

```

Then install the appropriate GPU-enabled TensorFlow binary for your platform. For example, for MacOSX:

```

$ export TF_BASE_URL=https://storage.googleapis.com/tensorflow
$ export TF_BINARY_URL=$TF_BASE_URL/mac/gpu/tensorflow-0.10.0-py3-none-any.whl
$ pip3 install --upgrade $TF_BINARY_URL

```

Now you can open up a python shell and check that TensorFlow detects and uses CUDA and cuDNN properly by importing TensorFlow and creating a session:

```

>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurl.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797

```

```
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0: Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

Looks good! TensorFlow detected the CUDA and cuDNN libraries, and it used the CUDA library to detect the GPU card (in this case an Nvidia Grid K520 card).



The instructions above may change slightly in the future, so please check out TensorFlow's installation page for more details (and other installation options, such as building from source).

Managing the GPU RAM

By default TensorFlow automatically grabs all the RAM in all available GPUs the first time you run a graph, so you will not be able to start a second TensorFlow program while the first one is still running. If you try, you will get the following error:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from
device: CUDA_ERROR_OUT_OF_MEMORY
```

One solution is to run each process on different GPU cards: to do this, the simplest option is to set the `CUDA_VISIBLE_DEVICES` environment variable so that each process only sees the appropriate GPU cards. For example, you could start two programs like this:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program #1 will only see GPU cards 0 and 1 (numbered 0 and 1 respectively), and program #2 will only see GPU cards 2 and 3 (numbered 1 and 0 respectively). Everything will work fine (see [Figure 12-3](#)).

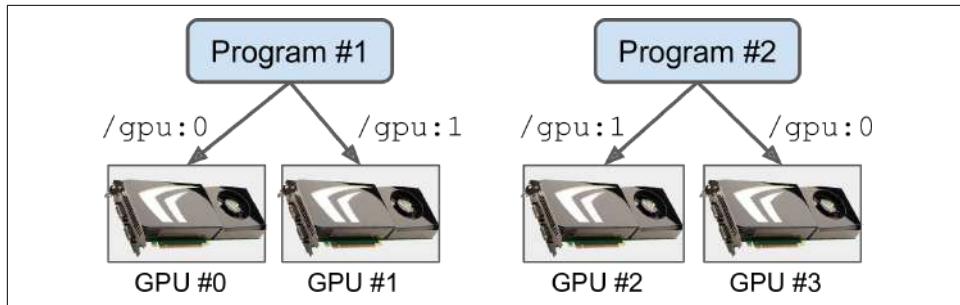


Figure 12-3. Each program gets two GPUs for itself

Another option is to tell TensorFlow to grab only a fraction of the memory. For example, to make TensorFlow grab only 40% of each GPU's memory, you must create a `ConfigProto` object, set its `gpu_options.per_process_gpu_memory_fraction` option to 0.4, and create the session using this configuration:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
```

Now two programs like this one can run in parallel using the same GPU cards (but not three since $3 \times 0.4 > 1$). See Figure 12-4.

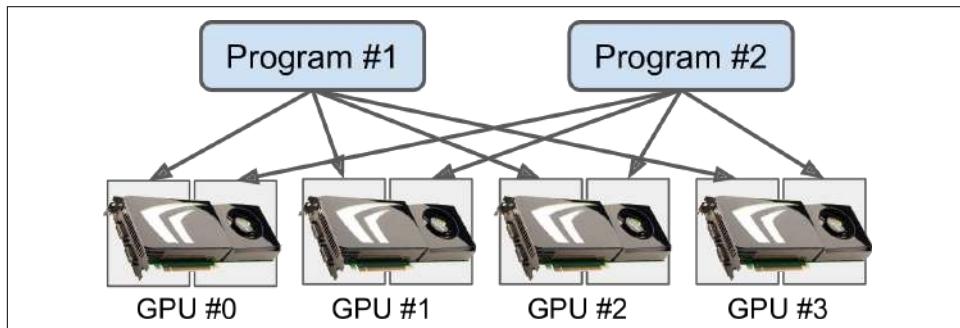


Figure 12-4. Each program gets all four GPUs, but with only 40% of the RAM each

If you run the `nvidia-smi` command while both programs are running, you should see that each process holds roughly 40% of the total RAM of each card:

```
$ nvidia-smi
[...]
+-----+
| Processes:                               GPU Memory |
| GPU     PID  Type  Process name        Usage      |
+-----+
| 0      5231  C    python              1677MiB |
| 0      5262  C    python              1677MiB |
```

	1	5231	C	python	1677MiB	
	1	5262	C	python	1677MiB	
[...]						

Yet another option is to tell TensorFlow to only grab memory when it needs it. To do this you must set `config.gpu_options.allow_growth` to `True`. However, TensorFlow never releases memory once it has grabbed it (to avoid memory fragmentation) so you may still run out of memory after a while. It may be harder to guarantee a deterministic behavior using this option, so in general you probably want to stick with one of the previous options.

Ok, now you have a working GPU-enabled TensorFlow installation. Let's see how to use it!

Placing operations on devices

The TensorFlow white-paper¹ presents a friendly *dynamic placer* algorithm that automatically distributes operations across all available devices, taking into account things like the measured computation time in previous runs of the graph, estimations of the size of the input and output tensors to each operation, the amount of RAM available in each device, communication delay when transferring data in and out of devices, hints and constraints from the user, and so on. Unfortunately this sophisticated algorithm is internal to Google: it was not released in the Open Source version of TensorFlow. The reason it was left out seems to be that in practice a small set of placement rules specified by the user actually results in more efficient placement than what the dynamic placer is capable of. However, the TensorFlow team is working on improving the dynamic placer, and perhaps it will eventually be good enough to be released.

Until then TensorFlow relies on the *simple placer*, which (as its name suggests) is very basic.

Simple placement

Whenever you run a graph, if TensorFlow needs to evaluate a node that is not placed on a device yet, it uses the simple placer to place it, along with all other nodes that are not placed yet. The simple placer respects the following rules:

- if a node was already placed on a device in a previous run of the graph, it is left on that device,
- else, if the user *pinned* a node to a device (see below), the placer places it on that device,

¹ “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”, Google Research (2015): <http://goo.gl/vSjA14>

- else, it defaults to GPU #0, or the CPU if there is no GPU.

As you can see, placing operations on the appropriate device is mostly up to you. If you don't do anything, the whole graph will be placed on the default device. To pin nodes onto a device, you must create a device block using the `device()` function. For example, the following code pins the variable `a` and the constant `b` on the CPU, but the multiplication node `c` is not pinned on any device, so it will be placed on the default device:

```
with tf.device("/cpu:0"):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)

c = a * b
```



The `"/cpu:0"` device aggregates all CPUs on a multi-CPU system. There is currently no way to pin nodes on specific CPUs or to use just a subset of all CPUs.

Logging placements

Let's check that the simple placer respects the placement constraints we have just defined. For this you can set the `log_device_placement` option to `True`: this tells the placer to log a message whenever it places a node. For example:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520,
pci bus id: 0000:00:03.0)
[...]
>>> x.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

The lines starting with "I" for Info are the log messages. When we create a session, TensorFlow logs a message to tell us that it has found a GPU card (in this case the Grid K520 card). Then the first time we run the graph (in this case when initializing the variable `a`), the simple placer is run and it places each node on the device it was assigned to. As expected, the log messages show that all nodes are placed on `"/cpu:0"` except the multiplication node which ends up on the default device `"/gpu:0"` (you

can safely ignore the prefix `/job:localhost/replica:0/task:0` for now, we will talk about it below). Notice that the second time we run the graph (to compute `c`), the placer is not used since all the nodes TensorFlow needs to compute `c` are already placed.

Dynamic placement function

When you create a device block, you can specify a function instead of a device name. TensorFlow will call this function for each operation it needs to place in the device block, and the function must return the name of the device to pin the operation on. For example, the following code pins all the variable nodes to `"/cpu:0"` (in this case just the variable `a`) and all other nodes to `"/gpu:0"`:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)
    c = a * b
```

You can easily implement more complex algorithms, such as pinning variables across GPUs in a round-robin fashion.

Operations and kernels

For a TensorFlow operation to run on a device, it needs to have an implementation for that device: this is called a *kernel*. Many operations have kernels for both CPUs and GPUs, but not all of them. For example, TensorFlow does not have a GPU kernel for integer variables, so the following code will fail when TensorFlow tries to place the variable `i` on GPU #0:

```
>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device
to node 'Variable': Could not satisfy explicit device specification
```

Note that TensorFlow infers that the variable must be of type `int32` since the initialization value is an integer. If you change the initialization value to `3.0` instead of `3`, or if you explicitly set `dtype=tf.float32` when creating the variable, everything will work fine.

Soft placement

By default, if you try to pin an operation on a device for which the operation has no kernel, you get the exception shown above when TensorFlow tries to place the operation on the device. If you prefer TensorFlow to fallback to the CPU instead, you can set the `allow_soft_placement` configuration option to `True`:

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)

config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
sess.run(i.initializer) # the placer runs and falls back to /cpu:0
```

So far we have discussed how to place nodes on different devices. Now let's see how TensorFlow will run these nodes in parallel.

Parallel execution

When TensorFlow runs a graph, it starts by finding out the list of nodes that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then starts evaluating the nodes with zero dependencies (ie. source nodes). If these nodes are placed on separate devices, they obviously get evaluated in parallel. If they are placed on the same device, they get evaluated in different threads, so they may run in parallel too (in separate GPU threads or CPU cores).

TensorFlow manages a thread pool on each device to parallelize operations (see [Figure 12-5](#)). These are called the *inter-op thread pools*. Some operations have multi-threaded kernels: they can use other thread pools (one per device) called the *intra-op thread pools*.

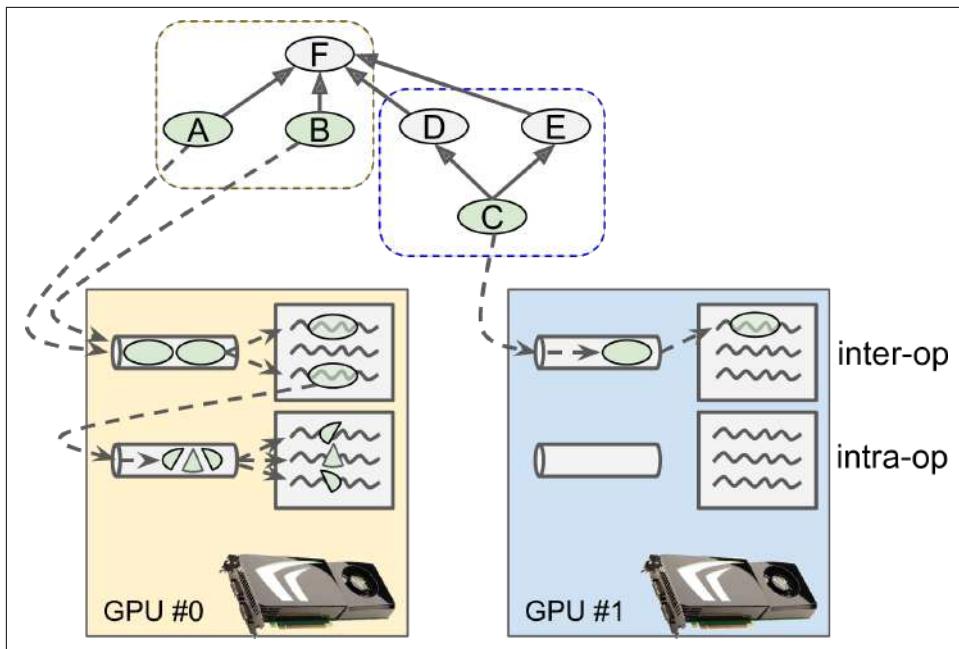


Figure 12-5. Parallelized execution of a TensorFlow graph

For example, in Figure 12-5, operations A, B and C are source ops so they can immediately be evaluated. Operations A and B are placed on GPU #0, so they are sent to this device’s inter-op thread pool, and immediately evaluated in parallel. Operation A happens to have a multi-threaded kernel: its computations are split in 3 parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #1’s inter-op thread pool.

As soon as operation C will finish, the dependency counters of operations D and E will be decremented and will both reach zero, so both operations will be sent to the inter-op thread pool to be executed.



You can control the number of threads per inter-op pool by setting the `inter_op_parallelism_threads` option. Note that the first session you start creates the inter-op thread pools. All other sessions will just reuse them unless you set the `use_per_session_threads` option to True. You can control the number of threads per intra-op pool by setting the `intra_op_parallelism_threads` option.

Control dependencies

In some cases, it may be wise to postpone the evaluation of an operation even though all the operations that it depends on have been executed. For example, if it uses a lot

of memory but its value is only needed much further in the graph, it would be best to evaluate it at the last moment to avoid needlessly occupying RAM that other operations may need. Another example is a set of operations that depend on data located outside of the device. If they all run at the same time, they may saturate the device's communication bandwidth, and they will end up all waiting on I/O. Other operations that need to communicate data will also be blocked. It would be preferable to execute these communication-heavy operations sequentially, allowing the device to perform other operations in parallel.

To postpone evaluation of some nodes, a simple solution is to add *control dependencies*. For example, the following code tells TensorFlow to evaluate `x` and `y` only after `a` and `b` have been evaluated:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Obviously, since `z` depends on `x` and `y`, evaluating `z` also implies waiting for `a` and `b` to be evaluated, even though it is not explicitly in the `control_dependencies()` block. Also, since `b` depends on `a`, the code above could be simplified by just creating a control dependency on `[b]` instead of `[a, b]`, but in some cases “explicit is better than implicit”.

Great! Now you know:

- how to place operations on multiple devices in any way you please,
- how these operations get executed in parallel,
- and how to create control dependencies to optimize parallel execution.

It's time to distribute computations across multiple servers!

Multiple devices across multiple servers

To run a graph across multiple servers, you first need to define a *cluster*. A cluster is composed of one or more TensorFlow servers, called *tasks*, typically spread across several machines (see [Figure 12-6](#)). Each task belongs to a *job*. A job is just a named group of tasks that typically have a common role, for example keeping track of the model parameters (such a job is usually named "ps" for *parameter server*), or performing computations (such a job is usually named "worker").

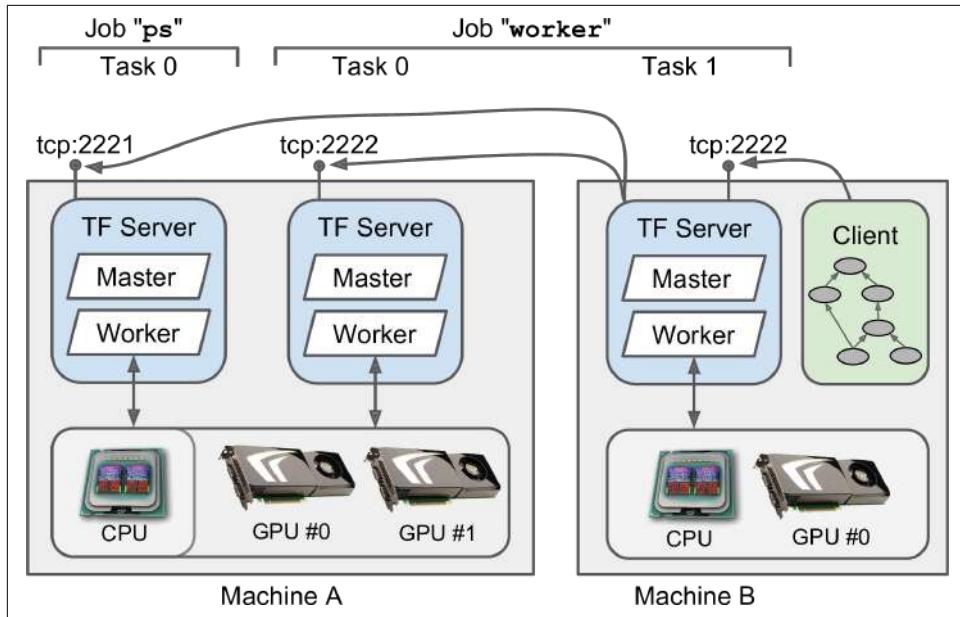


Figure 12-6. TensorFlow Cluster

The following *cluster specification* defines two jobs, "ps" and "worker", containing 1 task and 2 tasks respectively. In this example, machine A hosts two TensorFlow servers (ie. tasks), listening on different ports: one is part of the "ps" job, and the other is part of the "worker" job. Machine B just hosts one TensorFlow server, part of the "worker" job.

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221",  # /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222",  # /job:worker/task:1
    ]
})
```

To start a TensorFlow server, you must create a `Server` object, passing it the cluster specification (so it can communicate with other servers) and its own job name and task number. For example, to start the first worker task, you would run the following code on machine A:

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

It is usually simpler to just run one task per machine, but the example above demonstrates that TensorFlow allows you to run multiple tasks on the same machine if you

want². If you have several servers on one machine, you will need to ensure that they don't all try to grab all the RAM of every GPU, as explained earlier. For example, in [Figure 12-6](#) the "ps" task does not see the GPU devices, presumably its process was launched with `CUDA_VISIBLE_DEVICES=""`. Note that the CPU is shared by all tasks located on the same machine.

If you want the process to do nothing else than run the TensorFlow server, you can block the main thread by telling it to wait for the server to finish using the `join()` method (otherwise the server will be killed as soon as your main thread exits). Since there is currently no way to stop the server, this will actually block forever:

```
server.join() # blocks until the server stops (ie. never)
```

Opening a session

Once all the tasks are up and running (doing nothing yet), you can open a session on any of the servers, from a client located in any process on any machine (even from a process running one of the tasks), and use that session like a regular local session. For example:

```
a = tf.constant(1.0)
b = a + 2
c = a * 3

with tf.Session("grpc://machine-b.example.com:2222") as sess:
    print(c.eval()) # 9.0
```

This client code first creates a simple graph, then it opens a session on the TensorFlow server located on machine B (which we will call the *master*), and it instructs it to evaluate `c`. The master starts by placing the operations on the appropriate devices: in this example since we did not pin any operation on any device, the master simply places them all on its own default device, in this case Machine B's GPU device. Then it just evaluates `c` as instructed by the client, and it returns the result.

The master and worker services

The client uses the *gRPC* protocol (*Google Remote Procedure Call*) to communicate with the server. This is an efficient Open Source framework to call remote functions and get their outputs across a variety of platforms and languages³. It is based on HTTP2 which opens a connection and leaves it open during the whole session, allowing efficient bidirectional communication once the connection is established. Data is

² You can even start multiple tasks in the same process. It may be useful for tests but it is not recommended in production.

³ It is the next version of Google's internal *Stubby* service, which Google has used successfully for over a decade. See <http://grpc.io/> for more details.

transmitted in the form of *protocol buffers*, another Open Source Google technology. This is a lightweight binary data interchange format.



All servers in a TensorFlow cluster may communicate with any other server in the cluster, so make sure to open the appropriate ports on your firewall.

Every TensorFlow server provides two services: the *master service* and the *worker service*. The master service allows clients to open sessions and use them to run graphs. It coordinates the computations across tasks, relying on the worker service to actually execute computations on other tasks and get their results.

This architecture gives you a lot of flexibility. One client can connect to multiple servers by opening multiple sessions in different threads. One server can handle multiple sessions simultaneously from one or more clients. You can run one client per task (typically within the same process), or just one client to control all tasks. All options are open.

Pinning operations across tasks

You can use device blocks to pin operations on any device managed by any task, by specifying the job name, task index, device type and device index. For example, the following code pins `a` to the CPU of the first task in the "ps" job (that's the CPU on machine A), and it pins `b` to the second GPU managed by the first task of the "worker" job (that's GPU #1 on machine A). Finally, `c` is not pinned to any device, so the master places it on its own default device (machine B's GPU #0 device).

```
with tf.device("/job:ps/task:0/cpu:0")
    a = tf.constant(1.0)

with tf.device("/job:worker/task:0/gpu:1")
    b = a + 2

c = a + b
```

As earlier, if you omit the device type and index, TensorFlow will default to the task's default device: for example pinning an operation to "/job:ps/task:0" will place it on the default device of the first task of the "ps" job (machine A's CPU). If you also omit the task index (eg. "/job:ps"), TensorFlow defaults to "/task:0". If you omit the job name and the task index, TensorFlow defaults to the session's master task.

Sharding variables across multiple parameter servers

As we will see below, a common pattern when training a neural network on a distributed setup is to store the model parameters on a set of parameter servers (ie. the tasks in the "ps" job) while other tasks focus on computations (ie. the tasks in the "worker" job). For large models with millions of parameters, it is useful to shard these parameters across multiple parameter servers, to reduce the risk of saturating a single parameter server's network card. If you were to manually pin every variable to a different parameter server, it would be quite tedious. Fortunately, TensorFlow provides the `replica_device_setter()` function which distributes variables across all the "ps" tasks in a round-robin fashion. For example, the following code pins 5 variables to 2 parameter servers:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0) # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0) # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0) # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0) # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0) # pinned to /job:ps/task:0
```

Instead of passing the number of `ps_tasks`, you can pass the cluster spec `cluster=cluster_spec` and TensorFlow will simply count the number of tasks in the "ps" job.

If you create other operations in the block, other than just variables, TensorFlow automatically pins them to "/job:worker", which will default to the first device managed by the first task in the "worker" job. You can pin them to another device by setting the `worker_device` parameter, but a better approach is to use embedded device blocks. An inner device block can override the job, task or device defined in an outer block. For example:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0) # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    v2 = tf.Variable(2.0) # pinned to /job:ps/task:1 (+ defaults to /cpu:0)
    v3 = tf.Variable(3.0) # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    [...]
    s = v1 + v2          # pinned to /job:worker (+ defaults to task:0/gpu:0)
    with tf.device("/gpu:1"):
        p1 = 2 * s      # pinned to /job:worker/gpu:1 (+ defaults to /task:0)
        with tf.device("/task:1"):
            p2 = 3 * s  # pinned to /job:worker/task:1/gpu:1
```



This example assumes that the parameter servers are CPU-only, which is typically the case since they only need to store and communicate parameters, not perform intensive computations.

Sharing state across sessions using resource containers

When using a plain *local session* (not the distributed kind), each variable's state is managed by the session itself: as soon as it ends, all variable values are lost. Moreover, multiple local sessions cannot share any state, even if they both run the same graph: each session has its own copy of every variable (as we discussed in [Chapter 9](#)). In contrast, when using *distributed sessions*, variable state is managed by *resource containers* located on the cluster itself, not by the sessions. So if you create a variable named `x` using one client session, it will automatically be available to any other session on the same cluster (even if both sessions are connected to a different server). For example, consider the following client code:

```
# simple_client.py
import tensorflow as tf
import sys

x = tf.Variable(0.0, name="x")
increment_x = tf.assign(x, x + 1)

with tf.Session(sys.argv[1]) as sess:
    if sys.argv[2:]==["init"]:
        sess.run(x.initializer)
    sess.run(increment_x)
    print(x.eval())
```

Let's suppose you have a TensorFlow cluster up and running on machines A and B, port 2222, you could launch the client, have it open a session with the server on machine A and tell it to initialize the variable, increment it and print its value by launching the following command:

```
$ python3 simple_client.py grpc://machine-a.example.com:2222 init
1.0
```

Now if you launch the client with the following command, it will connect to the server on machine B and magically reuse the same variable `x` (this time we don't ask the server to initialize the variable):

```
$ python3 simple_client.py grpc://machine-b.example.com:2222
2.0
```

This feature cuts both ways: it's great if you want to share variables across multiple sessions, but if you want to run completely independent computations on the same cluster you will have to be careful not to use the same variable names by accident. One way to ensure that you won't have name clashes is to wrap all your construction phase inside a variable scope with a unique name for each computation, for example:

```
with tf.variable_scope("my_problem_1"):
    [...] # Construction phase of problem 1
```

A better option is to use a container block:

```
with tf.container("my_problem_1"):
    [...] # Construction phase of problem 1
```

This will use a container dedicated to problem #1, instead of the default one (whose name is an empty string ""). One advantage is that variable names remain nice and short. Another advantage is that you can easily reset a named container. For example, the following command will connect to the server on machine A and ask it to reset the container named "my_problem_1", which will free all the resources this container used (and also close all sessions open on the server). Any variable managed by this container must be initialized before you can use it again):

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Resource containers make it easy to share variables across sessions in flexible ways. For example [Figure 12-7](#) shows 4 clients running different graphs on the same cluster, but sharing some variables. Clients A and B share the same variable x managed by the default container, while clients C and D share another variable named x managed by the container named "my_problem_1". Note that client C even uses variables from both containers.

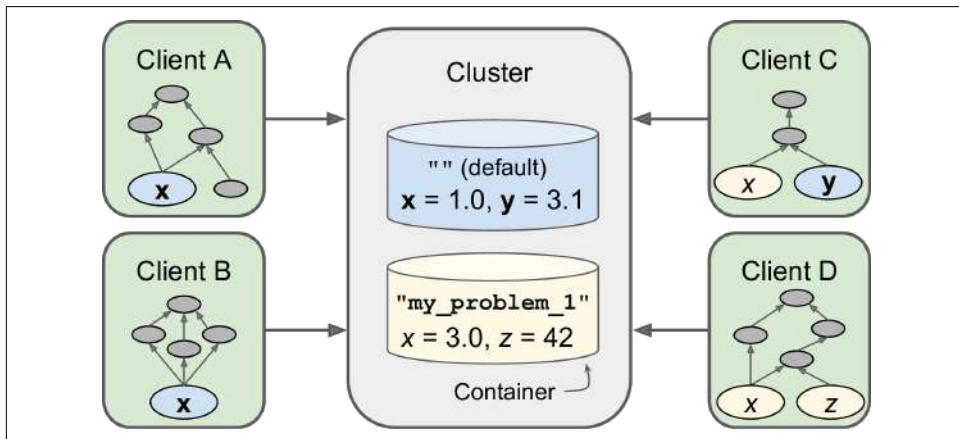


Figure 12-7. Resource containers

Resource containers also take care of preserving the state of other stateful operations, namely queues and readers. Let's take a look at queues first.

Asynchronous communication using TensorFlow queues

Queues are another great way to exchange data between multiple sessions: for example, one common use case is to have a client create a graph that loads the training data and pushes it into a queue, while another client creates a graph that pulls the data from the queue and trains a model (see [Figure 12-8](#)). This can speed up training con-

siderably because the training operations don't have to wait for the next mini-batch at every step.

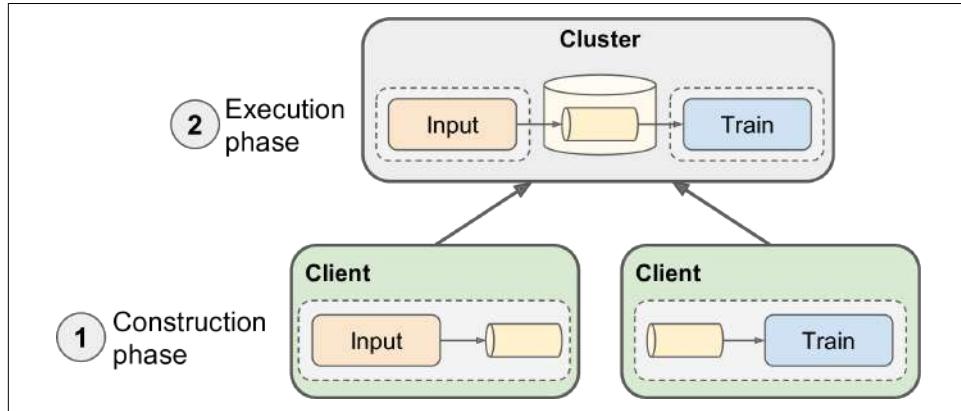


Figure 12-8. Using queues to load the training data asynchronously

TensorFlow provides various kinds of queues. The simplest kind is the *First-In First-Out (FIFO)* queue. For example, the following code creates a FIFO queue which can store up to 10 tensors containing 2 float values each:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],  
                  name="q", shared_name="shared_q")
```



To share variables across sessions, all you had to do was to specify the same name and container on both ends. With queues TensorFlow does not use the `name` attribute, it uses the `shared_name` instead, so it is important to specify it (even if it is the same as the `name`). And of course use the same container.

Enqueuing data

To push data to a queue, you must create an `enqueue` operation. For example, the following code pushes 3 training instances to the queue:

```
# training_data_loader.py  
import tensorflow as tf  
  
q = [...]  
training_instance = tf.placeholder(tf.float32, shape=(2))  
enqueue = q.enqueue([training_instance])  
  
with tf.Session("grpc://machine-a.example.com:2222") as sess:  
    sess.run(enqueue, feed_dict={training_instance: [1., 2.]})  
    sess.run(enqueue, feed_dict={training_instance: [3., 4.]})  
    sess.run(enqueue, feed_dict={training_instance: [5., 6.]})
```

Instead of enqueueing instances one by one, you can enqueue several at a time using an `enqueue_many` operation:

```
[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue_many,
              feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.]]})
```

Both examples enqueue the same 3 tensors to the queue.

Dequeuing data

To pull the instances out of the queue on the other end, you need to use a `dequeue` operation:

```
# trainer.py
import tensorflow as tf

q = [...]
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue)) # [1., 2.]
    print(sess.run(dequeue)) # [3., 4.]
    print(sess.run(dequeue)) # [5., 6.]
```

In general you will want to pull a whole mini-batch at once, instead of pulling just one instance at a time. To do so, you must use a `dequeue_many` operation, specifying the mini-batch size:

```
[...]
batch_size = 2
dequeue_mini_batch= q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue_mini_batch)) # [[1., 2.], [4., 5.]]
    print(sess.run(dequeue_mini_batch)) # blocked waiting for another instance
```

When a queue is full the `enqueue` operation will block until items are pulled out by a `dequeue` operation. Similarly, when a queue is empty (or you are using `dequeue_many()` and there are less items than the mini-batch size), the `dequeue` operation will block until enough items are pushed into the queue using an `enqueue` operation.

Queues of tuples

Each item in a queue can be a tuple of tensors (of various types and shapes) instead of just a single tensor. For example the following queue stores pairs of tensors, one of type `int32` and shape `()`, the other of type `float32` and shape `[3,2]`:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32], shapes=[[[], [3,2]], name="q", shared_name="shared_q")
```

The enqueue operation must be given pairs of tensors (note that each pair represents only one item in the queue):

```
a = tf.placeholder(tf.int32, shape=())
b = tf.placeholder(tf.float32, shape=(3, 2))
enqueue = q.enqueue((a, b))

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={a: 10, b:[[1., 2.], [3., 4.], [5., 6.]]})
    sess.run(enqueue, feed_dict={a: 11, b:[[2., 4.], [6., 8.], [0., 2.]]})
    sess.run(enqueue, feed_dict={a: 12, b:[[3., 6.], [9., 2.], [5., 8.]]})
```

On the other end, the `dequeue()` function now creates a pair of dequeue operations:

```
dequeue_a, dequeue_b = q.dequeue()
```

In general you should run these operations together:

```
with tf.Session([...]) as sess:
    a_val, b_val = sess.run([dequeue_a, dequeue_b])
    print(a_val) # 10
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```



If you run `dequeue_a` on its own, it will dequeue a pair and return only the first element: the second element will be lost (and similarly, if you run `dequeue_b` on its own, the first element will be lost).

The `dequeue_many()` function also returns a pair of operations:

```
batch_size = 2
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

You can use it as you would expect:

```
with tf.Session([...]) as sess:
    a, b = sess.run([dequeue_a, dequeue_b])
    print(a) # [10, 11]
    print(b) # [[[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0., 2.]]]
    a, b = sess.run([dequeue_a, dequeue_b]) # blocked waiting for another pair
```

Closing a queue

It is possible to close a queue to signal to the other sessions that no more data will be enqueued:

```
close_q = q.close()

with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Subsequent executions of `enqueue` or `enqueue_many` operations will raise an exception. By default any pending enqueue request will be honored, unless you call `q.close(cancel_pending_enqueues=True)`.

Subsequent executions of `dequeue` or `dequeue_many` operations will continue to succeed as long as there are items in the queue, but they will fail when there are not enough items left in the queue. If you are using a `dequeue_many` operation and there are a few instances left in the queue, but less than the mini-batch size, they will be lost. You may prefer to use a `dequeue_up_to` operation instead: it behaves exactly like `dequeue_many` except when a queue is closed and there are less than `batch_size` instances left in the queue, in which case it just returns them.

RandomShuffleQueue

TensorFlow also supports a couple more types of queues, including `RandomShuffleQueue`: a `RandomShuffleQueue` can be used just like a `FIFOQueue` except that items are dequeued in a random order. This can be useful to shuffle training instances at each epoch during training. First, let's create the queue:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                           dtypes=[tf.float32], shapes=[()],
                           name="q", shared_name="shared_q")
```

The `min_after_dequeue` specifies the minimum number of items that must remain in the queue after a `dequeue` operation. This ensures that there will be enough instances in the queue to have enough randomness (once the queue is closed, the `min_after_dequeue` limit is ignored). Now suppose that you enqueued 22 items in this queue (floats 1. to 22.), here is how you could dequeue them:

```
dequeue = q.dequeue_many(5)

with tf.Session([...]) as sess:
    print(sess.run(dequeue)) # [ 20.  15.  11.  12.   4.]  (17 items left)
    print(sess.run(dequeue)) # [  5.  13.   6.   0.  17.]  (12 items left)
    print(sess.run(dequeue)) # 12 - 5 < 10: blocked waiting for 3 more instances
```

PaddingFIFOQueue

A PaddingFIFOQueue can also be used just like a FIFOQueue except that it accepts tensors of variable sizes along any dimension (but with a fixed rank). When dequeuing them with a `dequeue_many` or `dequeue_up_to` operation, each tensor is padded with zeros along every variable dimension to make it the same size as the largest tensor in the mini-batch. For example, you could enqueue 2D tensors (matrices) of arbitrary sizes:

```
q = tf.PaddingFIFOQueue(capacity=50, dtypes=[tf.float32], shapes=[(None, None)]
                         name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]})      # 3x2
    sess.run(enqueue, feed_dict={v: [[1.]}])                                # 1x1
    sess.run(enqueue, feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) # 2x4
```

If we just dequeue one item at a time, we get the exact same tensors that were enqueued. But if we dequeue several items at a time (using `dequeue_many()` or `dequeue_up_to()`), the queue automatically pads the tensors appropriately. For example, if we dequeue all 3 items at once, all tensors will be padded with zeros to become 3×4 tensors, since the maximum size for the first dimension is 3 (first item) and the maximum size for the second dimension is 4 (third item):

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
[[[ 1.  2.  0.  0.]
 [ 3.  4.  0.  0.]
 [ 5.  6.  0.  0.]]
 
 [[ 1.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
 
 [[ 7.  8.  9.  5.]
 [ 6.  7.  8.  9.]
 [ 0.  0.  0.  0.]]]
```

This type of queue can be useful when dealing with variable length inputs, such as sequences of words (see [???](#)).

Ok let's pause for a second: so far you have learned to distribute computations across multiple devices and servers, share variables across sessions and communicate asynchronously using queues. Before you start training neural networks, there's one last topic we need to discuss: how to efficiently load training data.

Loading data directly from the graph

So far we have assumed that the clients would load the training data and feed it to the cluster using placeholders. This is simple and works quite well for simple setups, but it is rather inefficient since it transfers the training data several times:

1. From the filesystem to the client,
2. From the client to the master task,
3. Possibly from the master task to other tasks where the data is needed.

It gets worse if you have several clients training various neural networks using the same training data (for example for hyperparameter tuning): if every client loads the data simultaneously, you may end up even saturating your file server or the network's bandwidth.

Preload the data into a variable

For datasets that can fit in memory, a better option is to load the training data once and assign it to a variable, then just use that variable in your graph. This is called *preloading* the training set. This way the data will only be transferred once from the client to the cluster (but it may still need to be moved around from task to task depending on which operations need it). The following code shows how to load the full training set into a variable:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False, collections=[],
                         name="training_set")

with tf.Session([...]) as sess:
    data = [...] # load the training data from the datastore
    sess.run(training_set.initializer, feed_dict={training_set_init: data})
```

You must set `trainable=False` so the optimizers don't try to tweak this variable. You should also set `collections=[]` to ensure that this variable won't get added to the `GraphKeys.VARIABLES` collection which is used for saving and restoring checkpoints.



This example assumes that all your training set (including the labels) consists only of `float32` values. If that's not the case, you will need one variable per type.

Reading the training data directly from the graph

If the training set does not fit in memory, a good solution is to use *reader operations*: these are operations capable of reading data directly from the file system. This way

the training data never needs to flow through the clients at all. TensorFlow provides readers for various file formats:

- CSV
- Fixed length binary records
- TensorFlow's own `TFRecords` format, based on protocol buffers.

Let's look at a simple example reading from a CSV file (for other formats, please check out the API documentation). Suppose you have file named `my_test.csv` that contains training instances, and you want to create operations to read it. Suppose it has the following content, with two float features `x1` and `x2` and one integer `target` representing a binary class:

```
x1, x2, target
1. , 2. , 0
4. , 5. , 1
7. , , 0
```

First, let's create a `TextLineReader` to read this file. A `TextLineReader` opens a file (once we tell it which one to open) and it reads lines one by one. It is a stateful operation, like variables and queues: it preserves its state across multiple runs of the graph, keeping track of which file it is currently reading and what is its current position in this file.

```
reader = tf.TextLineReader(skip_header_lines=1)
```

Next, we create a queue that the reader will pull from to know which file to read next. We also create an enqueue operation and a placeholder to push any filename we want to the queue, and we create an operation to close the queue once we have no more files to read.

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
```

Now we are ready to create a `read` operation that will read one record (ie. a line) at a time and return a key/value pair. The key is the record's unique identifier (a string composed of the file name, a colon ":" and the line number), and the value is simply a string containing the content of the line:

```
key, value = reader.read(filename_queue)
```

We have all we need to read the file line by line! But we are not quite done yet: we need to parse this string to get the features and target:

```
x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1.]])
features = tf.pack([x1, x2])
```

The first line uses TensorFlow's CSV parser to extract the values from the current line. The default values are used when a field is missing (in this example the third training instance's x2 feature), and they are also used to determine the type of each field (in this case two floats and one integer).

Finally, we can push this training instance and its target to a `RandomShuffleQueue` that we will share with the training graph (so it can pull mini-batches from it), and we create an operation to close that queue when we are done pushing instances to it.

```
instance_queue = tf.RandomShuffleQueue(  
    capacity=10, min_after_dequeue=2,  
    dtypes=[tf.float32, tf.int32], shapes=[[2],[[]]],  
    name="instance_q", shared_name="shared_instance_q")  
enqueue_instance = instance_queue.enqueue([features, target])  
close_instance_queue = instance_queue.close()
```

Wow! That was a lot of work just to read a file. Plus we only created the graph, now we need to run it:

```
with tf.Session([...]) as sess:  
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})  
    sess.run(close_filename_queue)  
    try:  
        while True:  
            sess.run(enqueue_instance)  
    except tf.errors.OutOfRangeError as ex:  
        pass # no more records in the current file and no more files to read  
    sess.run(close_instance_queue)
```

First we open the session, then we enqueue the filename "my_test.csv" and we immediately close that queue since we will not enqueue any more filenames. Then we run an infinite loop to enqueue instances one by one. The `enqueue_instance` depends on reader reading the next line, so at every iteration a new record is read until it reaches the end of the file. At that point it tries to read the filename queue to know which file to read next, and since the queue is closed it throws an `OutOfRangeError` exception (if we did not close the queue, it would just remain blocked until we pushed another filename or closed the queue). Lastly we close the instance queue so that the training operations pulling from it won't get blocked forever. [Figure 12-9](#) summarizes what we have learned: it represents a typical graph for reading training instances from a set of CSV files.

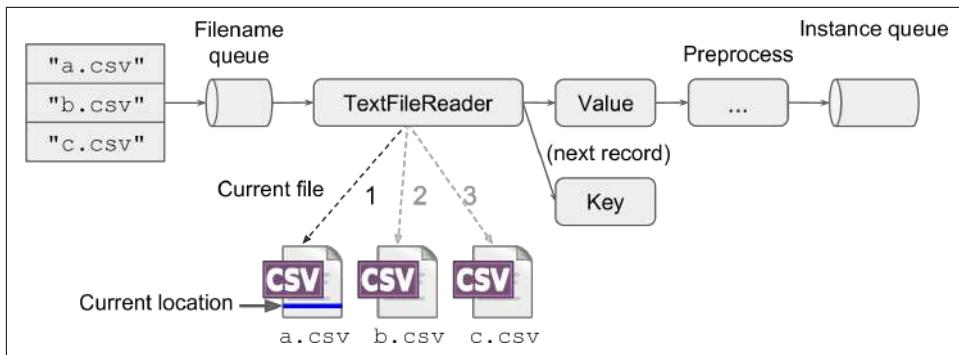


Figure 12-9. A graph dedicated to reading training instances from CSV files

In the training graph, you need to create the shared instance queue and simply dequeue mini-batches from it:

```

instance_queue = tf.RandomShuffleQueue(..., shared_name="shared_instance_q")
mini_batch_instances, mini_batch_targets = instance_queue.dequeue_up_to(2)
[...] # use the mini_batch instances and targets to build the training graph
training_op = [...]

with tf.Session(...) as sess:
    try:
        for step in range(max_steps):
            sess.run(training_op)
    except tf.errors.OutOfRangeError as ex:
        pass # no more training instances

```

In this example, the first mini-batch will contain the first two instances of the CSV file, and the second mini-batch will contain the last instance.



TensorFlow queues don't handle sparse tensors well, so your training instances are sparse you should parse the records after the instance queue.

This architecture will only use one thread to read records and push them to the instance queue. You can get a much higher throughput by having multiple threads read simultaneously from multiple files using multiple readers. Let's see how.

Multithreaded readers using a Coordinator and a QueueRunner

To have multiple threads read instances simultaneously, you could create python threads (using the `threading` module) and manage them yourself. However, TensorFlow provides some tools to make this simpler: the `Coordinator` class and the `QueueRunner` class.

A Coordinator is a very simple object whose sole purpose is to coordinate stopping multiple threads. First you create a coordinator:

```
coord = tf.train.Coordinator()
```

Then you give it to all threads that need to stop jointly, and their main loop looks like this:

```
while not coord.should_stop():
    [...] # do something
```

Any thread can request that every thread stop by calling the coordinator's `request_stop()` method:

```
coord.request_stop()
```

Every thread will stop as soon as they reach they finish their current iteration. You can wait for all of them to finish by calling the coordinator's `join()` method, passing it the list of threads:

```
coord.join(list_of_threads)
```

A QueueRunner runs multiple threads that each run an enqueue operation repeatedly, filling up a queue as fast as possible. As soon as the queue is closed, the next thread that tries to push an item to the queue will get an `OutOfRangeError`: it catches it and immediately tells other threads to stop using a Coordinator. The following code shows how you can use a QueueRunner to have 5 threads reading instances simultaneously and pushing them to an instance queue:

```
[...] # same construction phase as earlier
queue_runner = tf.train.QueueRunner(instance_queue, [enqueue_instance] * 5)

with tf.Session() as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    coord = tf.train.Coordinator()
    enqueue_threads = queue_runner.create_threads(sess, coord=coord, start=True)
```

The first line creates the QueueRunner and tells it to run 5 threads, all running the same `enqueue_instance` operation repeatedly. Then we start a session and we enqueue the name of the files to read (in this case just "my_test.csv"). Next we create a Coordinator that the QueueRunner will use to stop gracefully, as explained above. Finally, we tell the QueueRunner to create the threads and start them. They will read all training instances and push them to the instance queue, then they will all stop gracefully.

This will be a bit more efficient than earlier, but we can do better: currently all threads are reading from the same file. We can make them read simultaneously from separate files instead (assuming the training data is sharded across multiple CSV files) by creating multiple readers (see Figure 12-10).

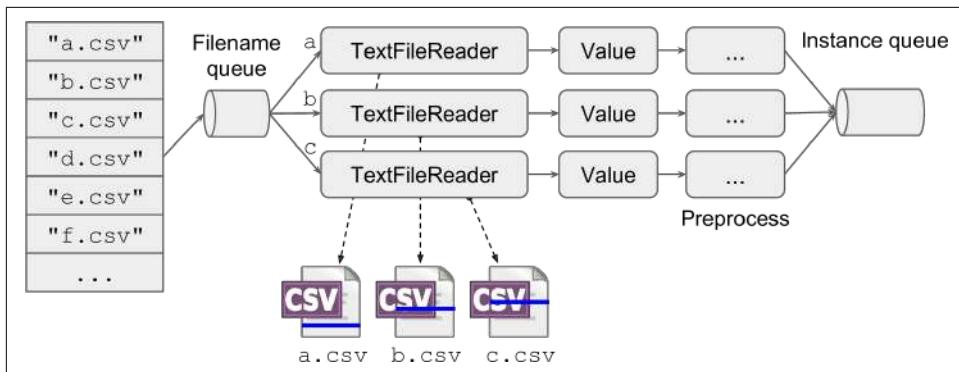


Figure 12-10. Reading simultaneously from multiple files

For this we need to write a small function to create a reader and the nodes that will read and push one instance to the instance queue:

```
def read_and_push_instance(filename_queue, instance_queue):
    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)
    x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1.]])
    features = tf.pack([x1, x2])
    enqueue_instance = instance_queue.enqueue([features, target])
    return enqueue_instance
```

Next we define the queues:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()

instance_queue = tf.RandomShuffleQueue([...])
```

And finally we create the QueueRunner, but this time we give it a list of different enqueue operations: each operation will use a different reader, so the threads will simultaneously read from different files:

```
read_and_enqueue_ops = [
    read_and_push_instance(filename_queue, instance_queue)
    for i in range(5)]
queue_runner = tf.train.QueueRunner(instance_queue, read_and_enqueue_ops)
```

The execution phase is then the same as above: first push the names of the files to read, then create a Coordinator and create and start the QueueRunner threads. This time all threads will read from different files simultaneously until all files are read entirely, then the QueueRunner will close the instance queue so that other ops pulling from it don't get blocked.

Other convenience functions

TensorFlow also offers a few convenience functions to simplify some common tasks when reading training instances. Here are just a few (see the API documentation for the full list).

The `string_input_producer()` takes a 1D tensor containing a list of filenames, and it creates a thread that pushes one filename at a time to the filename queue, then it closes the queue. If you specify a number of epochs, it will cycle through the filenames once per epoch before closing the queue. By default, it shuffles the filenames at each epoch. It creates a `QueueRunner` to manage its thread, and it adds it to the `Graph Keys.QUEUE_RUNNER` collection. To start every `QueueRunner` in that collection, you can call the `tf.train.start_queue_runners()` function. Note that if you forget to start the `QueueRunner`, the filename queue will be open and empty, and your readers will be blocked forever.

There are a few other *producer* functions that similarly create a queue and a corresponding `QueueRunner` for running an enqueue operation (eg. `input_producer()`, `range_input_producer()`, `slice_input_producer()`).

The `shuffle_batch()` function takes a list of tensors (eg. `[features, target]`) and creates:

- a `RandomShuffleQueue`,
- a `QueueRunner` to enqueue the tensors to the queue (added to the `Graph Keys.QUEUE_RUNNER` collection),
- a `dequeue_many` operation to extract a mini-batch from the queue.

This makes it easy to manage in a single process a multithreaded input pipeline feeding a queue and a training pipeline reading mini-batches from that queue. Also checkout the `batch()` and `batch_join()` and `shuffle_batch_join()` functions that provide similar functionality.

Ok! You now have all the tools you need to start training and running neural networks efficiently across multiple devices and servers on a TensorFlow cluster. Let's review what you have learned:

- using multiple GPU devices,
- setting up and starting a TensorFlow cluster,
- distributing computations across multiple devices and servers,
- sharing variables (and other stateful ops such as queues and readers) across sessions using containers,
- coordinating multiple graphs working asynchronously using queues,

- reading inputs efficiently using readers, queue runners and coordinators.

Now let's use all of this to parallelize neural networks!

Parallelizing neural networks on a TensorFlow cluster

First we will look at how to parallelize several neural networks by simply placing each one on a different device, then we will look at the much trickier problem of training a single neural network across multiple devices and servers.

One neural network per device

The most trivial way to train and run neural networks on a TensorFlow cluster is to take the exact same code you would use for a single device on a single machine, specify the master server's address when creating the session, and you're done! Your code will be running on the server's default device. You can change the device that will run your graph simply by putting your code's construction phase within a device block.

By running several client sessions in parallel (in different threads or different processes), connecting them to different servers and configuring them to use different devices, you can quite easily train or run many neural networks in parallel, across all devices and all machines in your cluster (see [Figure 12-11](#)). The speedup is almost linear⁴: training 100 neural networks across 50 servers with 2 GPUs each will not take much longer than training just 1 neural network on 1 GPU.

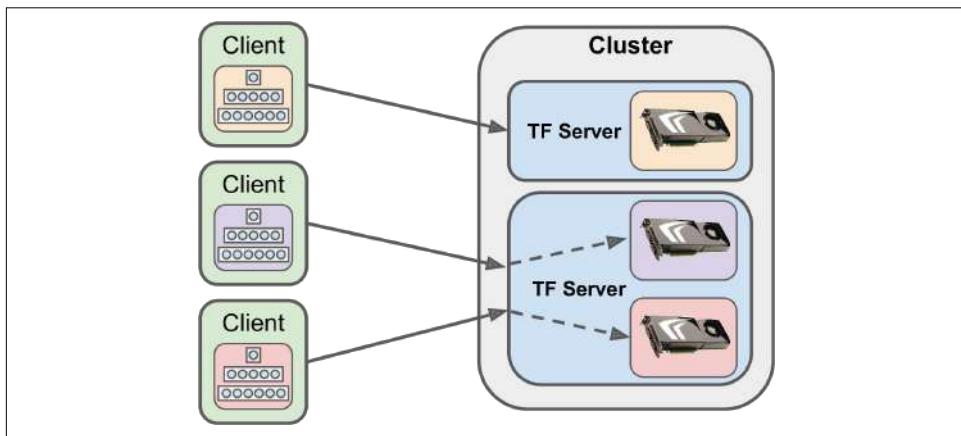


Figure 12-11. Training one neural network per device

⁴ Not 100% linear if you wait for all devices to finish, since the total time will be the time taken by the slowest device.

This solution is perfect for hyperparameter tuning: each device in the cluster will train a different model with its own set of hyperparameters. The more computing power you have, the larger the hyperparameter space you can explore.

It also works perfectly if you host a web service that receives a large number of *Queries Per Second* (QPS) and you need your neural network to make a prediction for each query. Simply replicate the neural network across all devices on the cluster and dispatch queries across all devices. By adding more servers you can handle an unlimited number of QPS (however this will not reduce the time it takes to process a single request since it will still have to wait for a neural network to make a prediction).



Another option is to serve your neural networks using *TensorFlow Serving*. It is an Open Source system, released by Google in February 2016, designed to serve a high volume of queries to Machine Learning models (typically built with TensorFlow). It handles model versioning, so you can easily deploy a new version of your network to production, or experiment with various algorithms without interrupting your service, and it can sustain a heavy load by adding more servers. For more details, check out <https://tensorflow.github.io/serving/>.

In-graph vs between-graph replication

You can also parallelize the training of a large ensemble of neural networks by simply placing every neural network on a different device (ensembles were introduced in [Chapter 7](#)). However, once you want to *run* the ensemble, you will need to aggregate the individual predictions made by each neural network to produce the ensemble's prediction: this requires a bit of coordination.

There are two major approaches to handling a neural network ensemble (or any other graph that contains large chunks of independent computations):

- You can create one big graph, containing every neural network, each pinned to a different device, plus the computations needed to aggregate the individual predictions from all the neural networks (see [Figure 12-12](#)). Then you just create one session to any server in the cluster and let it take care of everything (including waiting for all individual predictions to be available before aggregating them): this approach is called *in-graph replication*.

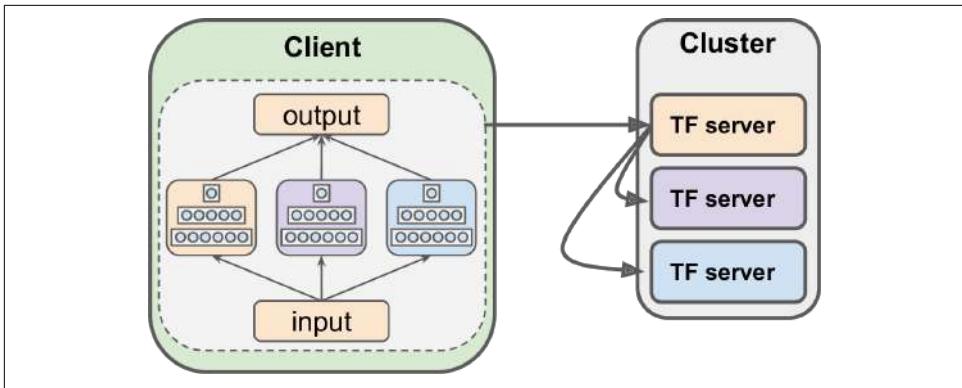


Figure 12-12. In-graph replication

- Alternatively, you can create one separate graph for each neural network and handle synchronization between these graphs yourself. This approach is called *between-graph replication*. One typical implementation is to coordinate the execution of these graphs using queues (see [Figure 12-13](#)). A set of clients handle one neural network each, reading from its dedicated input queue, and writing to its dedicated prediction queue. Another client is in charge of reading the inputs and pushing them to all the input queues (copying all inputs to every queue). Finally, one last client is in charge of reading one prediction from each prediction queue and aggregate them to produce the ensemble's prediction.

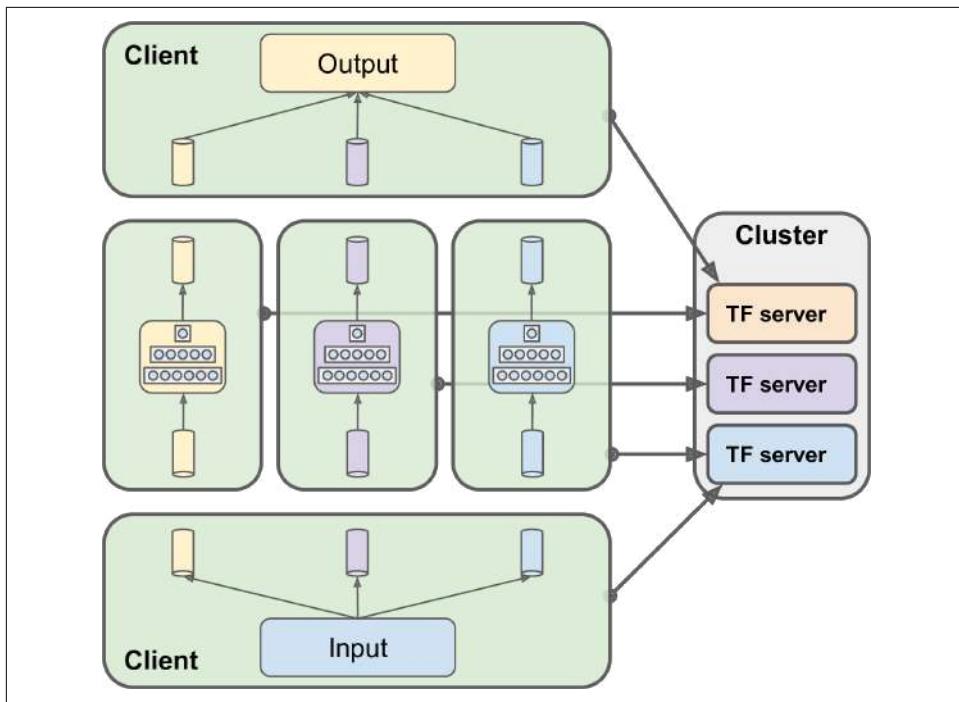


Figure 12-13. Between-graph replication

These solutions have their pros and cons. In-graph replication is somewhat simpler to implement since you don't have to manage multiple clients and multiple queues. However, between-graph replication is a bit easier to organize into well bounded and easy to test modules. Moreover, it gives you more flexibility: for example you could add a dequeue timeout in the aggregator client so that the ensemble would not fail even if one of the neural network clients crashes or if one neural network takes too long to produce its prediction. TensorFlow lets you specify a timeout when calling the `run()` function by passing a `RunOptions` with `timeout_in_ms`:

```
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000 # 1s timeout
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # the dequeue operation timed out after 1s
```

Another way you can specify a timeout is to set the session's `operation_timeout_in_ms` configuration option, but in this case the `run()` function times out if *any* operation takes longer than the timeout delay:

```

config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000 # 1s timeout for every operation

with tf.Session([...], config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
    except tf.errors.DeadlineExceededError as ex:
        [...] # the dequeue operation timed out after 1s

```

Model parallelism

So far we have run each neural network on a single device. What if we want to run a single neural network across multiple devices? This requires chopping your model into separate chunks and running each chunk on a different device. This is called *model parallelism*. Unfortunately, it turns out to be pretty tricky and it really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained with model parallelism (see Figure 12-14). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work since each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically, for example with the left half of each layer on one device, and the right part on another device? This is slightly better since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (represented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (especially if it is across separate machines).

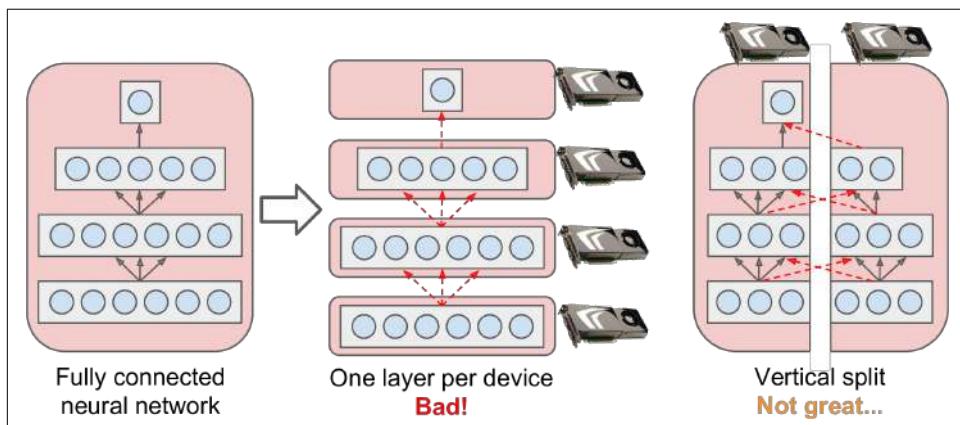


Figure 12-14. Splitting a fully connected neural network

However, as we will see in ???, some neural network architectures such as Convolutional Neural Networks contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way.

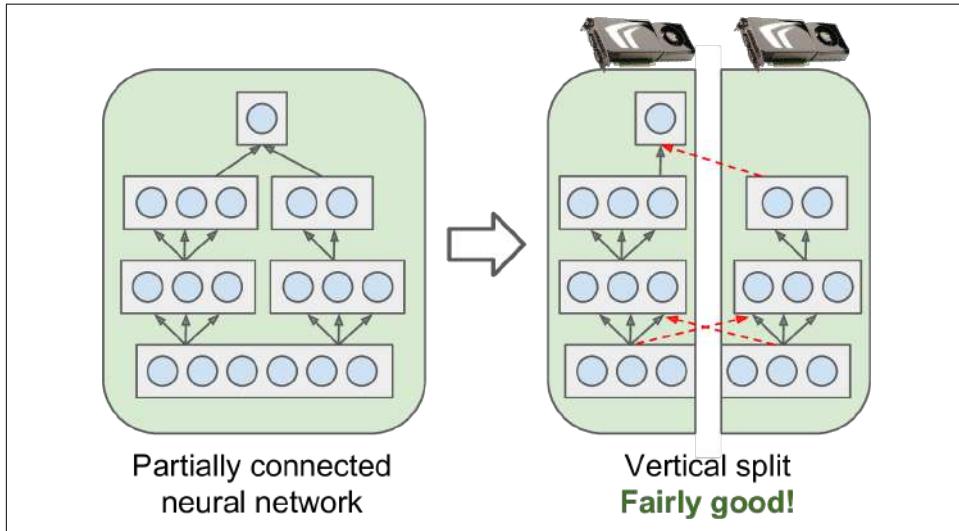


Figure 12-15. Splitting a partially connected neural network

Moreover, as we will see in ???, some deep Recurrent Neural Networks are composed of several layers of *memory cells* (see the left side of Figure 12-16). A cell's output at time t is fed back to its input at time $t + 1$ (as you can see more clearly on the right side of the diagram). If you split such a network horizontally, placing each layer on a different device, then at the first step only one device will be active, at the second step two will be active, and by the time the signal propagates to the output layer all devices will be active simultaneously. There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel often outweighs the communication penalty.

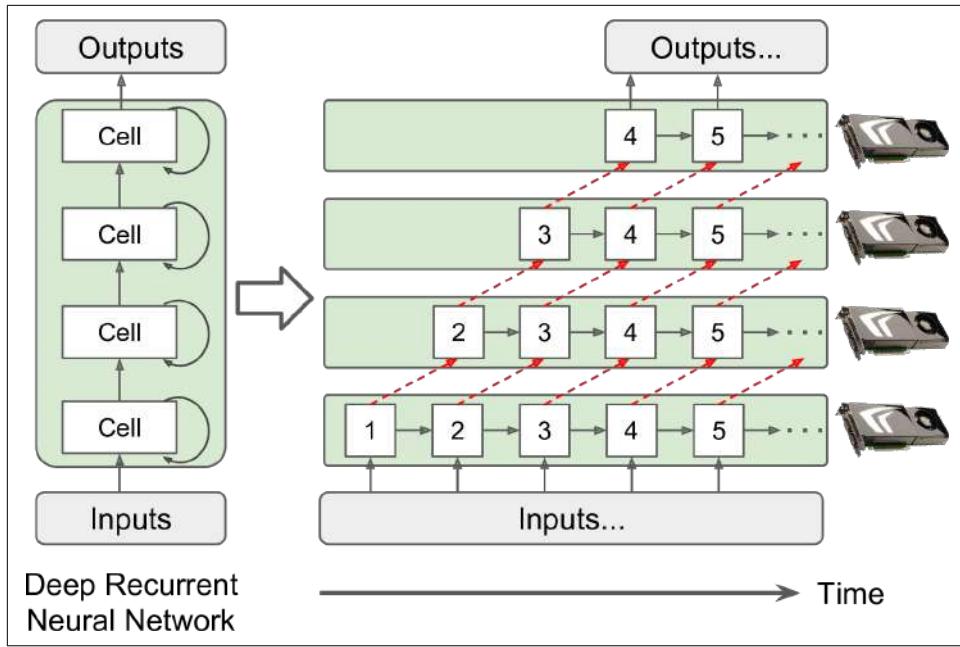


Figure 12-16. Splitting a deep recurrent neural network

In short model parallelism can speed up running or training some types of neural networks, but not all, and it requires special care and tuning such as making sure that devices that need to communicate the most run on the same machine.

Data parallelism

Another way to parallelize the training of a neural network is to replicate it on each device, run a training step simultaneously on all replicas using a different mini-batch for each, then aggregate the gradients to update the model parameters. This is called *data parallelism* (see Figure 12-17).

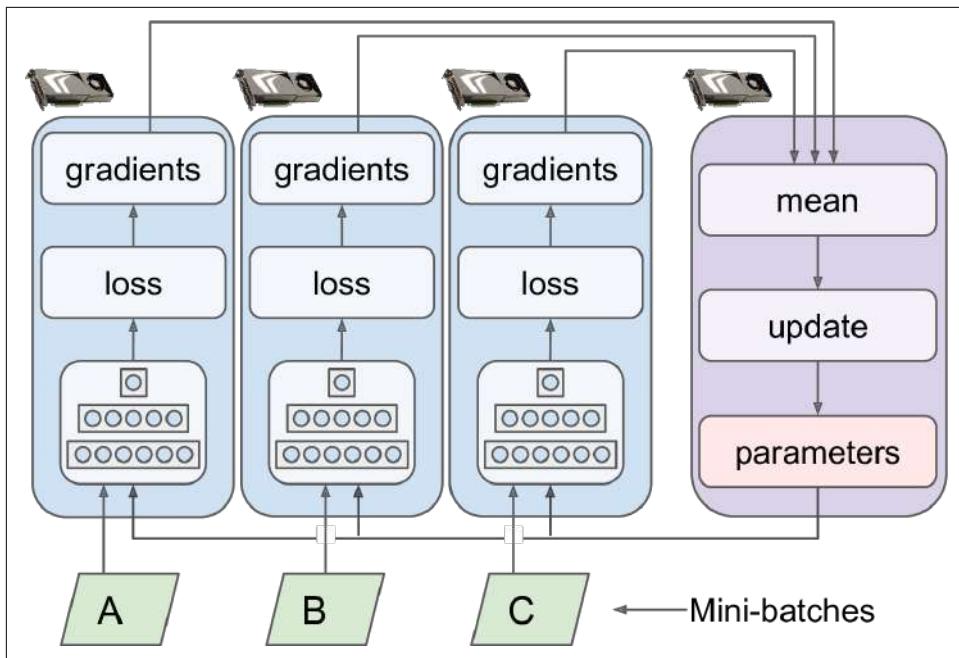


Figure 12-17. Data parallelism

There are two variants of this approach: *synchronous updates* and *asynchronous updates*.

Synchronous updates

With *synchronous updates*, the aggregator waits for all gradients to be available before computing the average and applying the result (ie. using the aggregated gradients to update the model parameters). Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so all other devices will have to wait for them at every step. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.



To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically $\sim 10\%$). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.⁵

Asynchronous updates

With asynchronous updates, whenever a replica has finished computing the gradients, it immediately uses them to update the model parameters. There is no aggregation (remove the “mean” step on [Figure 12-17](#)), and no synchronization. Replicas just work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice, because of its simplicity, the absence of synchronization delay and a better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$) times if there are N replicas) and there is no guarantee that the computed gradients will still be pointing in the right direction (see [Figure 12-18](#)). When gradients are severely out-of-date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge.

⁵ This name is slightly confusing since it sounds like some replicas are special, doing nothing. In reality all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others).

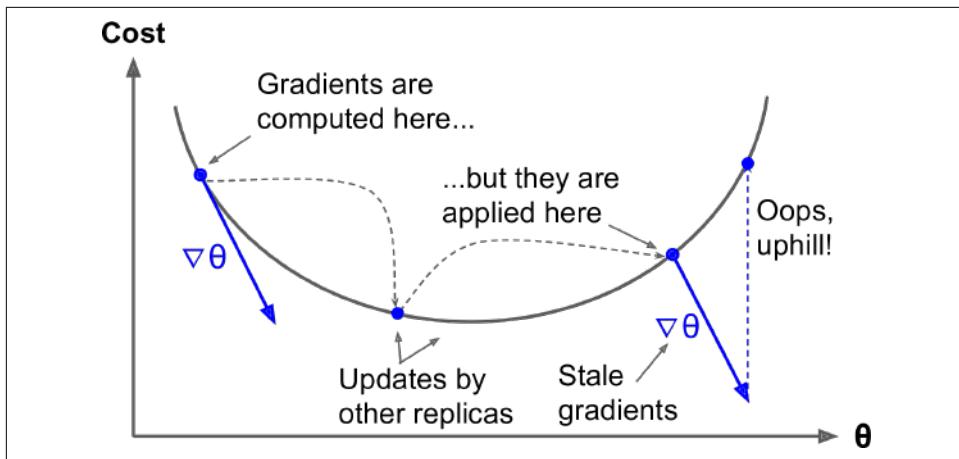


Figure 12-18. Stale gradients when using asynchronous updates

There are a few ways to reduce the effect of stale gradients:

- reduce the learning rate,
- drop stale gradients or scale them down,
- adjust the mini-batch size,
- start the first few epochs using just one replica (this is called the *warmup phase*): stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper⁶ published by the Google Brain team in April 2016 benchmarked various approaches and found that data parallelism with synchronous updates using a few spare replicas was the most efficient, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates quite yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism still requires communicating the model parameters from the parameter servers to every replica at the beginning of every training step, and the gradients in the other direction at the end of each training step. Unfortunately this means that there always comes a point where adding an extra GPU will not improve performance at all because the

⁶ “Revisiting Distributed Synchronous SGD”, Jianmin Cheng *et al.* (2016): <http://goo.gl/9GCiPb>

time spent moving the data in and out of GPU RAM (and possibly across the network) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just increase saturation and slow down training.



For some models, typically relatively small and trained on a very large training set, you are often better off training the model on a single machine with a single GPU.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is small) and also for large sparse models since the gradients are typically mostly zeros, so they can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, reported⁷ typical speed ups of 25-40x when distributing computations across 50 GPUs for dense models, and 300x speedup for sparser models trained across 500 GPUs. As you can see sparse models really do scale better. Here are a few concrete examples:

- Neural Machine Translation: 6x speedup on 8 GPUs
- Inception/ImageNet: 32x speedup on 50 GPUs
- RankBrain: 300x speedup on 500 GPUs

These numbers represent the state of the art in Q1 2016: beyond a few dozens of GPUs for a dense model or few hundred GPUs for a sparse model, saturation kicks in and performance degrades. There is plenty of research going on to solve this problem (exploring peer to peer architectures rather than centralized parameter servers, using lossy model compression, optimizing when and what the replicas need to communicate, and so on), so there will likely be a lot of progress in parallelizing neural networks in the next few years.

In the meantime, here are a few simple steps you can take to reduce the saturation problem:

- Group your GPUs on few servers rather than scattering them across many servers. This will avoid unnecessary network hops.
- Shard the parameters across multiple parameter servers (as discussed earlier).

⁷ See this great presentation: <http://goo.gl/E4ypxo>

- Drop the model parameters' float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, without much impact on the convergence rate or the model's performance.



Although 16 bit precision is the minimum for training neural network, you can actually drop down to 8 bit precision after training to reduce the size of the model and speed up computations. This is called *quantizing* the neural network. It is particularly useful for deploying and running pretrained models on mobile phones. See this great post by Pete Warden: <http://goo.gl/09Cb6v>.

TensorFlow implementation

To implement data parallelism using TensorFlow, you first need to choose whether you want in-graph replication or between-graph replication, and whether you want synchronous updates or asynchronous updates. Let's look at how you would implement each combination (see the exercises and the Jupyter notebooks for complete code examples).

With in-graph replication + synchronous updates, you build one big graph containing all the model replicas (placed on different devices), and a few nodes to aggregate all their gradients and feed them to an optimizer. Your code opens a session to the cluster and simply runs the training operation repeatedly.

With in-graph replication + asynchronous updates, you also create one big graph, but with one optimizer per replica, and you run one thread per replica, repeatedly running the replica's optimizer.

With between-graph replication + asynchronous updates, you run multiple independent clients (typically in separate processes), each training the model replica as if it were alone in the world, but the parameters are actually shared with other replicas (using a resource container).

With between-graph replication + synchronous updates, once again you run multiple clients, each training a model replica based on shared parameters, but this time you wrap the optimizer (eg. a `MomentumOptimizer`) within a `SyncReplicasOptimizer`. Each replica uses this optimizer as it would use any other optimizer, but under the hood this optimizer sends the gradients to a set of queues (one per variable) which is read by one of the replica's `SyncReplicasOptimizer`, called the *chief*. The chief aggregates the gradients and applies them, then writes a token to a *token queue* for each replica, signaling them that they can go ahead and compute the next gradients. This approach supports having *spare replicas*.

If you go through the exercises, you will implement each of these four solutions. You will easily be able to apply what you have learned to train large deep neural networks

across dozens of servers and GPUs! In the following chapters we will go through a few more important neural network architectures before we tackle reinforcement learning.

Exercises

1. If you get a CUDA_ERROR_OUT_OF_MEMORY when starting your TensorFlow program, what is probably going on? What can you do about it?
2. What is the difference between pinning an operation on a device and placing an operation on a device?
3. If you are running on a GPU-enabled TensorFlow installation, and you just use the default placement, will all operations be placed on the first GPU?
4. If you pin a Variable to "/gpu:0", can it be used by operations placed on /gpu:1? Or by operations placed on "/cpu:0"? Or by operations pinned to devices located on other servers?
5. Can two operations placed on the same device run in parallel?
6. What is a control dependency and when would you want to use one?
7. Suppose you train a DNN for days on a TensorFlow cluster, and immediately after your training program ends you realize that you forgot to save the model using a `Saver`. Is your trained model lost?
8. Train several DNNs in parallel on a TensorFlow cluster, using different hyperparameter values. This could be DNNs for MNIST classification or any other task you are interested in. The simplest option is to write a single client program that trains only one DNN, then run this program in multiple processes in parallel, with different hyperparameter values for each client. The program should have command line options to control what server and device the DNN should be placed on, and what resource container and hyperparameter values to use (make sure to use a different resource container for each DNN). Use a validation set or cross validation to select the top 3 models.
9. Create an ensemble using the top 3 models from the previous exercise. Define it in a single graph, ensuring that each DNN runs on a different device. Evaluate it on the validation set: does the ensemble perform better than the individual DNNs?
10. Train a DNN using between-graph replication and data parallelism with asynchronous updates, timing how long it takes to reach a satisfying performance. Next, try again using synchronous updates. Do synchronous updates produce a better model? Is training faster? Split the DNN vertically and place each vertical

slice on a different device, and train the model again: is training any faster? Is the performance any different?

Solutions to these exercises are available in [Appendix A](#).

Exercise solutions



Solutions to the coding exercises are available in the online Jupyter notebooks at <https://github.com/ageron/handson-ml>.

Chapter 1

1. Machine Learning is about building systems that can learn from data. Learning means getting better at some task, given some performance measure.
2. Machine Learning is great for complex problems for which we have no algorithmic solution, or to replace long lists of hand-tuned rules, as well as to build systems that adapt to fluctuating environments, and finally to help humans learn (eg. data mining).
3. A labeled training set is a training set that contains the desired solution (a.k.a. a label) for each instance.
4. The two most common supervised tasks are regression and classification.
5. Common unsupervised tasks include clustering, visualization, dimensionality reduction and association rule learning.
6. Reinforcement learning is likely to perform best if we want a robot to learn to walk in various unknown terrains since this is typically the type of problems that reinforcement learning tackles. It might be possible to express the problem as a supervised or semi-supervised learning problem, but it would be less natural.
7. If you don't know how to define the groups then you can use a clustering algorithm (unsupervised learning) to segment your customers into clusters of similar customers. However, if you know what groups you would like to have, then you

can feed many examples of each group to a classification algorithm (supervised learning) and it will classify all your customers into these groups.

8. Spam detection is a typical supervised learning problem: the algorithm is fed with many emails along with their label (spam or not spam).
9. An online learning system can learn incrementally, as opposed to a batch learning system. This makes it capable of adapting rapidly to changing data, autonomous systems, as well as for training on very large quantities of data.
10. Out-of-core algorithms can handle vast quantities of data that cannot fit in a computer's main memory. An out-of-core learning algorithm chops the data into mini-batches and uses online learning techniques to learn from these mini-batches.
11. An instance based learning system learns the training data by heart then when given a new instance it uses a similarity measure to find the most similar learned instances and uses them to make predictions.
12. A model has one or more model parameters that determine what it will predict given a new instance (eg. the slope of a linear model). A learning algorithm tries to find optimal values for these parameters such that the model generalizes well to new instances. A hyperparameter is a parameter of the learning algorithm itself, not of the model (eg. the amount of regularization to apply).
13. Model based learning algorithms search for an optimal value for the model parameters such that the model will generalize well to new instances. Such systems are usually trained by minimizing a cost function that measures how bad the system is at making predictions on the training data, plus a penalty for model complexity if the model is regularized. To make predictions, the new instance's features are fed into the model's prediction function, using the parameter values found by the learning algorithm.
14. Some of the main challenges in Machine Learning are the lack of data, poor data quality, non-representative data, uninformative features, excessively simple models that underfit the training data and excessively complex models that overfit the data.
15. If a model performs great on the training data but generalizes poorly to new instances, the model is likely overfitting the training data (or we got extremely lucky on the training data). Possible solutions to overfitting are: get more data, simplify the model (select a simpler algorithm, or reduce the number of parameters or features used, or regularize the model), reduce the noise in the training data.
16. A test set is used to estimate the generalization error that a model will make on new instances, before launching the model in production.

17. A validation set is used to compare models. It makes it possible to select the best model and tune the hyperparameters.
18. If you tune hyperparameters using the test set, you risk overfitting the test set, and the generalization error you measure will be optimistic (you may launch a model that performs worse than you expect).
19. Cross-validation is a technique that makes it possible to compare models (for model selection and hyperparameter tuning) without the need for a separate validation set. This saves precious training data.

Chapter 2 and Chapter 3

See the Jupyter notebooks available at <https://github.com/ageron/handson-ml>.

Chapter 4

1. If you have a training set with millions of features you can use Stochastic Gradient Descent or Mini-batch Gradient Descent, and perhaps Batch Gradient Descent if the training set fits in memory. But not the Normal Equation because the computational complexity grows quickly with the number of features (more than quadratically).
2. If the features in your training set have very different scales, the cost function will have the shape of an elongated bowl, so the Gradient Descent algorithms will take a long time to converge. To solve this you should scale the data before training the model. Note that the Normal Equation will work just fine without scaling.
3. Gradient Descent cannot get stuck in a local minimum when training a Logistic Regression model because the cost function is convex¹.
4. If the optimization problem is convex (such as Linear Regression or Logistic Regression), and assuming the learning rate is not too high, then all Gradient Descent algorithms will approach the global optimum and end up producing fairly similar models. However, unless you gradually reduce the learning rate, Stochastic GD and Mini-batch GD will never truly converge: instead, they will keep jumping back and forth around the global optimum. This means that even if you let them run for a very long time, these Gradient Descent algorithms will produce slightly different models.
5. If the validation error consistently goes up after every epoch, then one possibility is that the learning rate is too high and the algorithm is diverging. If the training

¹ If you draw a straight line between any two points on the curve, the line never crosses the curve.

error also goes up, then this is clearly the problem and you should reduce the learning rate. However, if the training error is not going up, then your model is overfitting the training set and you should stop training.

6. Due to their random nature, both Stochastic Gradient Descent and Mini-batch Gradient Descent are not guaranteed to make progress at every single training iteration. So if you immediately stop training when the validation error goes up, you may stop much too early, before the optimum is reached. A better option is to save the model at regular intervals, and when it has not improved for a long time (meaning it will probably never beat the record), you can revert to the best saved model.
7. Stochastic Gradient Descent has the fastest training iteration since it only considers one training instance at a time, so it is generally the first to reach the vicinity of the global optimum (or Mini-batch GD with a very small mini-batch size). However, only Batch Gradient Descent will actually converge, given enough training time. As mentioned above, Stochastic GD and Mini-batch GD will bounce around the optimum, unless you gradually reduce the learning rate.
8. If the validation error is much higher than the training error, this is likely because your model is overfitting the training set. One way to try to fix this is to reduce the polynomial degree: a model with less degrees of freedom is less likely to overfit. Another thing you can try is to regularize the model, for example by adding an ℓ_2 penalty (Ridge) or an ℓ_1 penalty (Lasso) to the cost function: this will also reduce the degrees of freedom of the model. Lastly, you can try to increase the size of the training set.
9. If both the training error and the validation error are almost equal and fairly high, the model is likely underfitting the training set, which means it has a high bias. You should try reducing the regularization hyperparameter α .
10. Let's see:
 - A model with some regularization typically performs better than a model without any regularization, so you should generally prefer Ridge Regression over plain Linear Regression.²
 - Lasso regression uses an ℓ_1 penalty which tends to push the weights down to exactly zero. This leads to sparse models, where all weights are zero except for the most important weights. This is a way to perform feature selection automatically, which is good if you suspect that only few features actually matter. When you are not sure, you should prefer Ridge regression.

² Moreover, the Normal Equation requires computing the inverse of a matrix, but that matrix is not always invertible. In contrast, the matrix for Ridge regression is always invertible.

- Elastic Net is generally preferred over Lasso since Lasso may behave erratically in some cases (when several features are strongly correlated or when there are more features than training instances). However, it does add an extra hyper-parameter to tune. If you just want Lasso without the erratic behavior, you can just use Elastic Net with an `l1_ratio` close to 1.
11. If you want to classify pictures as outdoor/indoor and daytime/nighttime, since these are not exclusive classes (ie. all 4 combinations are possible) you should train two Logistic Regression classifiers.
 12. See the Jupyter notebooks available at <https://github.com/ageron/handson-ml>.

Chapter 5

1. The fundamental idea behind Support Vector Machines is to fit the widest possible “street” between the classes. In other words, the goal is to have the largest possible margin between the decision boundary that separates the two classes and the training instances. When performing soft margin classification, the SVM searches for a compromise between perfectly separating the two classes and having the widest possible street (ie. a few instances may end up on the street). Another key idea is to use kernels when training on non-linear datasets.
2. After training an SVM, a *support vector* is any instance located on the “street” (see the previous answer), including its border. The decision boundary is entirely determined by the support vectors: any instance that is *not* a support vector (ie. off the street) has no influence whatsoever, you could remove them, add more instances or move them around, as long as they stay off the street they won’t affect the decision boundary. Computing the predictions only involves the support vectors, not the whole training set.
3. SVMs try to fit the largest possible “street” between the classes (see the first answer), so if the training set is not scaled, the SVM will tend to neglect small features (see [Figure 5-2](#)).
4. An SVM classifier can output the distance between the test instance and the decision boundary, and you can use this as a confidence score. However, this score cannot be directly converted into an estimation of the class probability. If you set `probability=True` when creating an SVM in Scikit-Learn, then after training it will calibrate the probabilities using Logistic Regression on the SVM’s scores (trained by an additional 5-fold cross-validation on the training data). This will add the `predict_proba()` and `predict_log_proba()` methods to the SVM.
5. This question only applies to linear SVMs since kernelized can only use the dual form. The computational complexity of the primal form of the SVM problem is proportional to the number of training instances m , while the computational

complexity of the dual form is proportional to a number between m^2 and m^3 . So if there are millions of instances, you should definitely use the primal form, because the dual form will be much too slow.

6. If an SVM classifier trained with an RBF kernel underfits the training set, there might be too much regularization. To decrease it, you need to increase `gamma` or `C` (or both).
7. Let's call the QP parameters for the hard-margin problem \mathbf{H}' , \mathbf{f}' , \mathbf{A}' and \mathbf{b}' (see “[Quadratic Programming](#)” on page 183). The QP parameters for the soft-margin problem have m additional parameters ($n_p = n + 1 + m$) and m additional constraints ($n_c = 2m$). They can be defined like so:
 - \mathbf{H} is equal to \mathbf{H}' , plus m columns of 0s on the right and m rows of 0s at the bottom:
$$\mathbf{H} = \begin{pmatrix} \mathbf{H}' & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{0} & \\ \vdots & & \ddots \end{pmatrix}$$
 - \mathbf{f} is equal to \mathbf{f}' with m additional elements, all equal to the value of the hyper-parameter C .
 - \mathbf{b} is equal to \mathbf{b}' with m additional elements, all equal to 0.
 - \mathbf{A} is equal to \mathbf{A}' , with an extra $m \times m$ identity matrix \mathbf{I}_m appended to the right, $-\mathbf{I}_m$ just below it, and the rest filled with zeros:
$$\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}$$
8. See the Jupyter notebooks available at <https://github.com/ageron/handson-ml>.
9. See the Jupyter notebooks.
10. See the Jupyter notebooks.

Chapter 6

1. The depth of a well balanced binary tree containing m leaves is equal to $\log_2(m)$ ³, rounded up. A binary Decision Tree (one that only makes binary decisions, as is the case of all trees in Scikit-Learn), will end up more or less well balanced at the end of training, with one leaf per training instance if it is trained without restrictions. Thus if the training set contains one million instances, the Decision

³ \log_2 is the binary log, $\log_2(m) = \log(m)/\log(2)$.

Tree will have a depth of $\log_2(10^6) \approx 20$ (actually a bit more since the tree will generally not be perfectly well balanced).

2. A node's Gini impurity is generally lower than its parent's. This is ensured by the CART training algorithm's cost function, which splits each node in a way that minimizes the weighted sum of its children's Gini impurities. However, if one child is smaller than the other, it is possible for it to have a higher Gini impurity than its parent, as long as this increase is more than compensated by a decrease of the other child's impurity. For example, consider a node containing 4 instances of class A and 1 of class B. Its Gini impurity is $1 - \frac{1^2}{5} - \frac{4^2}{5} = 0.32$. Now suppose the dataset is one-dimensional and the instances are lined up in the following order: A, B, A, A, A. You can verify that the algorithm will split this node after the 2nd instance, producing one child node with instances A, B and the other child node with instances A, A, A. The first child node's Gini impurity is $1 - \frac{1^2}{2} - \frac{1^2}{2} = 0.5$, which is higher than its parent. This is compensated by the fact that the other node is pure, so the overall weighted Gini impurity is $\frac{2}{5} \times 0.5 + \frac{3}{5} \times 0 = 0.2$, which is lower than the parent's Gini impurity.
3. If a Decision Tree is overfitting the training set, it may be a good idea to decrease `max_depth`, since this will constrain the model, regularizing it.
4. Decision Trees don't care whether or not the training data is scaled or centered, that's one of the nice things about them. So if a Decision Tree underfits the training set, scaling the input features will just be a waste of time.
5. The computational complexity of training a Decision Tree is $O(n \times m \times \log(m))$. So if you multiply the training set size by 10, the training time will be multiplied by $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. If $m = 10^6$, then $K \approx 11.7$, so you can expect the training time to be roughly 11.7 hours.
6. Presorting the training set only speeds up training if the dataset is smaller than a few thousand instances. If it contains 100,000 instances, setting `presort=True` will considerably slow down training.
7. See the Jupyter notebooks available at <https://github.com/ageron/handson-ml>.
8. See the Jupyter notebooks.

Chapter 7

1. If you have trained 5 different models and they all achieve 95% precision, you can try combining them into a voting ensemble, this will often give you even bet-

ter results. It works better if the models are very different (eg. an SVM classifier, a Decision Tree classifier, a Logistic Regression classifier, and so on). It is even better if they are trained on different training instances (that's the whole point of Bagging and Pasting ensembles), but if not it will still work as long as the models are very different.

2. A hard Voting Classifier just counts the votes of each classifier in the ensemble and picks the class that gets the most votes. A soft Voting Classifier computes the average estimated class probability for each class and picks the class with the highest probability. This gives high confidence votes more weight and often performs better, but it works only if every classifier is able to estimate class probabilities (eg. for the SVM classifiers in Scikit-Learn you must set `probability=True`).
3. It is quite possible to speed up training of a Bagging ensemble by distributing it across multiple servers, since each predictor in the ensemble is independent of the others. The same goes for Pasting ensembles and Random Forests, for the same reason. However, each predictor in a Pasting ensemble is built based on the previous predictor, training is necessarily sequential: you will not gain anything by distributing training across multiple servers. Regarding stacking ensembles, all the predictors in a given layer are independent of each other, so they can be trained in parallel on multiple servers. However, the predictors in one layer can only be trained after the predictors in the previous layer have all been trained.
4. With Out-of-Bag evaluation, each predictor in a Bagging ensemble is evaluated using instances that it was not trained on (they were held out). This makes it possible to have a fairly unbiased evaluation of the ensemble without the need for an additional validation set. Thus, you have more instances available for training, and your ensemble can perform slightly better.
5. When growing a tree in a Random Forest, only a random subset of the features is considered for splitting at each node. This is true as well for Extra-Trees, but they go one step further: rather than searching for the best possible thresholds, like regular Decision Trees do, they use random thresholds for each feature. This extra randomness acts like a form of regularization: if a Random Forest overfits the training data, Extra-Trees might perform better. Moreover, since Extra-Trees don't search for the best possible thresholds, they are much faster to train than Random Forests. However, they are neither faster nor slower than Random Forests when making predictions.
6. If your AdaBoost ensemble underfits the training data, you can try increasing the number of estimators or reducing the regularization hyperparameters of the base estimator. You may also try slightly increasing the learning rate.
7. If your Gradient Boosting ensemble overfits the training set you should try decreasing the learning rate. You could also use Early Stopping to find the right number of predictors (you probably have too many).

8. See the Jupyter notebooks available at <https://github.com/ageron/handson-ml>.
9. See the Jupyter notebooks.



Solutions for other chapters are coming soon.

APPENDIX B

Machine Learning project checklist

This checklist can guide you through your Machine Learning projects. There are 8 main steps:

1. Frame the problem and look at the big picture
2. Get the data.
3. Explore it, gain insights.
4. Prepare it to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and short-list the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

Frame the problem and look at the big picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions / workarounds (if any)?
4. How should you frame this problem? (supervised/unsupervised, online/offline, etc.)
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?

7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

Get the data

Note: Automate as much as possible so you can easily get fresh data.

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (eg. anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside and never look at it (no data snooping!).

Explore the data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
 - name
 - type (categorical, int/float, bounded/unbounded, text, structured, etc.)

- % of missing values
 - noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
 - possibly useful for the task?
 - type of distribution (Gaussian, uniform, logarithmic, etc.)
4. For supervised learning tasks, identify the target attribute(s).
 5. Visualize the data.
 6. Study the correlations between attributes.
 7. Study how you would solve the problem manually.
 8. Identify the promising transformations you may want to apply.
 9. Identify extra data that would be useful (go back to “[Get the data](#)” on page 408).
 10. Document what you have learned.

Prepare the data

Notes:

- Work on copies of the data (keep the original dataset intact).
 - Write functions for all data transformations you apply, for 4 reasons:
 - so you can easily prepare the data the next time you get a fresh dataset,
 - so you can apply these transformations in future projects,
 - to clean & prepare the test set,
 - to clean & prepare new data instances once your solution is live,
 - and to make it easy to treat your preparation choices as hyperparameters.
1. Data cleaning:
 - Fix or remove outliers (optional).
 - Fill in missing values (eg. with zero, mean, median...) or drop their rows (or columns).
 2. Feature selection (optional):
 - Drop the attributes that provide no useful information for the task.
 3. Feature engineering, where appropriate:
 - Discretize continuous features.

- Decompose features (eg. categorical, date/time, etc.).
 - Add promising transformations of features (eg. $\log(x)$, \sqrt{x} , x^2 , etc.).
 - Aggregate features into promising new features.
4. Feature scaling: standardize or normalize features.

Short-list promising models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or random forests).
 - Once again, try to automate these steps as much as possible.
1. Train many quick & dirty models from different categories (eg. linear, naive Bayes, SVM, random forest, neural net, etc.) using standard parameters.
 2. Measure and compare their performance.
 - For each model, use N-fold cross-validation and compute the mean and standard deviation of the performance measure on the N folds.
 3. Analyze the most significant variables for each algorithm.
 4. Analyze the types of errors the models make.
 - What data would a human have used to avoid these errors?
 5. Quick round of feature selection and engineering.
 6. One or two more quick iterations of the 5 previous steps.
 7. Short-list the top 3-5 most promising models, preferring models that make different types of errors.

Fine-tune the system

Notes:

- You will want to use as much data as possible for this step, especially as you move towards the end of fine-tuning.
- As always automate what you can.

1. Fine-tune the hyperparameters using cross-validation.
 - Treat your data transformation choices as hyperparameters, especially when you are not sure about them (eg. should I replace missing values with zero or with the median value? Or just drop the rows?).
 - Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (eg. using Gaussian process priors¹).
2. Try ensemble methods. Combining your best models will often perform better than running them individually.
3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.



Don't tweak your model after measuring the generalization error:
you would just start overfitting the test set.

Present your solution

1. Document what you have done.
2. Create a nice presentation.
 - Making sure you highlight the big picture first.
3. Explain why your solution achieves the business objective.
4. Don't forget to present interesting points you noticed along the way.
 - What worked and what did not.
 - List your assumptions and your system's limitations.
5. Ensure your key findings are communicated through beautiful visualizations or easy to remember statements (eg. "the median income is the number one predictor of housing prices").

¹ As described in Snoek, Larochelle, Adams (2012): <https://goo.gl/PEFfGr>

Launch!

1. Get your solution ready for production (eg. plug to production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
 - Beware of slow degradation too: models tend to “rot” as data evolves.
 - Measuring performance may require a human pipeline (eg. via a crowdsourcing service).
 - Also monitor your inputs' quality (eg. a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

APPENDIX C

Autodiff

This appendix explains how TensorFlow’s autodiff feature works, and how it compares to other solutions.

Suppose you define a function $f(x, y) = x^2y + y + 2$, and you need its partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are: manual differentiation, symbolic differentiation, numerical differentiation, forward-mode autodiff and finally reverse-mode autodiff. TensorFlow implements this last option. Let’s go through each of these options.

Manual differentiation

The first approach is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the partial derivatives manually. For the function $f(x, y)$ defined above, it is not too hard, you just need to use 5 rules:

- the derivative of a constant is 0,
- the derivative of λx is λ (where λ is a constant),
- the derivative of x^λ is $\lambda x^{\lambda - 1}$, so the derivative of x^2 is $2x$,
- the derivative of a sum of functions is the sum of these functions’ derivatives,
- the derivative of λ times a function is λ times its derivative.

From these rules, you can derive the following equations:

Equation C-1. Partial derivatives of $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. The good news is that deriving the mathematical equations for the partial derivatives like we just did can be automated: this is called *symbolic differentiation*.

Symbolic differentiation

Figure C-1 shows how symbolic differentiation works on an even simpler function $g(x, y) = 5 + xy$. The graph for that function is represented on the left. After symbolic differentiation, we get the graph on the right, which represents the partial derivative $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regards to y).

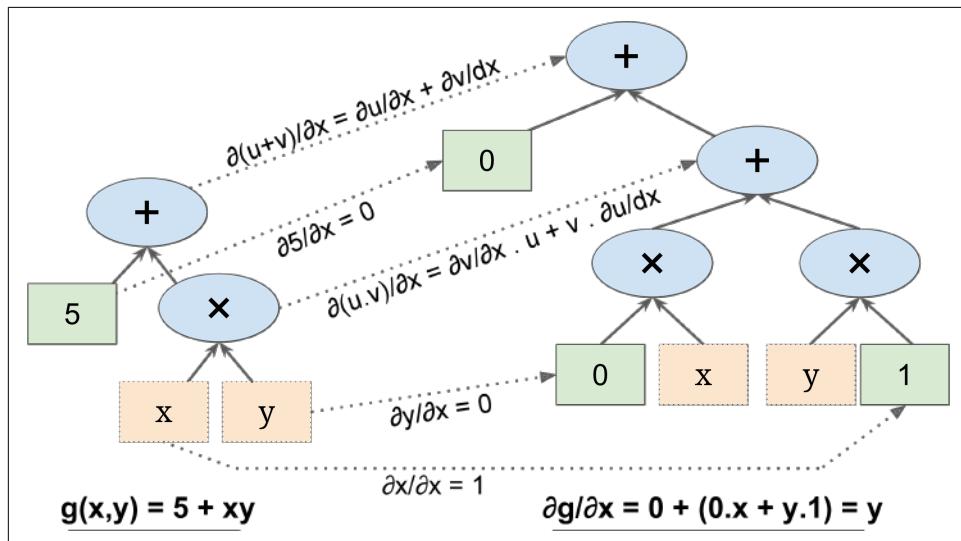


Figure C-1. Symbolic differentiation

The algorithm starts by getting the partial derivative of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The

variable x returns the constant 1 since $\frac{\partial x}{\partial x} = 1$, and the variable y returns the constant 0 since $\frac{\partial y}{\partial x} = 0$ (if we were looking for the partial derivative with regards to y , it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g . Calculus tells us that the derivative of the product of two functions u and v is $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + \frac{\partial u}{\partial x} \times v$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g . As we mentioned above, the derivative of a sum of functions is the sum of these functions' derivatives. So we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$.

However, it can be simplified (a lot). A few trivial pruning steps can be applied to this graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\frac{\partial g}{\partial x} = y$.

In this case, simplification is fairly easy, but for a more complex function, symbolic differentiation can produce a huge graph that may be tough to simplify and lead to sub-optimal performance. Most importantly, symbolic differentiation cannot deal with functions defined with arbitrary code, for example the following function discussed in [Chapter 9](#):

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(a - i)
    return z
```

Numerical differentiation

The simplest solution is to compute an approximation of the derivatives, numerically. Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point, or more precisely:

Equation C-2. Derivative of a function $h(x)$ at point x_0

$$\begin{aligned} h'(x) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\epsilon \rightarrow 0} \frac{h(x_0 + \epsilon) - h(x_0)}{\epsilon} \end{aligned}$$

So if we want to calculate the partial derivative of $f(x, y)$ with regards to x , at $x_0 = 3$ and $y_0 = 4$, we can simply compute $f(3 + \epsilon, 4) - f(3, 4)$ and divide the result by ϵ , using a very small value for ϵ . That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complex functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call `f()` at least 3 times (we called it 4 times in the code above, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When dealing with large neural networks, this makes numerical differentiation way too inefficient.

However, numerical differentiation is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

Forward-mode autodiff

Forward-mode autodiff is neither numerical differentiation nor symbolic differentiation, but in some ways it is their love child. It relies on *dual numbers* which are (weird but fascinating) numbers of the form $a + b\epsilon$ where a and b are real numbers and ϵ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$). You can think of the dual number $42 + 24\epsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this is not rigorous, it is just to give you some intuition of what dual numbers are). A dual number is represented in memory as a pair of floats. For example $42 + 24\epsilon$ is represented by the pair $(42.0, 24.0)$.

Dual numbers can be added, multiplied, and so on. For example:

Equation C-3. A few operations with dual numbers

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Most importantly, it can be shown that $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, so computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. **Figure C-2** shows how forward-mode autodiff computes the partial derivative of $f(x, y)$ with regards to x at $x = 3$ and $y = 4$. All we need to do is compute $f(3 + \epsilon, 4)$: this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\frac{\partial f}{\partial x}(3, 4)$.

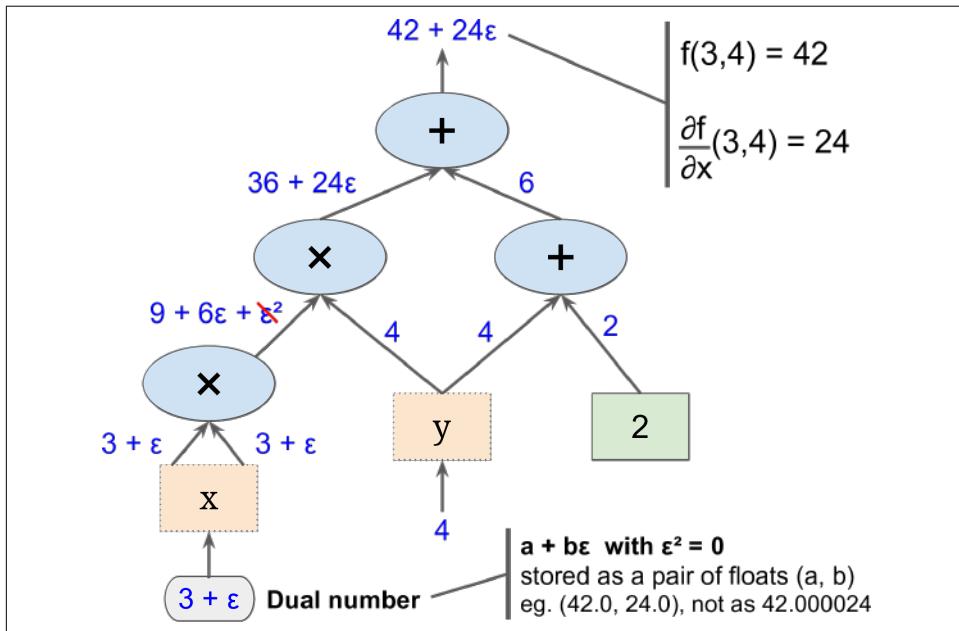


Figure C-2. Autodiff – forward mode

To compute $\frac{\partial f}{\partial y}(3, 4)$ we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \epsilon$.

So forward-mode autodiff is much more accurate than numerical differentiation, but it suffers from the same major flaw: if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial derivatives. This is where

reverse-mode autodiff shines: it can compute all of them in just two passes through the graph.

Reverse-mode autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (ie. from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (ie. from the output to the inputs) to compute all the partial derivatives. Figure C-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (eg. $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7 : $f(3, 4) = n_7 = 42$.

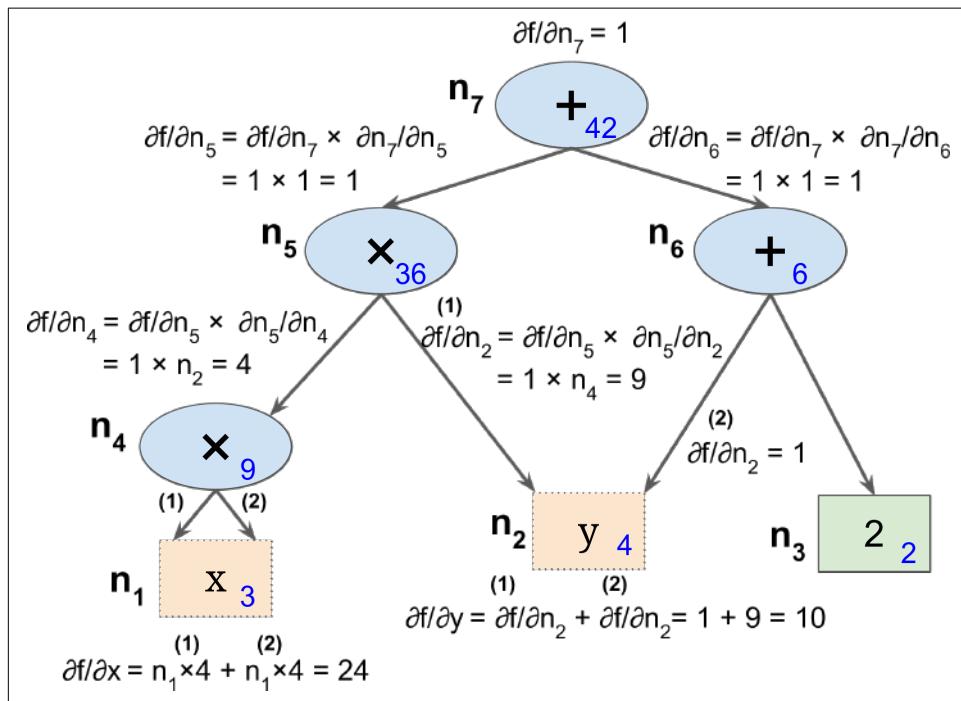


Figure C-3. Autodiff – reverse mode

The idea is to gradually go down the graph, computing the partial derivative of $f(x, y)$ with regards to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*:

Equation C-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$ so trivially $\frac{\partial f}{\partial n_7} = 1$.

Let's continue down the graph to n_5 : how much does f vary when n_5 varies? The answer is $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. We already know that $\frac{\partial f}{\partial n_7} = 1$, so all we need is $\frac{\partial n_7}{\partial n_5}$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\frac{\partial n_7}{\partial n_5} = 1$, so $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Now we can proceed to node n_4 : how much does f vary when n_4 varies? The answer is $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Since $n_5 = n_4 \times n_2$, we find that $\frac{\partial n_5}{\partial n_4} = n_2$, so $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x, y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\frac{\partial f}{\partial x} = 24$ and $\frac{\partial f}{\partial y} = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it only requires one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regards to all the inputs. Most importantly, it can deal with functions defined by arbitrary code. It can also handle non-differentiable functions, as long as you ask it to compute the partial derivatives at points that are differentiable.



If you implement a new type of operation in TensorFlow and you want to make it compatible with autodiff, then you need to provide a function that builds a subgraph to compute its partial derivatives with regards to its inputs. For example, suppose you implement a function that computes the square of its input $f(x) = x^2$, then you would need to provide the corresponding derivative function $f'(x) = 2x$.

Index