



**Università
degli Studi
di Ferrara**

Dipartimento di Fisica e Scienze della Terra
Master Degree in Physics

ARTIFICIAL INTELLIGENCE

Francesco Tralli¹

A.Y. 2020/2021

(Last revised August 15, 2021)

https://drive.google.com/open?id=1S9c3__0lItatnKvXhQwKFGItglqwd8MU

¹francesco.Tralli@edu.unife.it

Contents

	Page
Preface	vii
1 Problem solving by uninformed and informed search	1
1.1 Intelligent agents	2
1.1.1 Rationality	2
1.1.2 Table-driven agents	3
1.1.3 Simple reflex agents	4
1.1.4 Model-based reflex agents	6
1.1.5 Goal-based agents	6
1.1.6 Utility-based agents	7
1.1.7 Learning agents	7
1.1.8 The nature of the environment	8
1.2 Example problems	9
1.2.1 Toy problems	9
1.2.1.1 Missionaries and cannibals	9
1.2.1.2 8-puzzle	10
1.2.1.3 8-queens problem	12
1.2.2 Real-world problems	13
1.3 Setting a problem and searching solutions	13
1.3.1 Performance	14
1.4 Uninformed search	15
1.4.1 Breadth-first search	15
1.4.2 Uniform-cost search	17
1.4.3 Depth-first search	17
1.4.4 Depth-limited search	19
1.4.5 Iterative deepening depth-first search	19
1.4.6 Bidirectional search	21
1.5 Heuristic functions and informed search	21
1.5.1 Greedy best-first search	22
1.5.2 A* search	22
1.5.3 Inventing heuristic functions	25
1.5.4 Learning through experience	28
1.6 Local search	28
1.6.1 Genetic algorithms	31
1.6.1.1 Traveling salesman problem	33
2 Monte Carlo techniques	38
2.1 A short review of statistics	38

Contents

2.1.1	Sum of many variables	39
2.1.2	Central limit theorem	42
2.2	Pseudo-random number generation	44
2.2.1	Linear congruential generator	44
2.2.2	Combining generators	46
2.2.3	Transformation/inversion method	47
2.2.4	Rejection method	49
2.2.5	Special technique for Gaussian distributions	50
2.3	Metropolis algorithm	52
2.3.1	Simulated annealing	55
2.3.2	Traveling salesman problem	57
2.3.3	Monte Carlo integration	60
2.3.4	Correlations	62
3	Neural networks	65
3.1	Basic introduction to neuroscience	65
3.1.1	Neurons and brain	65
3.2	Artificial neural networks	69
3.2.1	Deterministic networks: associative memory	69
3.2.2	Learning by Hebb's rule	72
3.2.3	Diederich and Opper method	75
3.2.4	Spin glasses	78
3.2.4.1	Parallel versus sequential dynamics	79
3.2.5	Neural "motion pictures"	81
3.2.6	Stochastic neurons	83
3.2.6.1	Single pattern	86
3.2.6.2	Several patterns	87
3.2.7	Special learning rules	89
3.3	Simple perceptron	90
3.3.1	The exclusive-OR (XOR) gate	95
3.4	Multilayer perceptron	98
3.4.1	Solution of the XOR problem	98
3.4.2	Error back-propagation	100
3.4.2.1	Arbitrary number of hidden layers	101
3.4.3	Boolean functions	104
3.4.4	Continuous functions	107
3.4.5	Generalization and fitting	110
3.5	Boltzmann machines	113
3.5.1	Information theory	115
3.5.2	Mutual information	116
3.5.3	The "Boltzmann" learning rule	119
3.5.3.1	Applications	123
3.6	Restrictive Boltzmann machines	124
4	Machine learning techniques	126
4.1	Introduction to machine learning	128
4.1.1	Supervised/unsupervised learning	128
4.1.1.1	Supervised learning	128
4.1.1.2	Unsupervised learning	129
4.1.1.3	Semisupervised learning	130

4.1.1.4	Reinforcement learning	130
4.1.2	Batch and online learning	131
4.1.2.1	Batch learning	131
4.1.2.2	Online learning	132
4.1.3	Instance-based and model-based learning	132
4.1.3.1	Instance-based learning	133
4.1.3.2	Model-based learning	133
4.2	Testing and validating	133
4.3	Quality of data and preprocessing	135
4.4	California housing prices	136
4.4.1	Frame the problem	136
4.4.2	Select a performance measure	136
4.4.3	Get the data	137
4.4.4	Create a test set	140
4.4.5	Data visualization	142
4.4.6	Looking for correlations	144
4.4.7	Experimenting with attribute combinations	146
4.4.8	Data cleaning	147
4.4.9	Handling text and categorical attributes	148
4.4.10	Feature scaling	149
4.4.11	Transformation pipelines	150
4.4.12	Select and train a model	150
4.4.13	Better evaluation using cross-validation	152
4.4.14	Fine-tune your model	154
4.4.15	Analyze the best models and their errors	156
4.4.16	Evaluate your system on the test set	156
4.5	Training models	157
4.5.1	Linear regression	157
4.5.2	Gradient descent	161
4.5.2.1	Stochastic gradient descent	163
4.5.2.2	Mini-batch gradient descent	164
4.5.3	Polynomial regression	165
4.5.4	Learning curves	166
4.5.4.1	On the nature of errors	168
4.5.5	Regularized linear models	169
4.5.5.1	Ridge regression	169
4.5.5.2	Lasso regression	170
4.5.5.3	Early stopping	170
4.5.6	Logistic regression	171
4.5.6.1	Estimating probabilities	172
4.5.6.2	Training and cost function	172
4.5.6.3	Decision boundaries	173
4.5.6.4	Softmax regression	175
4.6	Support vector machines	176
4.6.1	Linear SVM classification	176
4.6.1.1	Soft margin classification	177
4.6.2	Nonlinear SVM classification	178
4.6.2.1	Polynomial kernel	179
4.6.3	Mathematical formulation	180

Contents

4.6.3.1	Decision function and predictions	180
4.6.3.2	Training objective	181
4.6.3.3	The dual problem	182
4.6.4	Kernelized SVMs	183
4.7	Dimensionality reduction	184
4.7.1	The curse of dimensionality	184
4.7.2	Main approaches for dimensionality reduction	185
4.7.2.1	Projection	185
4.7.2.2	Manifold learning	186
4.7.3	PCA	187
4.7.3.1	Determination of the principal components	188
4.7.3.2	Choosing the right number of dimensions	190
4.7.3.3	PCA for compression	190
4.7.4	Kernel PCA	191
4.7.5	LLE	192
4.8	Unsupervised learning techniques	193
4.8.1	Clustering	193
4.8.1.1	K-means	194
4.8.2	Gaussian mixtures	201
4.8.2.1	Anomaly detection	204
4.8.2.2	Selecting the number of clusters	204
4.9	Artificial neural networks with Keras	206
4.9.1	Summary of artificial neural networks	207
4.9.2	Building an image classifier using Keras	214
4.9.2.1	Using Keras to load the dataset	214
4.9.2.2	Creating the model using the sequential API	215
4.9.2.3	Compiling the model	217
4.9.2.4	Training and evaluating the model	218
4.10	Training deep neural networks (credit: Francesco Di Clemente)	219
4.10.1	Learning rate scheduling	220
4.10.2	The vanishing/exploding gradients problems	220
4.10.3	Faster optimizers	223
4.10.3.1	Momentum optimization	224
4.10.3.2	Nesterov accelerated gradient	224
4.10.3.3	AdaGrad	225
4.10.3.4	RMSProp	226
4.10.3.5	Adam	226
4.10.4	Avoiding overfitting through regularization	227
4.10.4.1	Dropout	227
4.10.4.2	Monte Carlo dropout	228
4.10.5	Example: Higgs dataset	229
4.11	Convolutional neural networks	229
4.11.1	Convolutional layers	230
4.11.1.1	Filters	231
4.11.1.2	Stacking multiple feature maps	232
4.11.1.3	Memory requirements	234
4.11.2	Pooling layers	235
4.11.3	Data augmentation	236

5 Adversarial search	238
5.1 Go	239
5.1.1 Basic rules	239
5.1.1.1 Liberties and capture	240
5.1.1.2 Ko rule	240
5.1.1.3 Suicide	241
5.1.1.4 Life and death	242
5.1.1.5 Scoring rules	243
5.1.2 Tactics	243
5.1.3 Ranks and ratings	243
5.2 Game programming	244
5.2.1 History of chess programming	244
5.2.2 Complexity of chess and Go	245
5.2.3 Modeling the three phases of Go	248
5.2.3.1 Programming a Go game	250
5.2.3.2 Coloring the graph	251
5.3 Adversarial search	252
5.3.1 Optimal decision: minimax	254
5.3.1.1 Multiplayer games	255
5.3.2 Alpha–beta pruning	255
5.3.3 Evaluation function	256
5.3.3.1 Horizon effect	258
5.3.4 Monte Carlo tree search	259
5.4 Reinforcement learning	262
5.4.1 Policy search	262
5.4.2 Introduction to OpenAI gym	263
5.4.3 The credit assignment problem	264
5.4.4 Policy gradients	266
5.4.5 Markov decision processes	267
5.5 Alpha Go, AlphaGo Zero & Alpha Zero	269
5.5.1 The algorithm	270
5.5.1.1 The pillars	270
5.5.1.2 The training	272
5.5.1.3 Remarks	274
5.5.2 Performances	276
Appendix. Programs and computer codes	280
A ASSO: associative memory	281
A.1 Program description	283
A.2 Snapshots	285
B ASSCOUNT: associative memory for time sequences	289
B.1 Program description	290
B.2 Numerical experiments	291
B.3 Snapshots	291
C PERBOOL: learning Boolean functions with back-propagation	294
C.1 Program description	295

Contents

C.2 Numerical experiments	297
C.3 Snapshots	299
D PERFUNC: learning continuous functions with back-propagation	303
D.1 Program description	303
D.2 Numerical experiments	305
D.3 Snapshots	306
References	309
Books	309
Articles	309
Incollections	310
Technical reports	310
Lectures notes	310
Online & websites	310

Preface

These notes are based on the lectures of Artificial Intelligence (Master Degree in Physics) held by Professor Alessandro Drago at Università degli Studi di Ferrara during the academic year 2020/2021. In addition to the knowledge provided by the professor himself during the course, about a dozen of various texts (books, articles, websites, etc.) have been consulted for the preparation of these notes, in order to make the final result more accurate and complete. The entire list of all references can be easily found at the very end of this document. Nonetheless, the main reference textbooks on which this text is based, as recommended by the professor himself, are:

- “Artificial Intelligence: A Modern Approach” by S. Russell and P. Norvig [1, 2] for the introductory part (Chapter 1) and for the section concerning adversarial search in Chapter 5;
- “Numerical Recipes: The Art of Scientific Computing” by W. H. Press et al. [3] for random number generation and Monte Carlo techniques (Chapter 2);
- “Neural Networks: An Introduction” by B. Müller, J. Reinhardt, M. T. Strickland [6] for the part of neural networks (Chapter 3);
- “Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow” by Aurélien Géron [8, 9] for the part of machine learning (Chapter 4) and for the section “Reinforcement learning” in the last chapter (Chapter 5).

Also, the appendix of this book contains a collection of instructive programs taken from [6] and written in C.

Furthermore, in order to make it easier to understand the covered topics, within the text are embedded some examples of codes based on calculation programs such as Python or Mathematica. These are also attached to the pdf file and indicated in the text with a small push pin icon . By default setting Adobe Acrobat Reader does not allow you to open or save attachments with particular extensions that can represent a potential security risk. These are, for instance, executable files (.exe), archives (.rar), but also (and unfortunately) Python files (.py) because they are the most suspected type of files to carry malicious contents, open other dangerous file or launch applications. In order to change these settings, follow the procedure indicated at the following link: <https://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/attachments.html?highlight=blacklist>. Nevertheless, unfortunately, these modified settings are restored to the default ones whenever Adobe Acrobat Reader is updated (and blocking automatic updates is quite tricky).

There is at disposal also a great collection of very complicated (and complete) codes from Russell and Norvig [1] that covers all the textbook. If you’re interested in, you can find them on GitHub, but they have a lot of dependencies, also internal dependencies, so you have to follow the instructions written at the link <https://github.com/aimacode/aima-python>.

Preface

Instead, the examples taken from Géron [8, 9] are available for consultation at the following link, written also inside the book itself: <https://github.com/ageron/handson-ml2>.

Having said that, as exhaustive as they may seem, these notes should be considered as a study support for a better follow-up of the course, and not as a substitute for face-to-face lessons.

These notes are in constant update and it is therefore possible to find the most recent version at the link shown on the front page; if for some reasons the Google Drive link stops working, just contact me via mail or in person. Another remark is that some old versions of Adobe Acrobat Reader report an error (error code 131) when consulting the file. Although I have not yet fully understood the origin, it is possible to solve it by lowering the pdf version (e.g. from 1.5 to 1.4). Anyway, contact me and I will provide you a working version.

Readers are also strongly encouraged to report *any* kind of error (misprints, misspellings, mistakes) to the author. Of course, any errors that remain are my sole responsibility.

Francesco Tralli



CHAPTER 1

Problem solving by uninformed and informed search

Contents

1.1	Intelligent agents	2
1.1.1	Rationality	2
1.1.2	Table-driven agents	3
1.1.3	Simple reflex agents	4
1.1.4	Model-based reflex agents	6
1.1.5	Goal-based agents	6
1.1.6	Utility-based agents	7
1.1.7	Learning agents	7
1.1.8	The nature of the environment	8
1.2	Example problems	9
1.2.1	Toy problems	9
1.2.1.1	Missionaries and cannibals	9
1.2.1.2	8-puzzle	10
1.2.1.3	8-queens problem	12
1.2.2	Real-world problems	13
1.3	Setting a problem and searching solutions	13
1.3.1	Performance	14
1.4	Uninformed search	15
1.4.1	Breadth-first search	15
1.4.2	Uniform-cost search	17
1.4.3	Depth-first search	17
1.4.4	Depth-limited search	19
1.4.5	Iterative deepening depth-first search	19
1.4.6	Bidirectional search	21
1.5	Heuristic functions and informed search	21
1.5.1	Greedy best-first search	22
1.5.2	A* search	22
1.5.3	Inventing heuristic functions	25
1.5.4	Learning through experience	28
1.6	Local search	28
1.6.1	Genetic algorithms	31
1.6.1.1	Traveling salesman problem	33

1.1 Intelligent agents

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in Figure 1.1. For instance, a human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators. A robotic agent instead might have cameras and infrared range finders for sensors and various motors for actuators.

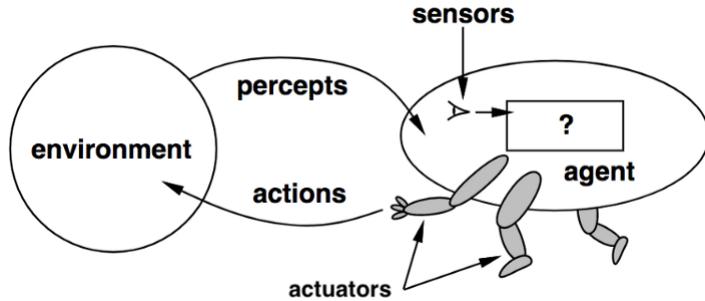


Figure 1.1: Agents interact with environments through sensors and actuators.

We use the term **percept** to refer to the agent's perceptual inputs at any given instant. An agent's **percept sequence** is the complete history of everything the agent has ever perceived. In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived. *By specifying the agent's choice of action for every possible percept sequence, we have said more or less everything there is to say about the agent.* Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

To illustrate these ideas, we use a very simple example, the vacuum cleaner world shown in Figure 1.2. This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares *A* and *B*. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt or do nothing. For example, one very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square.

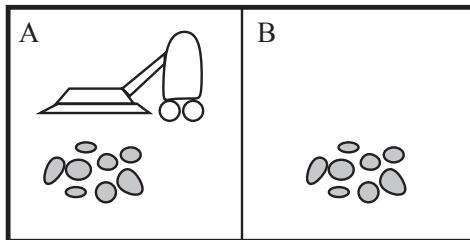


Figure 1.2: A vacuum-cleaner world with just two locations.

1.1.1 Rationality

Looking at Table 1.1 we see that various vacuum-world agents can be defined simply by filling in the right-hand column in various ways. The obvious question, then, is this: What

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
:	:

Table 1.1: Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 1.2.

is the right way to fill out the table? In other words, what makes an agent good or bad? A **rational agent** is an agent that does the right thing (conceptually speaking, every entry in the table for the agent function is filled out correctly). Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing? We answer this age-old question in an age-old way: by considering the *consequences* of the agent’s behavior. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is “desirable”, then the agent has performed well (and hence is said rational). This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

Notice that we said *environment* states, not *agent* states. If we define success in terms of agent’s opinion of its own performance, an agent could achieve perfect rationality simply by deluding itself that its performance was perfect.

Obviously, there is not one fixed performance measure for all tasks and agents; typically, a designer will devise one appropriate to the circumstances. *As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.*

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent’s prior knowledge of the environment.
- The actions that the agent can perform.
- The agent’s percept sequence to date.

This leads to a **definition of a rational agent**:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

1.1.2 Table-driven agents

So far we have talked about agents by describing behavior—the action that is performed after any given sequence of percepts. Now we must bite the bullet and talk about how the insides work. The job of AI is to design an *agent program* that implements the agent

function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators, which we call the *architecture*. Notice the difference between the agent program, which takes the current percept as input, and the agent function, which takes the entire percept history. The agent program takes just the current percept as input because nothing more is available from the environment; if the agent’s actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

In a simple **table-driven agent**, a lookup table is used to match every possible percept sequence to the corresponding action. The table—an example of which is given for the vacuum world in Table 1.1—represents explicitly the agent function that the agent program embodies. To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence. Table-driven agent programs are the most effective form of implementing the desired agent function, but it comes with a penalty of occupying humongous amounts of space. Moreover, no agent could ever learn all the right table entries from its experience.

It is instructive to consider why the table-driven approach to agent construction is doomed to failure. Let \mathcal{P} be the set of possible percepts and let T be the lifetime of the agent, i.e. the total number of percepts it will receive. The lookup table will then contain $\sum_{t=1}^T |\mathcal{P}|^t$ entries. Even the lookup table for chess—a tiny, well-behaved fragment of the real world—would have at least 10^{150} entries! Forget about the lookup table for the taxi driving agent.

Despite all this, table-driven agents do what we want: they implement the desired agent function. The key challenge for artificial intelligence is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table. We have many examples showing that this can be done successfully in other areas: for example, the huge tables of square roots (or logarithms) used by engineers and schoolchildren prior to the 1970s have now been replaced by a five-line program for Newton’s method running on electronic calculators.

Let us discuss four basic kinds of agent programs, used in almost all intelligent systems.

1.1.3 Simple reflex agents

The simplest kind of agent is the **simple reflex agent**. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Table 1.1 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in Figure 1.3.

```
function REFLEX-VACUUM-AGENT([location, status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Figure 1.3

Notice that the vacuum agent program is very small indeed compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of possibilities from 4^T to just 4. A further, small reduction comes from the fact that when the current square is dirty, the action does not depend on the location.

Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call “the car in front is braking”. Then, this triggers some established connection in the agent program to the action “initiate braking”.

Simple reflex agents have the admirable property of being simple, but they turn out to be of limited intelligence. These agents will work only if the correct decision can be made on the basis of only the current percept—that is, only if the environment is *fully observable*. Even a little bit of unobservability can cause serious trouble. For example, the braking rule given earlier assumes that the condition “the car in front is braking” can be determined from the current percept—a single frame of video. This works if the car in front has a centrally mounted brake light. Unfortunately, older models have different configurations of taillights, brake lights, and turn-signal lights, and it is not always possible to tell from a single image whether the car is braking. A simple reflex agent driving behind such a car would either brake continuously and unnecessarily, or, worse, never brake at all.

We can see a similar problem arising in the vacuum world. Suppose that a simple reflex vacuum agent is deprived of its location sensor and has only a dirt sensor. Such an agent has just two possible percepts: [*Dirty*] and [*Clean*]. It can *Suck* in response to [*Dirty*]; but what should it do in response to [*Clean*]? Moving *Left* fails (forever) if it happens to start in square *A*, and moving *Right* fails (forever) if it happens to start in square *B* (Figure 1.4). Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

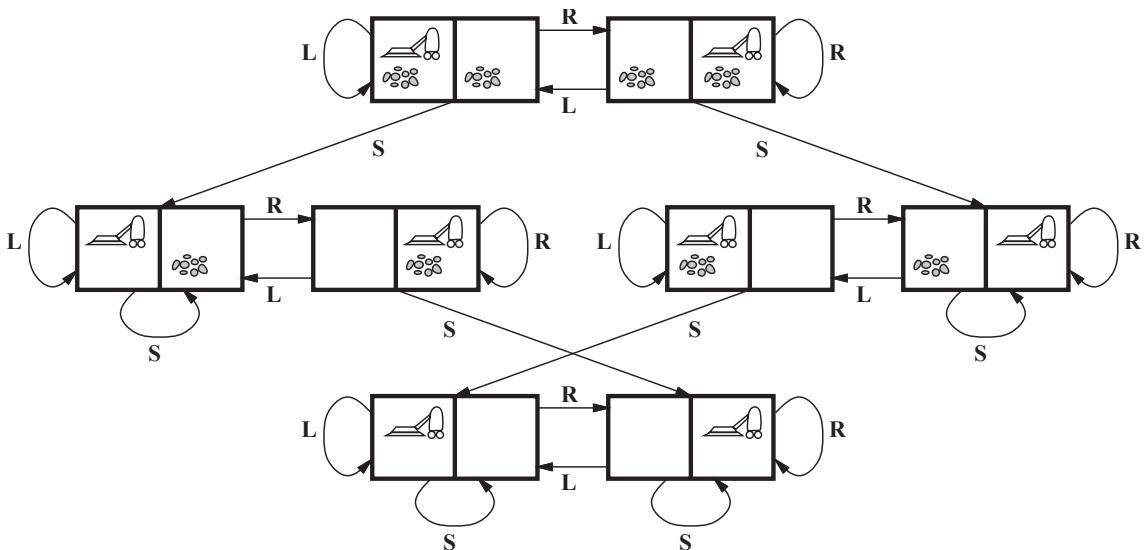


Figure 1.4: The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Suck. This world has 8 states; if we had n rooms, we would have $n2^n$ possible states.

Escape from infinite loops is possible if the agent can *randomize* its actions. For example, if the vacuum agent perceives [*Clean*], it might flip a coin to choose between *Left* and *Right*¹. It is easy to show that the agent will reach the other square in an average of two steps. Then, if that square is dirty, the agent will clean it and the task will be complete. Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.

¹Something similar will happen in neutral nets, as we will see. There we could fall in a false minimum; therefore we randomize the choice in order to exit and avoid getting stuck.

Note that randomized behavior of the right kind can be rational in some multiagent environments. But in single-agent environments, randomization is usually not rational. It is a useful trick that helps a simple reflex agent in some situations, but in most cases we can do much better with more sophisticated deterministic agents.

1.1.4 Model-based reflex agents

The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now. That is, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously. In other words, the taxi driver keeps track of the previous information about lights, and this allows it to understand (through the encoded knowledge) if the car in front is breaking and not parking.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program. First, we need some information about how the world evolves independently of the agent. Second, we need some information about how the agent's own actions presumably affect the world. This knowledge about “how the world works” is called a model of the world. An agent that uses such a model is called a **model-based agent**. In summary, a model-based reflex agent tries to complete the information from the environment by modeling how the environment will behave presumably in the near future, and it takes its decisions also according to that that. Clearly, the correspondent agent program depends on the sophistication of the environment.

1.1.5 Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the **goal-based agent** needs some sort of *goal information* that describes situations that are desirable—for example, being at the passenger's destination. The agent program can combine this with the model (the same information as was used in the model-based reflex agent) to choose actions that achieve the goal.

Notice that decision making of this kind is fundamentally different from the condition-action rules described earlier, since now it involves consideration of the future (what will happen if I do such-and-such?). In the reflex agent designs, this information is not explicitly represented, because the built-in rules map *directly* from percepts to actions, without questioning the goodness of its choice. The reflex agent brakes when it sees brake lights. A goal-based agent, in principle, *could reason* that if the car in front has its brake lights on, it will slow down.

Although the goal-based agent appears less efficient, it is *more flexible* because the knowledge that supports its decisions is represented explicitly and can be modified. If it starts to rain, the agent can update its knowledge of how effectively its brakes will operate; this will automatically cause all of the relevant behaviors to be altered to suit the new conditions. For the reflex agent, on the other hand, we would have to rewrite many condition-action rules. The goal-based agent's behavior can easily be changed to go to a different destination, simply by specifying that destination as the goal. The reflex agent's

rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

1.1.6 Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable or cheaper than others. Goals just provide a crude binary distinction between “happy” and “unhappy” states. A more general performance measure should allow an *external* comparison of different world states according to exactly how happy they would make the agent.

We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi’s destination. An agent’s **utility function** is essentially an internalization of the performance measure. If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

1.1.7 Learning agents

We have described agent programs with various methods for selecting actions. We have not, so far, explained how the agent programs come into being. The best method is to build learning machines and then to teach them. Learning allows the agent, now called a **learning agent**, to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. Learning agents are essentially made by four conceptual components.

- The *performance element*, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.
- The *learning element*, which is responsible for making improvements. The learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future (acts as an intermediary).
- The *critic* tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent’s success. For example, a chess program could receive a percept indicating that it has checkmated its opponent, but it needs a performance standard to know that this is a good thing; the percept itself does not say so.
- The last component of the learning agent is the *problem generator*. It is responsible for suggesting actions that will lead to new and informative experiences. The point is that if the performance element had its way, it would keep doing the actions that are best, given what it knows. But if the agent is willing to explore a little and do some perhaps suboptimal actions in the short run, it might discover much better actions for the long run. The problem generator’s job is to suggest these exploratory actions².

²Problems can also be self-generated by the agents. There exist a special branch of learning called temporal difference (TD) which learn by bootstrapping from the current estimate of the value function

1.1.8 The nature of the environment

In our discussion of the rationality of the simple vacuum-cleaner agent, we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these under the heading of the *task environment*. For the acronymically minded, we call this the PEAS (Performance, Environment, Actuators, Sensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible. In Table 1.2, we have sketched the basic PEAS elements for a number of additional agent types.

Agent	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, test, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Table 1.2

The range of task environments that might arise in AI is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation.

Accessible vs. Inaccessible If an agent's sensory apparatus gives it access to the complete state of the environment then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action (which is not the case of the taxi driver because it can't predict what other people may do). An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

Deterministic vs. non-deterministic If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is inaccessible, however, then it may *appear* to be non-deterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or non-deterministic *from the point of view of the agent*.

(uses the difference in utilities between successive states). An example is the program AlphaZero, developed by artificial intelligence research company Google DeepMind to master the games of chess, shogi and Go.

Episodic vs. non-episodic In an episodic environment, the agent’s experience is divided into “episodes”. Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

Static vs. dynamic If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent’s performance score does (e.g. for chess game with a clock), then we say the environment is **semi-dynamic**.

Discrete vs. continuous If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	Yes	No	Yes

Table 1.3: Example of environments and their characteristic.

1.2 Example problems

1.2.1 Toy problems

1.2.1.1 Missionaries and cannibals

The missionaries and cannibals problem³ is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold at most two people. The boat cannot cross the river by itself with no people on board. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place (if they were, the cannibals would eat the missionaries).

³This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint [17].

There are basically two ways to solve this problem. The trivial one consists in trying manually until we get the solution (deprecated). A smarter option instead would be to reason about the possible states and legal actions (Fig. 1.5). As we did in the previous sections for the vacuum-world, we start with the initial state and we build a tree diagram contemplating all possible transitions from one bank to the other. Obviously, any node that has more cannibals than missionaries on either bank is in an invalid state, and is therefore removed from further consideration. When we find a path that connects the initial states with the final one (all people in the opposite bank), then we've found one possible solution. Naturally, we expect the solutions to be infinite in number because every action can restore the environment to the previous configuration, generating a sort of loop that does not provide any help with the resolution. Nonetheless, it is possible to verify that there are 4 possible shortest solutions, each one requiring 11 boat trips.

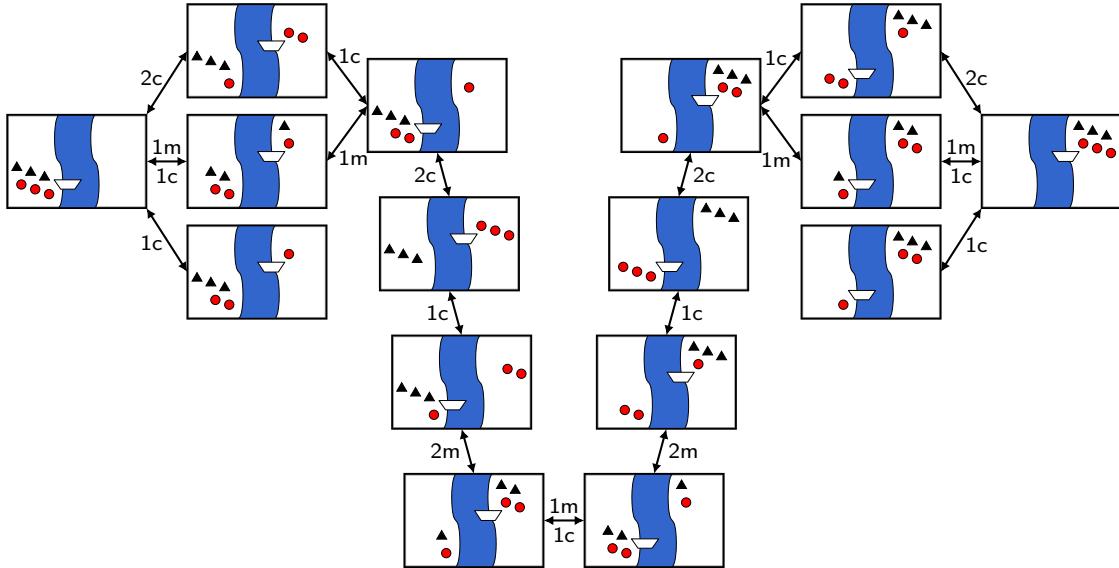


Figure 1.5: State space for the missionaries and cannibals problem. Only allowed states and transitions are shown. Black triangles represent missionaries and red circles represent cannibals. The 4 shortest solutions are easily identifiable.

The missionaries and cannibals problem is a well-known toy problem in artificial intelligence. A **toy problem** is a problem that is not of immediate scientific interest, yet is used as an expository device to explain a particular, more general, problem solving technique. Due to their quite simple formulations, toy problems are useful to test and demonstrate methodologies. They can be given a concise, exact description and hence are usable by different researchers to compare the performance of algorithms.

1.2.1.2 8-puzzle

Another example of toy problem is the 8-digit puzzle, an instance of which is shown in Figure 1.6. It consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state in which tiles are ordered, such as the one shown on the right of the figure. A generic n -puzzle has $(n + 1)!/2$ reachable states; so for an 8-puzzle we have $9!/2 = 181440$ possibilities, but this number increases dramatically e.g. for a 15-puzzle ($\simeq 10^{13}$). The factor 2 in the number of states comes from a parity argument, which shows that half of the starting positions for the n -puzzle are impossible to resolve, no matter how many moves

are made. This is done by considering a function of the tile configuration that is invariant⁴ under any valid move, and then using this to partition the space of all possible labeled states into two equivalence classes of reachable and unreachable states.

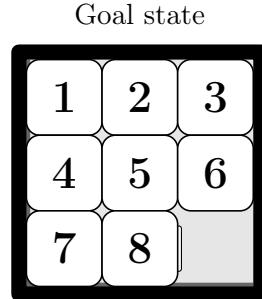


Figure 1.6: A typical instance of the 8-digit puzzle.

The family of sliding-block puzzles is known to be **NP-complete**. In computational complexity theory NP-complete problems are a class of problems whose solution can be guessed and verified quickly, i.e. in polynomial time (indeed NP stands for non-deterministic polynomial time⁵), but for which no efficient solution algorithm has been found. But what do “polynomial time” and “efficient” mean exactly? So-called easy, or tractable, problems can be solved by computer algorithms that run in polynomial time; i.e., for a problem of size n , the time or number of steps needed to find the solution is a polynomial function of n . On the other hand, algorithms for solving hard, or intractable, problems require times that are typically exponential functions of the problem size n . Polynomial-time algorithms are considered to be efficient, while exponential-time algorithms are considered inefficient, because the execution times of the latter grow much more rapidly as the problem size increases.

In summary, the general class of questions for which some algorithm can provide an answer in polynomial time is called “class P”. Instead, the class of questions for which an answer can be verified in polynomial time is called NP. It is easy to see that the complexity class P (all problems solvable, deterministically, in polynomial time) is contained in NP (problems where solutions can be verified in polynomial time), i.e. $P \subseteq NP$, because if a problem is solvable in polynomial time then a solution is also verifiable in polynomial time by simply solving the problem. But NP contains many more problems, the hardest of which are called NP-complete problems (“complete” in the sense of “most extreme”); if a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete. Thus, finding an efficient algorithm for any NP-complete problem would imply that an efficient algorithm can be found for all such problems, since any problem belonging to this class can be recast into any other member of the class. However, it is not known whether any polynomial-time algorithms will ever be found for NP-complete problems (P versus NP problem), and determining whether these problems are tractable or intractable remains one of the most important questions in theoretical computer science.

⁴In a 8-puzzle the invariant is the parity of the inversions. A pair of tiles is said to form an inversion if the values on the tiles are in reverse order with respect to their appearance in the goal state. In other words, we can linearize the sequence of tiles (ignoring the blank space) and two tiles a and b form an inversion if a precedes b in the sequence, but $a > b$ (practically we count how many lower numbers a tile precedes). It is easy to see that the parity of the inversion is invariant. In fact, when we slide a tile, we either make a row move or a column move: a row move doesn’t change the inversion count, whereas a column move can change it of -2 , 2 or 0 . As general rule, it follows that is not possible to solve an instance of 8-puzzle if the number of inversions is odd in the input state, while it is solvable if the number of inversions is even.

⁵Non-deterministic means that no particular rule is followed to make the guess.

1.2.1.3 8-queens problem

The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other (a queen attacks any piece in the same row, column or diagonal). This is a typical problem whose difficulty depends on the formulation.

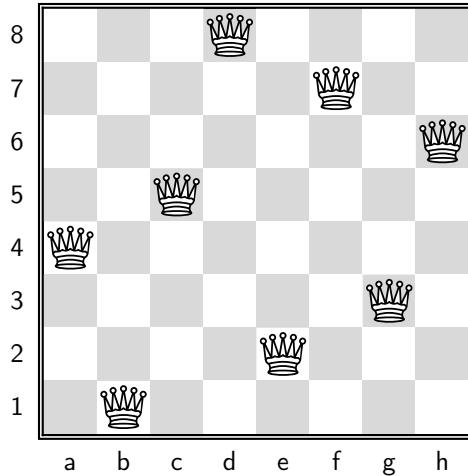


Figure 1.7: A possible solution of the 8-queens problem. In total there are 92 distinct solutions, but if we count as one solutions that differ only by the symmetry operations of rotation and reflection of the board, then we get just 12 fundamental solutions. See https://en.wikipedia.org/wiki/Eight_queens_puzzle, for instance.

- One way to approach the problem could be to consider a blank board and adding in succession all queens. The position of the first queen can be chosen among 64 empty squares; then, the second queen has at disposal $64 - 1 = 63$ locations, because one queen is already on the board and its position is inaccessible; by continuing in this fashion until we have placed all 8 queens on the board, we see that each progressive addition has at disposal one position less than the previous. Accordingly, in this incremental formulation, we have $64 \times 63 \times \dots \times 57 \sim 10^{14}$ possible configurations to investigate (only some of them are effectively solutions).
- A better formulation would prohibit placing a queen in any square that is already attacked in horizontal or vertical (forget about the diagonal for simplicity). Let's suppose to put the 8 queens already on the board all on different rows and columns. We start with the first queen in the leftmost column. Then, we put one by one the other queens in a column which is not yet occupied (this ensures no vertical attack); but in doing that, we also have to make sure that it cannot attack horizontally the previous ones. Accordingly, the first queen has 8 possibilities at disposal, the second 7 (8 minus the one vertically aligned with the first), the third 6 (8 minus 2), etc. In conclusion, in this formulation we have $8! = 40320$ possible configurations to investigate⁶, which is surely more comforting than the previous 10^{14} .

This is of course a big improvement, but for a 100-queens problem we have to investigate $100! \sim 10^{52}$, which is still enormous. Anyway, even 1 million queens can be solved by using a simple (but dedicated) algorithm!

⁶This procedure is similar to find the anagram of a 8-letter word: once fixed the different column, we can consider each queen's vertical position as a different position of a letter inside the string. The number of different anagrams of the string is indeed given by the permutation $8!$.

1.2.2 Real-world problems

Contrary to toy problems, a **real-world problem** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations. A typical example of real-world problem is the *route-finding problem*: suppose the agent has a map of Romania (Fig. 1.8), which provides him with information about the states it might get itself into and the actions it can take. Once given the initial state, the agent may be required to reach a certain destination respecting certain constraints (cost, time, etc.). So the route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications.

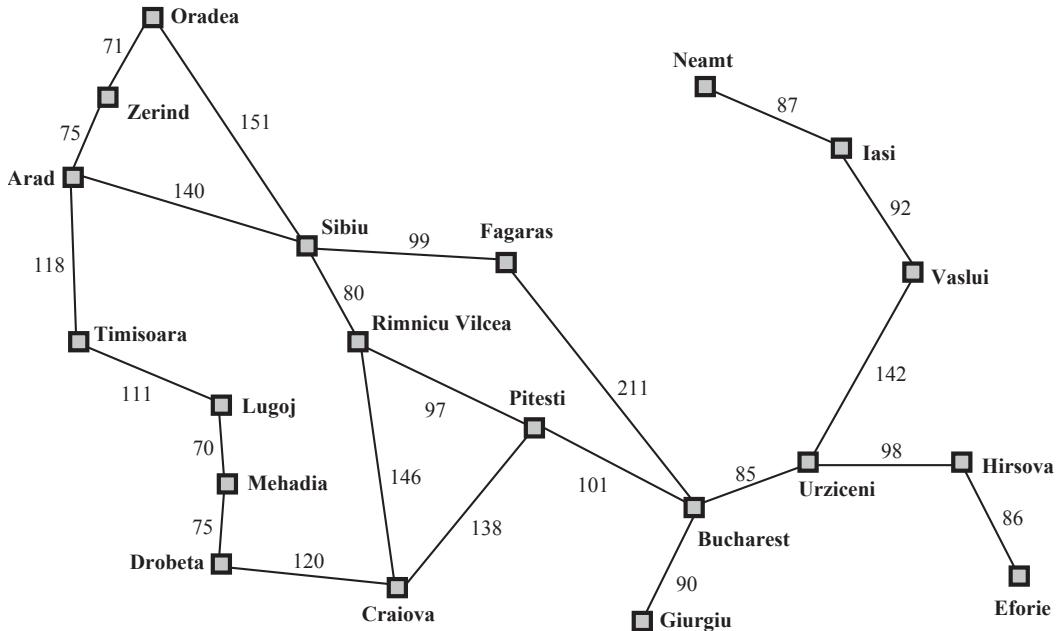


Figure 1.8: A simplified road map of part of Romania. Numbers are distances.

1.3 Setting a problem and searching solutions

Let's assume we are treating a static, fully accessible, discrete and deterministic environment. Generally speaking, a problem can be defined formally by four components:

- The *initial state* that the agent starts in (e.g. $\text{IN}[\text{TIMISOARA}]$).
- The possible *actions* the agent can take once given the present state. We can give a couple specifying the action and the successor, where successor refers to any state reachable from a given state by a single action. E.g. $(\text{Go}[\text{ARAD}], \text{IN}[\text{ARAD}])$.
- The *goal test*, which determines whether a given state is a goal state (e.g. $\text{IN}[\text{BUCHAREST}]?$). Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called “checkmate”, where the opponent’s king is under attack and can’t escape.
- The *path cost* function that assigns a numeric cost to each path. The single step cost to go from state x to y by the action a is denoted by $c(x, a, y)$. The cost can be the

1 Problem solving by uninformed and informed search

time spent, the distance traveled in a move, the steps required (e.g. in the 8-puzzle), etc. We assume that step costs are non-negative ($c \geq 0$).

The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm. A *solution* to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an *optimal solution* has the lowest path cost among all solutions.

But now the most important question: how to find solutions to a problem? How to set up a search algorithm capable of determining the solution? We have seen that a solution is an action sequence, so search algorithms work by considering various possible action sequences. The idea is to build up a search tree with the initial state at the root. Then we check if the initial state is the goal. If not, we expand the current state, that is we apply each legal action to the current state, thereby generating a new set of states. Now we can focus on one option (and put the others aside for later, in case the first choice does not lead to a solution) and we check again if it is the goal state and eventually we expand it. We obtain other child states, and at that point we can choose a non-expanded state and repeat the same procedure check/expand procedure, thus creating an ever increasing a *search graph*. The set of all states available for expansion (leaf nodes) at any given point is called the *frontier* or *fringe*, and naturally the process of expanding states on the frontier continues until either a solution is found or there are no more states to expand. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called search strategy.

Using this methodology we talked about states, but it would be more correct to talk about **nodes**. Each node of the tree basically contains five information (we can consider them as the definition of node):

- State: the state in the state space to which the node corresponds (n).
- Parent node: the node in the search tree that generated this node ($n - 1$). This is essentially the ability of tracking back.
- Action: the action a_n it was applied to the parent to generate the present node.
- Path-cost: the total cost from the initial state to the node: $g(n) = \sum_{i=1}^n c(i-1, a_i, i)$.
- Depth: the number of steps from the initial state.

Notice that in the search tree we can have repeated state, and hence we generate a loop. Considering such loopy paths means that the complete search tree is infinite because there is no limit to how often one can traverse a loop. Loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable.

1.3.1 Performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution, i.e. the lowest cost solution?

- Time needed: How long does it take to find a solution? Time is often measured in terms of the number of nodes generated (i.e. attempts) during the search.
- Memory needed: How much memory is needed to perform the search? In fact, once reached a node we must store in the memory all possibilities (i.e. all nodes) we have sifted up to that point, in order to continue.

The last two, also called time and space complexity, are always considered with respect to some measure of the problem difficulty. In AI complexity is expressed in terms of three quantities: b , the **branching factor** or maximum number of successors of any node; d , the **depth** of the shallowest goal node (i.e. the number of steps along the path from the root); m , the maximum length of any path in the state space (for loopy paths $m = \infty$).

1.4 Uninformed search

This section covers several search strategies that come under the heading of **uninformed search** or **blind search**. The term means we are not getting to the agent any type of information, beyond that provided in the problem definition. All the agent can do is to generate successors and distinguish a goal state from a non-goal state (it's not able to understand if its path is getting closer to the solution or not). All search strategies are distinguished by the order in which nodes are expanded. On the other hand, strategies that know whether one intermediate state is “more promising” than another are called **informed search**.

1.4.1 Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes at a given depth p in the search tree are expanded before any nodes at the next level $p + 1$ are expanded (Fig. 1.9).

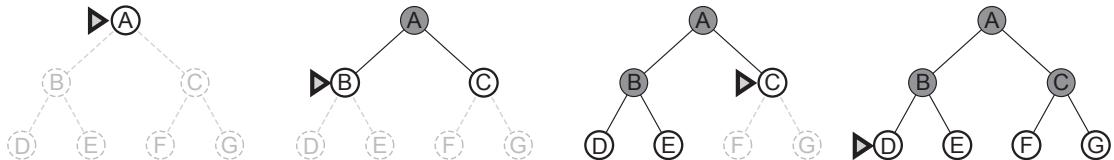


Figure 1.9: Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

How does breadth-first search rate according to the four criteria from the previous section? We can easily see that it is complete only if b is finite. Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now, the shallowest goal node is not necessarily the optimal one, because in term of total cost that goal node can be more expensive than a deeper goal node (the cost do not necessarily depend on the depth). The only possibility for the algorithm to find the optimal solution is when the cost increases with the depth p : $c(p+1) \geq c(p)$.

So far, the news about breadth-first search has been good. The news about time and memory is not so good, because all expanded nodes need to be kept in memory, and this is dramatic if the shallowest goal node is very deep. It is possible to demonstrate that time

1 Problem solving by uninformed and informed search

and memory complexity are

$$\sum_{p=1}^d b^p = b + b^2 + b^3 + \dots + b^d = O(b^{d+1}) \quad (1.4.1)$$

because if every state has b successors, then the root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on up to the depth d .

This is the general strategy, but from the computational point of view we need to store nodes in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a **queue**. In the breadth-first case we want to use a data structure that, when queried, gives us the *oldest* element, based on the *order* they were inserted. Therefore we use a so-called first-in, first-out queue (or simply **FIFO queue**), which pops the oldest element of the queue once investigated. In Figure 1.10 is shown an example of procedure: we initialize queue and visited arrays; then we start queuing the nodes that we can analyze, and once visited (i.e. goal-tested) according to the queue order we mark them as visited. The purpose of “Visited” is to avoid loops, i.e. to avoid visiting twice the same node if it can be reached in more than one way.

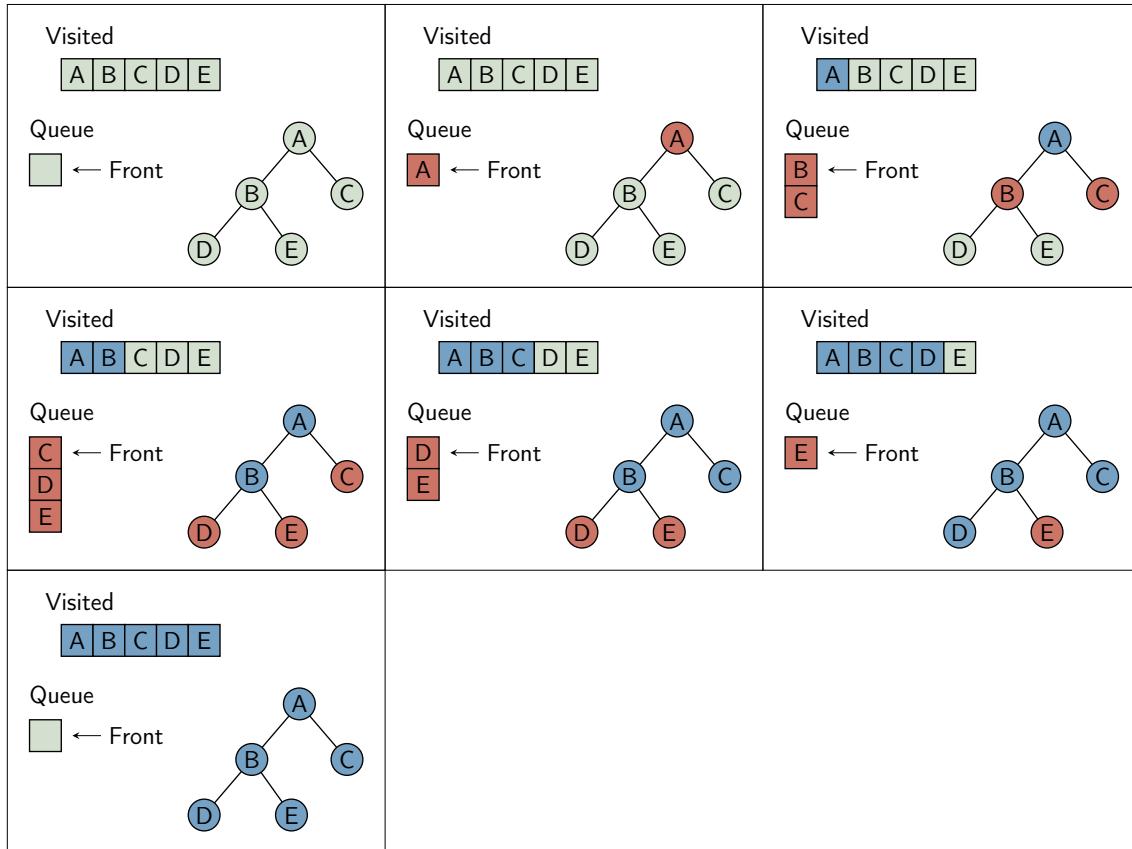


Figure 1.10: In order: initialize queue and visited arrays; queue node A into the queue; node A is marked as visited, queue B and C into the queue; mark B as visited, queue D and E into the queue; mark node C as visited, no node is queued because C has not adjacent nodes; mark node D as visited; mark E as visited, algorithm stops because E is the required node.

In Figure 1.11 is shown a little implementation of the Python code: `pop()` is an

inbuilt function in Python that removes and returns the last value from a list if the argument is empty or the given index value. Similarly, `append()` adds an item to the end of a list.

```

graph = {'A' : ['B', 'C'],
          'B' : ['D', 'E'],
          'C' : [],
          'D' : [],
          'E' : []}

# List to keeps track of visited nodes.
visited = []
# List to keep track of nodes currently in the queue.
# Initialize a queue (initially empty).
queue = []

def BFS(visited, graph, node):
    visited.append(node)
    queue.append(node)

    while queue:
        s = queue.pop(0)
        print(s, end = " ")

        for neighbor in graph[s]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)

# Driver code
BFS(visited, graph, 'A')

```

Figure 1.11: We built the graph using a so-called Python dictionary. A dictionary is a collection of data values in `key:value` pairs which is ordered.

1.4.2 Uniform-cost search

Instead of expanding the shallowest node, uniform-cost search expands the node with the lowest path cost. If all step costs are equal ($c = \text{const.}$), then breadth-first search coincides with uniform-cost search, and naturally that breadth-first search becomes optimal because it always expands the shallowest unexpanded node.

Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions. Completeness (and as a consequence optimality) is guaranteed provided the cost has a positive lower bound ϵ : for each action $c \geq \epsilon > 0$.

Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d . If C^* is the cost of the optimal solution, time and memory needed scale as $b^{C^*/\epsilon}$.

1.4.3 Depth-first search

Depth-first search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in Figure 1.12. The search proceeds immediately down to the deepest level of the search tree, where the nodes have no successors. When

1 Problem solving by uninformed and informed search

this happens, it means that branch we just have followed and analyzed is not satisfying; therefore all explored nodes with no descendants in the fringe are removed from memory (we cut the graph) and the search “backs up” to the next deepest node that still has unexplored successors.

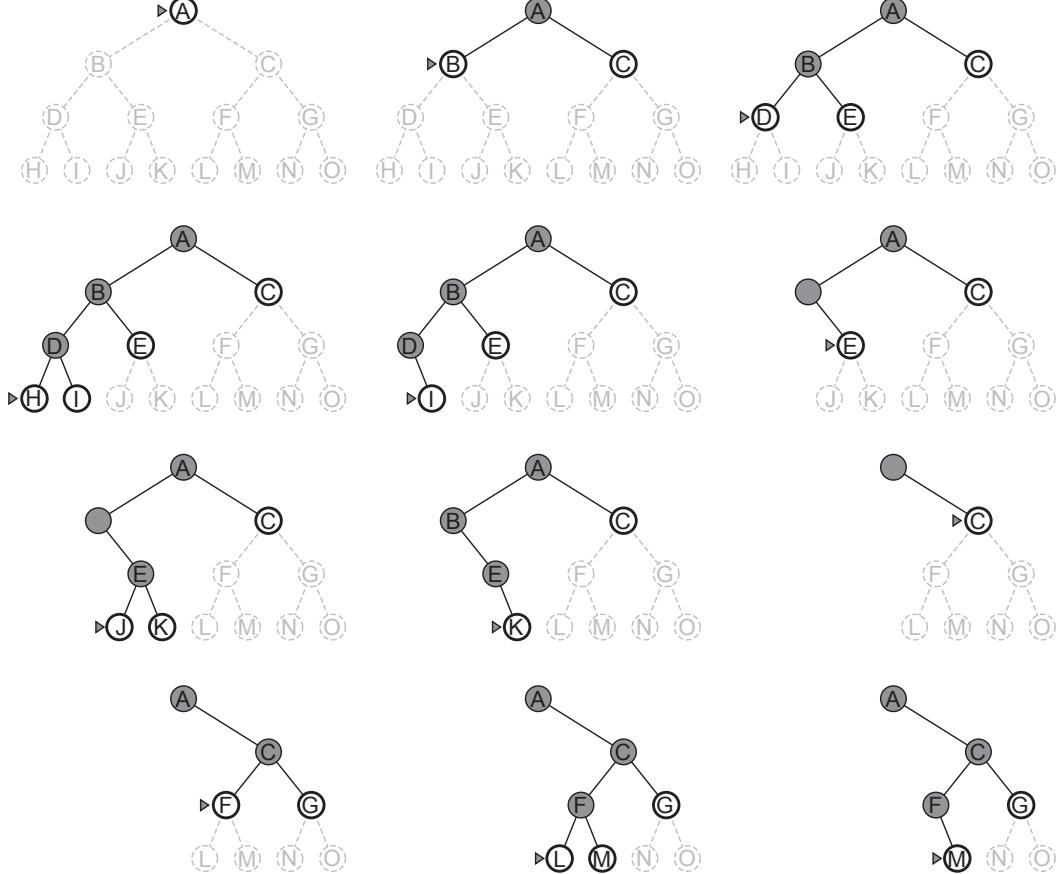


Figure 1.12: Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Since all dead nodes are removed from the memory, the memory needed for a depth-first search is significantly better than previous cases and goes as $O(bm)$ (if all successors are generated but not expanded). On the other hand, the time needed is $O(b^m)$ and hence is not so satisfying (actually it's the worst case) because m (maximum depth ever reached) can be much larger than d (depth of the shallowest solution). Furthermore, depth-first search it's not complete when $m = \infty$, because if we meet a never-ending path we run into a loop and we get stuck forever. Depth-first search is also not optimal.

From the computational point of view, depth-first search is different from breadth-first search, because now we always want to expand the deepest node of the fringe. Therefore, in the depth-first case we use a queue that, when queried, gives us the *newest* element that has been inserted. In this case we use a so-called last-in first-out queue, **LIFO queue** (also known as **stack**), which pops the newest element of the queue once investigated. To do this, we add the new node to the top through the command `insert(0, elem)` and remove it when visited through `pop(0)` (as before). In Figure 1.13 is shown an example of procedure.

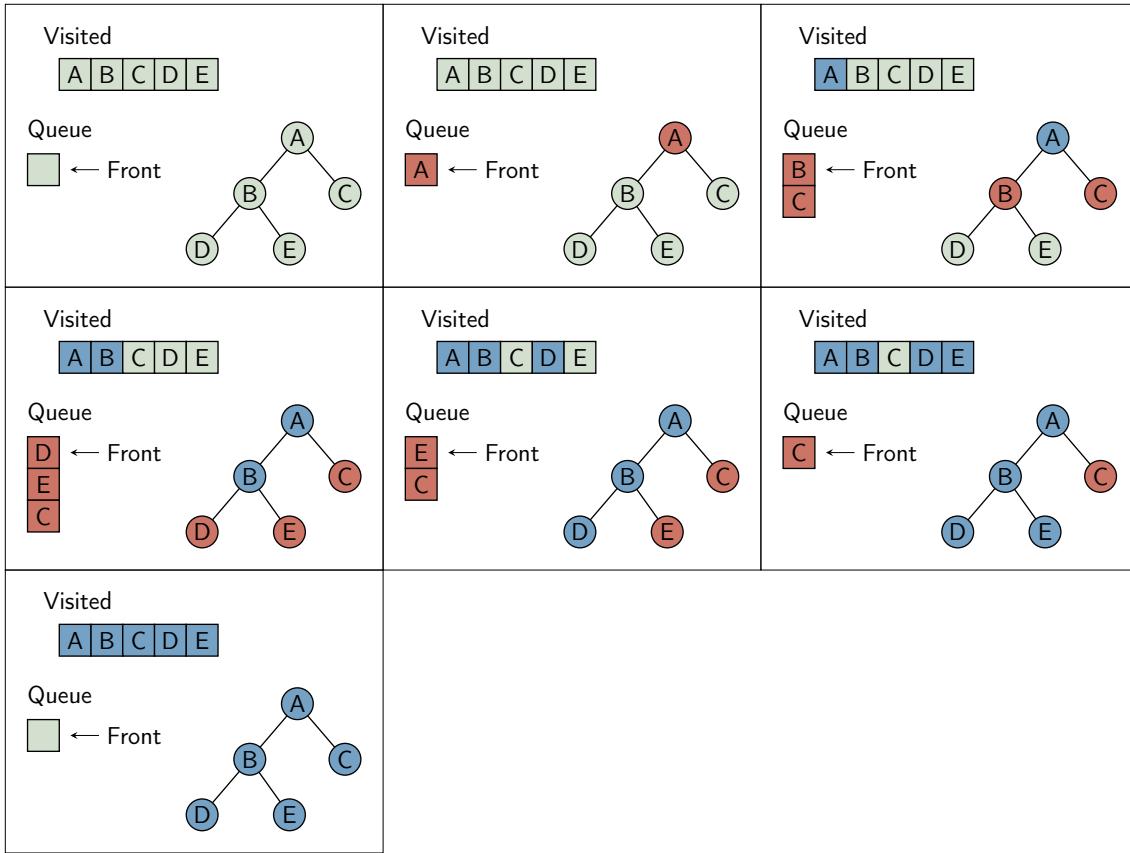


Figure 1.13: In order: initialize stack and visited arrays; push node A to the stack; node A is marked as visited, B and C are pushed into the stack; mark B as visited, D and E are pushed into the stack; mark node D as visited, no new node is put into the stack because D has not adjacent nodes; back track to node B, mark node E as visited and nothing goes into the stack; mark C as visited, algorithm stops because C is the required node.

1.4.4 Depth-limited search

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l . That is, nodes at depth l are treated as if they have no successors. This approach is called depth-limited search. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $l < d$, that is, when the shallowest goal is beyond the depth limit (this is likely when d is unknown). Depth-limited search is also be non-optimal. Similarly to depth-first search, the time needed is $O(b^l)$ and the memory needed is $O(bl)$. Naturally depth-first search can be viewed as a special case of depth-limited search with $l = \infty$.

1.4.5 Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy that finds the best depth limit. It does this by gradually increasing the limit l —first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit l reaches d , the depth of the shallowest goal node. The algorithm is shown in Figure 1.14.

Iterative deepening combines the benefits of previous strategies. Like depth-first search, its memory requirements are modest: $O(bd)$ to be precise. Like breadth-first search, it

1 Problem solving by uninformed and informed search

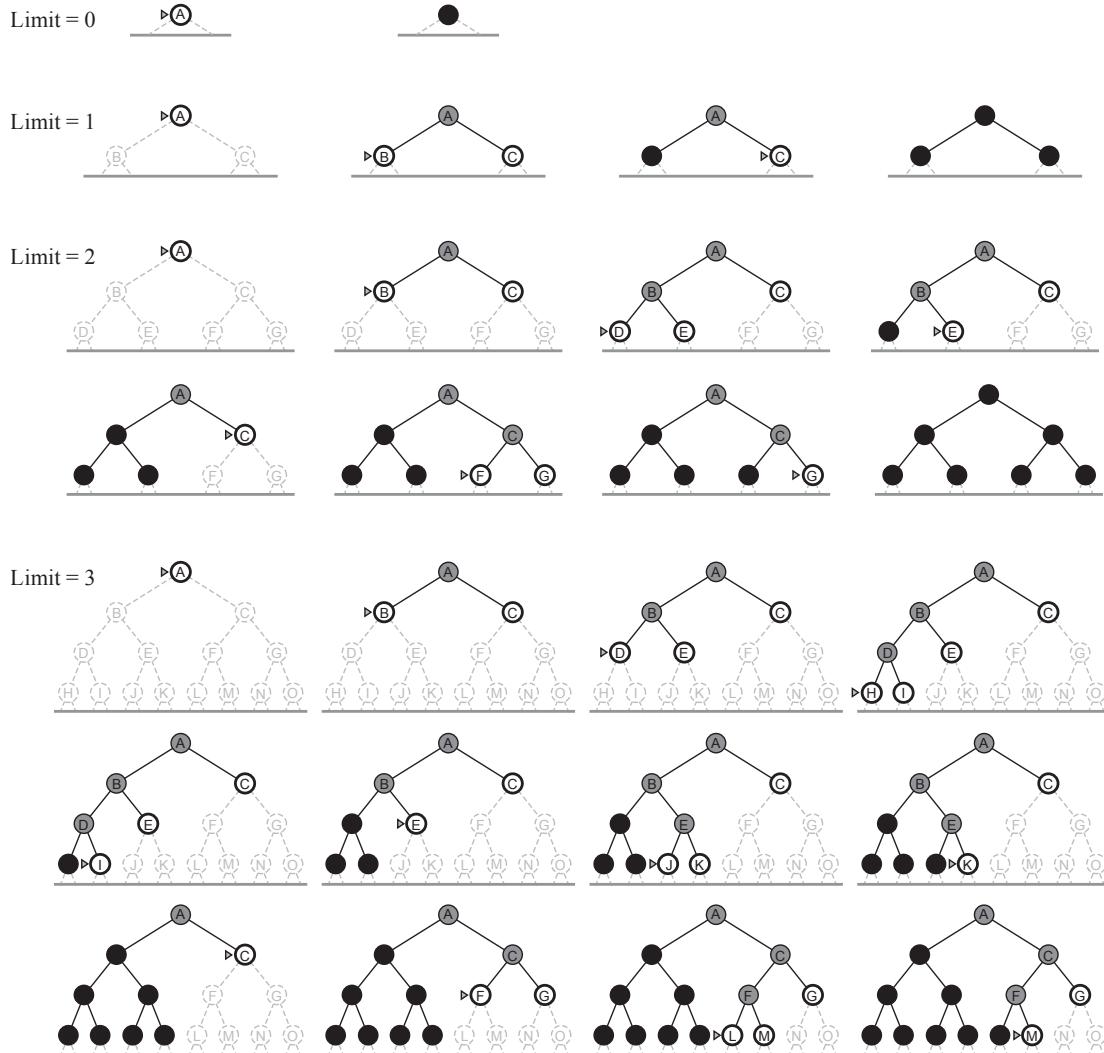


Figure 1.14

is complete when the branching factor b is finite and is optimal when the path cost is a non-decreasing function of the depth of the node ($c(p+1) \geq c(p)$).

From the point of view of time needed, iterative deepening search may seem wasteful because states are generated multiple times. However, it turns out this is not too costly: $O(b^d)$. The reason is that in a search tree with the same (or nearly the same) branching factor b at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

An alternative strategy of increasing path-cost limits can be formulated. It is not very efficient, but it constitutes the first step of the informed search algorithm A* (we will treat this later on).

Summarizing example

In [Figure 1.15](#) is provided an example of application of these 4 techniques (breadth-first, depth-first, depth-limited, iterative deepening) to the case of the Romania road map in Figure 1.8.

Distances between couples of cities are taken from an external plain text file. Be careful that in the depth-first case we handled the stack in the opposite way with respect to the previous drawing in Figure 1.13: here we added new nodes at the bottom of the stack (`append()`) and we also explored them from the bottom (`pop()`).

1.4.6 Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle (Figure 1.15). The motivation is that

$$b^{d/2} + b^{d/2} \ll b^d \quad (1.4.2)$$

or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal. This is certainly a smart idea, but it is not always possible since the goal need to be known. For instance, in game of chess we are not able to search back from goal state (checkmate).

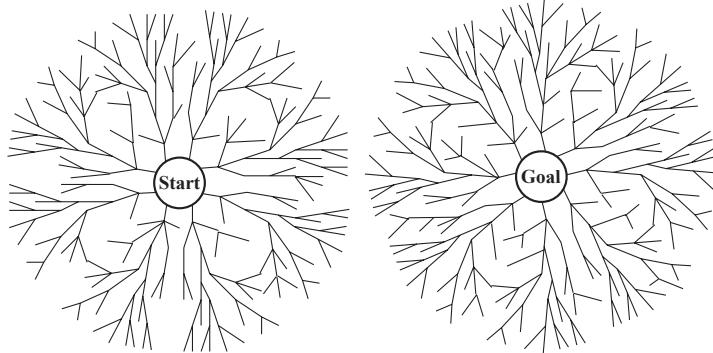


Figure 1.15: schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

1.5 Heuristic functions and informed search

This section shows how an informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy. The general approach we consider is called *best-first search*. Best-first search is an instance of the general tree search algorithm in which a node n is selected for expansion based on an **evaluation function** $f(n)$ (f is the guideline for the expansion). The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The choice of f determines the search strategy. Most best-first algorithms include as a component of f a **heuristic function**, denoted $h(n)$. In particular, the heuristic function is the *estimated cost of the cheapest path from the state at node n to a goal state*. For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest. Note the term “estimated”; in fact, if we knew exactly the cost we would have solved the problem. For now, we consider them to be arbitrary, non-negative, problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$.

1.5.1 Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it expands first nodes having the smallest $h(n)$ down till the end ($f(n) = h(n)$). Let us see how this works for route-finding problems in Romania; we use the *straight-line distance* heuristic function. If the goal is Bucharest, we need to know the straight-line distances of every city to Bucharest, which are shown in Table 1.4.

City	Dist.	City	Dist.
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Table 1.4

Figure 1.16 shows the progress of a greedy best-first search using h to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using the straight-line distance finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is *not optimal*, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.

Greedy best-first tree search is also *incomplete* even in a finite state space, much like depth-first search. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end. The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras. The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop.

1.5.2 A* search

The most widely known form of best-first search is called **A* search** (pronounced “A-star search”). It evaluates nodes by combining $g(n)$, the real cost from the starting point to reach the node n , and $h(n)$, the estimated cost to get from the node to the goal:

$$f(n) = g(n) + h(n) \quad (1.5.1)$$

f is the real cost from the beginning to reach n plus an estimate of what remains to be paid from n to the goal. It turns out that this strategy is more than just reasonable: provided

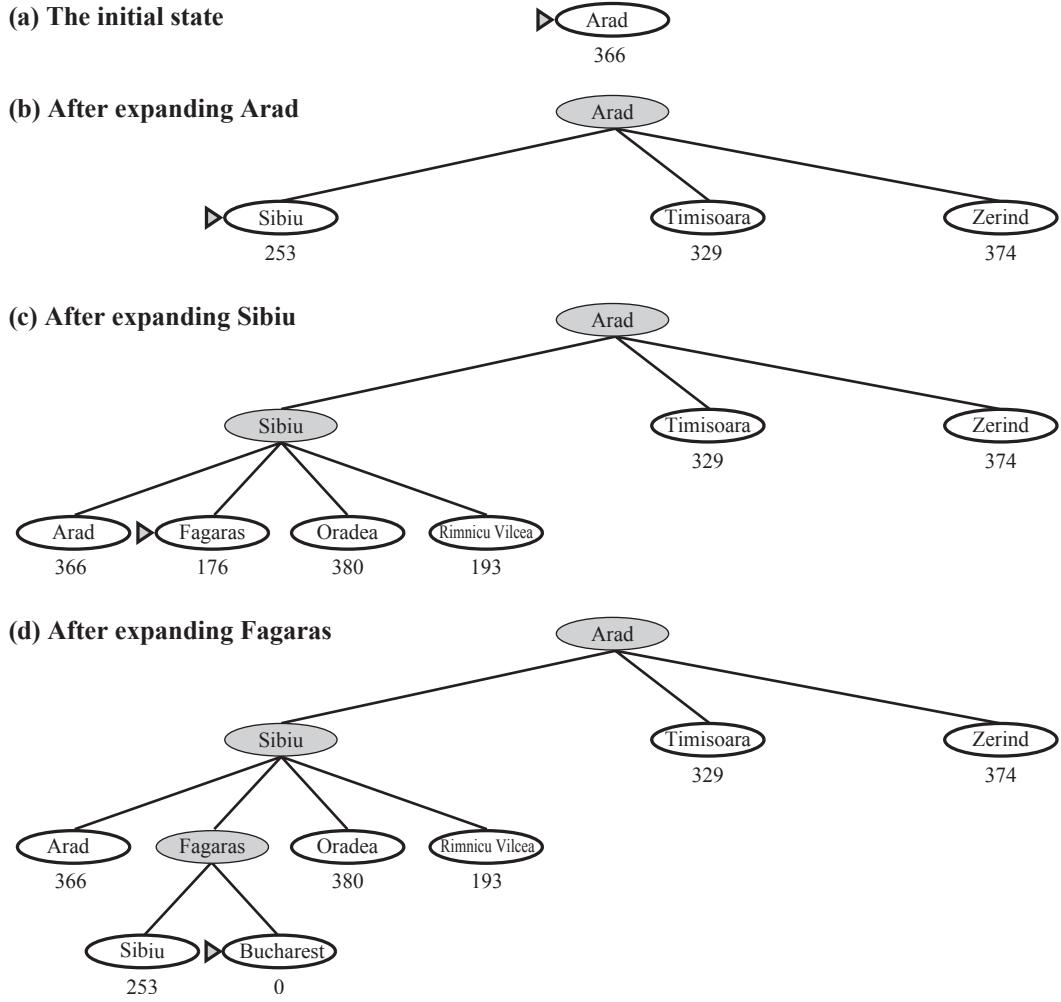


Figure 1.16

that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.

In particular, it is complete if $c \geq \varepsilon > 0$ and b is finite. In fact, the cost increases at least by ε at each step and since b is finite, then the total cost will eventually exceed the cost of the solution if a solution exists.

Optimality

The first condition we require for optimality is that $h(n)$ be an **admissible** heuristic function. An admissible heuristic function is one that *always underestimates* the cost to reach the goal:

$$h(n) \leq C(n, G) \quad (1.5.2)$$

An obvious example of an admissible heuristic is the straight-line distance that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

Proof. Suppose G_2 is not the optimal goal on the fringe; still being a goal, we have by definition $h(G_2) = 0$. Then the evaluation function reads

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* \quad (1.5.3)$$

where C^* is the cost of the optimal solution corresponding to the node G^* . Instead, let n be a node on the fringe along the optimal solution. Then, by hypothesis of h being admissible we can write

$$f(n) = g(n) + h(n) \leq g(n) + C(n, G^*) = C^* \quad (1.5.4)$$

because the last sum is the total real cost to reach the optimal goal from the beginning (i.e. from the root node). Therefore, from (1.5.3) and (1.5.4) we obtain the condition

$$f(n) \leq C^* < f(G_2) \quad (1.5.5)$$

But since n and G_2 are along the same fringe and A* search works by expanding nodes according to the minimization of the evaluation function, then this means that G_2 is not explored. And this happens for any other non-optimal solution G_i , because, having a lower estimate, the algorithm always prefers to visit the node n , instead of them. In other words, A* expands no nodes with $f(n) > C^*$. In this way the search avoids non-optimal solutions, thus always reaching the optimal goal (lowest cost solution), which is the condition of optimality. \square

A second, slightly stronger condition called **consistency** can be required to ensure optimality. A heuristic function $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n :

$$h(n) \leq c(n, a, n') + h(n') \quad (1.5.6)$$

This is a form of the general *triangle inequality*, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance that we have seen before, for instance.

It is fairly easy to show that every consistent heuristic is also admissible. Consistency is therefore a stricter requirement than admissibility, but one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent.

Proof. We want now to demonstrate that consistency implies admissibility, i.e. that the heuristic function always underestimates the cost to reach the goal. Let a_i be a sequence of actions along an optimal solution. Then, from the triangle inequality we may write

$$\begin{aligned} h(n) &\leq c(n, a_1, n_1) + h(n_1) \\ &\leq c(n, a_1, n_1) + c(n_1, a_2, n_2) + h(n_2) \\ &\leq c(n, a_1, n_1) + c(n_1, a_2, n_2) + \dots + c(n_{d-1}, a_d, G^*) + h(G^*) \\ &= C(n, G^*) + h(G^*) \\ &= C(n, G^*) \end{aligned} \quad (1.5.7)$$

because the sum of the step costs is the total cost along the best path and $h(G^*) = 0$ by definition. So we obtained that $h(n) \leq C(n, G^*)$, which means that a consistent heuristic function is also admissible (1.5.2). \square

Clearly, consistency implies admissibility, which in turn implies optimality. Let's see a different proof to demonstrate that consistency \Rightarrow optimality.

Proof. Let a be an action connecting n to the successor n' , along whatever path. Then $g(n') = g(n) + c(n, a, n')$ for some action a , and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) \quad (1.5.8)$$

where the last step follows from the triangle inequality. Therefore f is a not-decreasing function (or, said otherwise, it can only increase) along whatever path. So A* expands nodes having a not-decreasing cost and it expands *all* nodes having $f(n) < C^*$. Accordingly, the first goal it reaches has also the minimal cost. This is the best path, i.e. the first time we reach the goal, because after f would increase even more. \square

The fact that f -costs are increasing along any path also means that we can draw *contours* in the state space, just like the contours in a topographic map (Figure 1.17). Inside the contour labeled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A* expands the fringe node of lowest f -cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f -cost. In general the time needed still grows exponentially, unless h makes only logarithmic errors.

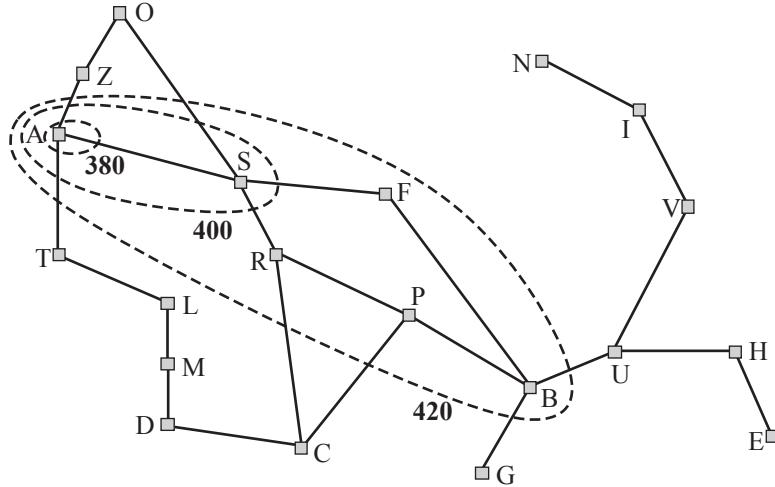


Figure 1.17

1.5.3 Inventing heuristic functions

We have seen before that the straight-line distance is a good idea if we want to minimize the distance covered. However, in general is not so easy and immediate to build a suitable heuristic function for whatever problem. A way out is to consider a “relaxed” version of the original problem, that is a simpler problem in which some aspect would not be allowed in the original one. If we are able to solve this new simpler version, then we can use the cost of an optimal solution as the heuristic function for the original problem.

For example, consider the 8-puzzle. As mentioned in previous sections, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration. Consider in particular the starting position and the goal indicated in Figure 1.18. This particular configuration has an optimal cost of $C^* = 26$ moves. Now, there are two commonly used ways to simplify the problem.

1. Tiles can jump in any direction with arbitrary steps into the blank space (misplaced tiles). In our specific case all of the eight tiles are out of position, so the start state

1 Problem solving by uninformed and informed search

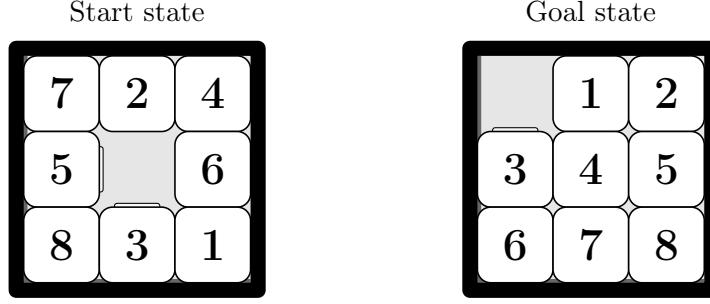


Figure 1.18

would reach the solution with the minimum cost $h_1 = 8$. h_1 is an admissible heuristic function for the original problem, because it is clear that any tile that is out of place must be moved at least once.

2. Tiles can move by one step also over already occupied locations. In this case the minimum cost is $h_2 = 18$ steps; this count of the number of steps is sometimes called **Manhattan distance**. h_2 is also admissible because all any move can do is move one tile one step closer to the goal.

In both cases, h_1 and h_2 underestimate the cost C^* of the original problem, so they are both admissible heuristic functions for the original problem. But which of the two is the best? If the estimate of the heuristic function was fortuitously equal to the real cost, then the problem would be already solved because we would exactly know how to browse the search tree. Therefore, we expect h_2 to be better than h_1 in terms of performance, because h_2 is closer to C^* than h_1 , and hence we need to explore less nodes in order to reach the solution. We thus say that h_2 dominates h_1 .

In Table 1.5 is shown a comparison of the efficiency of a blind search strategy (iterative deepening) to an A* search in terms of nodes generated and depth d of the solution. Notice how iterative deepening, which is still effective, is by far less efficient than A*(h_1), and the latter in turn of A*(h_2).

d	Search cost (nodes generated)			Effective branching factor		
	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	16416	—	1.48	1.26

Table 1.5

Another way to characterize the quality of a heuristic function is the **effective branching factor** b^* . If the total number of nodes generated by A* for a particular problem is N and the solution depth is d , then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes. Thus

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d \quad (1.5.9)$$

The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. Therefore, experimental measurements of b^* on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b^* close to 1, allowing fairly large problems to be solved at reasonable computational cost.

In  we provided the written implementation of A* search applied to the 8- and 15-puzzles. An interesting thing to notice is how the Manhattan distance, which is the minimum distance in steps between the starting board and the goal board, is extrapolated. First let `LEN` be the side of the puzzle (3 for the 8-puzzle, 4 for the 15-puzzle). Furthermore, starting from the top left tile of the board and continuing to the right and then down, we have indexed the tiles in the following way:

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \quad (1.5.10)$$

as if the whole object were a one-dimensional array. Be careful not to confuse these indices, which are the indices of the Python array, with the numbers 1 to 8 printed *on* the tiles. The starting point for calculating the Manhattan distance is to consider the various tiles one at a time and calculate how many steps the number printed on them should move to reach the corresponding goal position. To do this, we need to calculate the difference in the x position and in the y position of the initial and final configurations. Keeping in mind the way in which we indexed the board, however, we note that

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \bmod 3 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix} \quad (1.5.11)$$

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \text{div } 3 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix} \quad (1.5.12)$$

that is, by applying the modulo operation to an index we obtain its horizontal coordinate (x position), while by applying the integer division we obtain its vertical coordinate (y position). Therefore, we take a tile (for example the one with the number 4 printed on) and we calculate which indices `i` and `index` it corresponds to in the initial and final configurations, respectively. From the practical point of view, we first take an index `i` of a tile in the initial board and the function `self.board[i]` returns the number there located. Instead, `index = WINNING_BOARD.index(self.board[i])` is the goal index where the number `self.board[i]`, initially in position `i`, has to arrive. Finally, based on everything we have just said, the Manhattan distance is computed as `abs(i % LEN - index % LEN) + abs(i // LEN - index // LEN)` where the first term is Δx and the second is Δy ; here `%` and `//` are the standard syntaxes in Python for the modulo operation and integer division respectively. We repeat this procedure for each index and then we add all together.

1.5.4 Learning through experience

Another solution to the problem of inventing an adequate heuristic function is to learn from experience. “Experience” here means solving many times the same problem, i.e. solving lots of 8-puzzles, for instance. By solving many times a problem one can find correlations between certain characteristics of a state and the cost of the solution (e.g. a correlation between the number of misplaced tiles and the cost). Each optimal solution to a problem provides examples from which $h(n)$ can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search. Techniques for doing just this typically use neural nets and other methods.

1.6 Local search

The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. When a goal is found, the path to that goal also constitutes a solution to the problem. In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added. If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable. In addition to finding goals, local search algorithms are useful for solving pure *optimization problems*, in which the aim is to find the best state according to an objective function.

To understand local search, we find it useful to consider the state-space landscape (as in Figure 1.19). A landscape is just a plot in which the x -axis (or \mathbf{x} , if in more dimensions) corresponds to the current state of the problem, while the y -axis is defined by the value of the heuristic cost function or objective function (elevation). If elevation corresponds to cost, then the aim of local search is to find the lowest valley, i.e. a global minimum; if elevation corresponds to an objective function, then the aim is to find the highest peak, i.e. a global maximum (you can convert from one to the other just by inserting a minus sign). Local search algorithms explore this landscape.

The simplest way to explore the landscape in search of the goal is through a **hill-climbing** search algorithm (steepest-ascent version), which is simply a strategy that continually moves in the direction of increasing value, that is uphill. It terminates when it reaches a “peak” where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state. Hill climbing is sometimes called *greedy local search* because it grabs a good neighbor state without thinking ahead about where to go next. Generally, greedy algorithms often perform quite well; hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. Unfortunately, hill climbing often gets stuck for the following reasons.

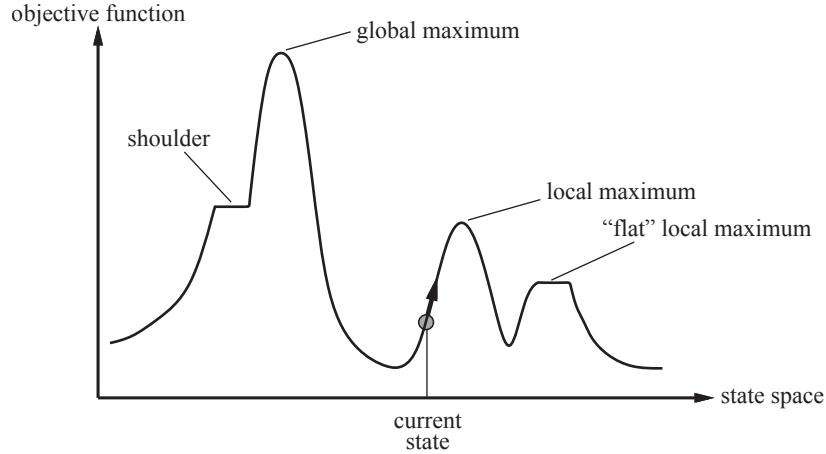


Figure 1.19

- **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.

For example, in Figure 1.20a is shown an 8-queens state with heuristic cost estimate $h = 17$, where h is the number of couples of queens seeing (attacking) one the other. The numbers on the chessboard indicate the value of h for each possible successor state obtained by moving a queen within its column in a specific location. The best moves ($h = 12$) are marked. In Figure 1.20b we have instead a local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost. Here hill-climbing search would get stuck forever because each subsequent move would increase the cost h .

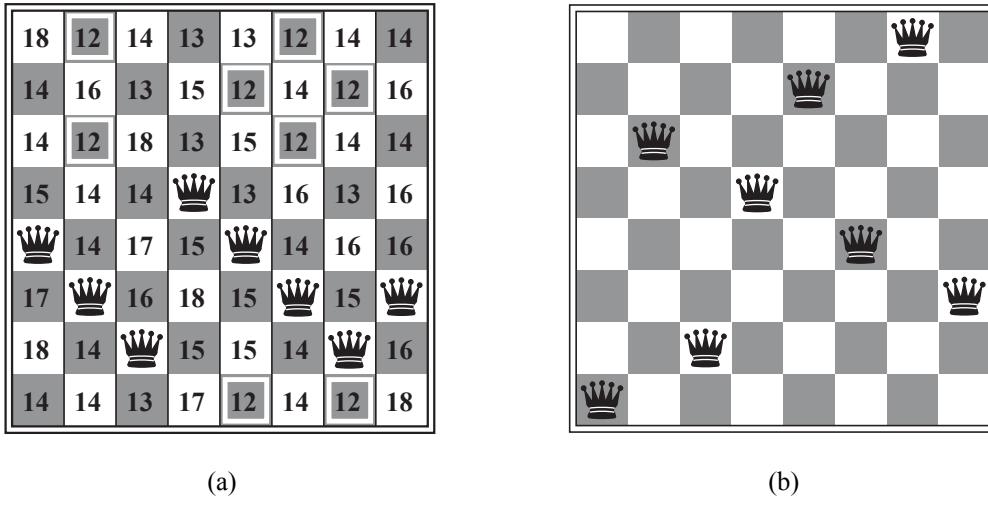


Figure 1.20

- **Plateaux:** a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau because it doesn't know in which direction it is better to move.

- **Ridges:** ridges are particular structures that arise in multidimensional problems and can become even worse in high dimensions. They result in a sequence of rising local maxima in which every neighbor appears to be downhill (Fig. 1.21). The grid of states (cyan circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. In fact, from each local maximum all the available actions point downhill. That's why ridges are very difficult for greedy algorithms to navigate.

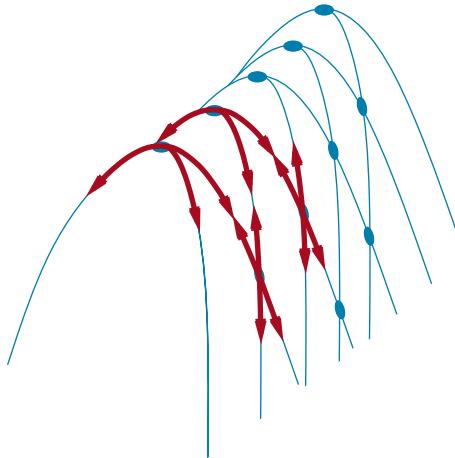


Figure 1.21

Thus a hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck. There are some typical ways to improve the strategy.

- **Random-restart hill climbing:** if it gets stuck, it restarts from a random generated initial state.
- **Monte Carlo techniques**, in particular *simulated annealing*: random walk and down-hill moves dictated by a certain probability.
- **Local beam search:** it begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. But, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the parallel search threads. At each step the algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

- **Stochastic beam search:** similar to local beam search, but instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.
- **Genetic algorithms:** combines two parents and tries to mimic natural selection.

1.6.1 Genetic algorithms

A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining parent states rather than by modifying a single state, in a way which is very similar to Darwin's natural selection. We call **mixing number** ρ the number of parents that come together to form offspring. The most common is $\rho = 2$: two parents combine their "genes" (parts of their representation) to form offspring. When $\rho = 1$ we have stochastic beam search, which can be seen as asexual reproduction. It is possible also to have $\rho > 2$, which occurs only rarely in nature but is easy enough to simulate on computers.

Genetic algorithms are essentially structured in three steps. The first one is the **selection** process for selecting the individuals who will become the parents of the next generation. One possibility is to select from all individuals with probability proportional to their fitness score (a GA terminology for the objective function): a *fitness function* should return higher values for better states. Another possibility is to randomly select n individuals ($n > \rho$), and then select the most fit ones as parents.

The second step is the **recombination** or **crossover** procedure, in which the parents chosen from the population mix their genes. One common approach to do that (assuming $\rho = 2$), is to randomly select a crossover point to split each of the parent strings, and recombine the parts to form two children: one with the first part of parent 1 and the second part of parent 2; the other with the second part of parent 1 and the first part of parent 2.

Finally, in analogy with Nature it is possible for the children to undergo random **mutations** of their genetic code. To include these natural fluctuations, we introduce the *mutation rate*, which determines how often offspring have random mutations to their representation. Once offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.

Once concluded these steps, the algorithm proceeds iteratively with the makeup of the next generation. Usually, this is just the newly formed offspring, but there are many variants of this selection rule. The next parents can also include a few top-scoring parents from the previous generation. In other words, a small portion of the best individuals from the last generation can be carried over (without any changes) to the next one (they are so good and promising that we don't want to waste them). This practice is called **elitism**, and guarantees that the overall fitness will never decrease over time. However care must be taken not to overdo this practice because you risk getting stuck in local minima. Another particular practice which can lead to a speedup of the process is the **culling**, in which all individuals below a given threshold are discarded.

Figure 1.22 shows an example of this procedure applied to the 8-queens problem. We begin with a population (a) of four randomly chosen states. Each state, or individual, is represented as a string of numbers where the order in which they are inserted reflects the queen's column on the chessboard and the numbers reflects the rows. For instance, a string like 24748552 means that the queen in column 1 is in the 2nd rows, the queen in column 2 is in the 4th rows, etc. Then, we must rate our states by the fitness function (b), which in our case is the number of non-attacking pairs of queens (28 for a solution). The values of the four states are 24, 23, 20 and 11, and we can convert them into percentage of being chosen as parents. According to these probabilities, two pairs are selected at random for reproduction (c) (it's like a beauty competition: the most beautiful are encountered more). Notice that one individual is selected twice and one not at all. For each pair to be mated, we randomly chop the genetic tracks at the same place and we switch them (d). From Figure 1.23 we see that this corresponds to glue two pieces of different board configurations. Finally (e), each location is subject to random mutation with a small independent probability. One digit was

1 Problem solving by uninformed and informed search

mutated in the first, third and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

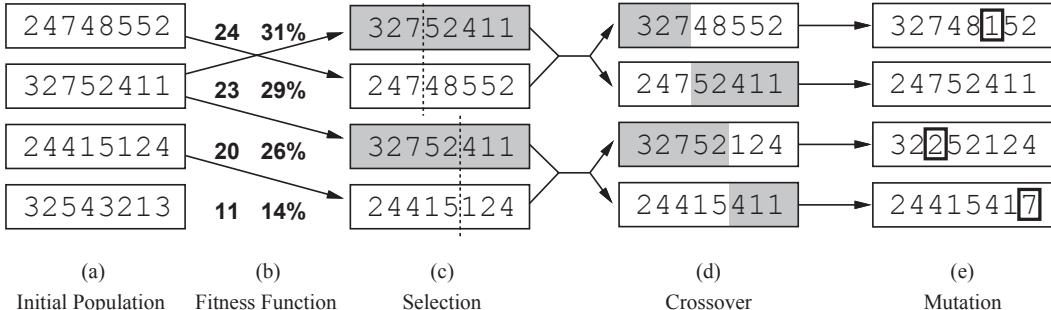


Figure 1.22

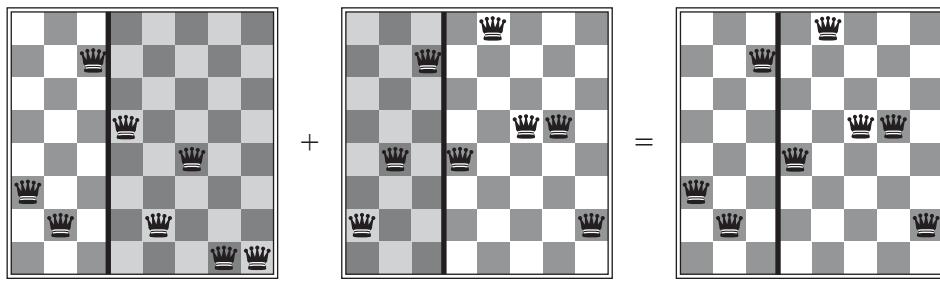


Figure 1.23

One question that may arise at this point is: why do these methods work quite effectively? Actually, these algorithms work only if there is some real connection between the elements we are trying to optimize. In other words, these methods work very well if they recognize in the system some sort of substructure (some sub-blocks) that after can be put together (a so-called **schema**). If we don't have such internal organization, then these methods do not give any advantage over brute-force search because the machine has essentially nothing to learn.

In Python genetic algorithms can be handled in a simpler way through the dedicated library **geneticalgorithm**. As written at the corresponding web page: “**geneticalgorithm** is a Python library distributed on PyPI for implementing standard and elitist genetic algorithms. This package solves continuous, combinatorial and mixed optimization problems with continuous, discrete and mixed variables. It provides an easy implementation of genetic algorithm in Python”. To install it use the package manager pip or type the line `pip install geneticalgorithm` in the terminal of your machine, provided pip is already installed. I should recommend to visit the pages <https://pip.pypa.io/en/stable/> and <https://pypi.org/project/geneticalgorithm/> for the complete procedure and documentation.

Anyway, let's see two very simple examples. In the first one assume we want to find a set \mathbf{x} that minimizes the function $f(\mathbf{x}) = x_1 + x_2 + x_3$, where $\mathbf{x} = (x_1, x_2, x_3)$ can be any real number⁷ in $[0, 10]$. This is a trivial problem and we already know that the answer is $\mathbf{x} = (0, 0, 0)$, where $f(\mathbf{x}) = 0$. However, notice that the algorithm provides an approximate solution. The library also gives us a plot showing the way in which the search reaches the goal.

⁷In principle we can also use integer variables in order to decrease the computation speed (a program indeed discretizes meticulously continuous variables, and this can generally slow down the process).

In the previous example the optimization problem was unconstrained. Instead, the second example  also allows to have an extra constraint so that the sum of x_1 and x_2 is equal or greater than 2. Obviously, the minimum of $f(\mathbf{x})$ is 2. In such a case, a trick in the code is to define a penalty function, which is added to the objective function $f(\mathbf{x})$. In our example the penalty is 0 if we are exploring the allowed region, while it is very large if we are in the forbidden region. Usually you may use a constant greater than the maximum possible value of the objective function if the maximum is known or if we have a guess of that. This helps the algorithm learn how to approach feasible domain.

1.6.1.1 Traveling salesman problem

The traveling salesman problem (TSP) is a famous problem which tries to give an answer to the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?” Even though the problem is computationally difficult, many heuristics and exact algorithms are known. The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest using brute-force search, but this is clearly impractical due to its enormous amount of time needed (once fixed the starting point, there are $(N - 1)!/2$ possible routes to investigate).



Figure 1.24: Solution of the traveling salesman problem applied to the 48 lower capitals of United States (i.e. forgetting Alaska and Hawaii).

In the following we'll provide a solution based on genetic algorithms, even if there are more suitable and appropriate ways to implement it, taken from <https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>. The complete notebook of the code can be consulted here . First of all, let's start with some definitions, rephrased in the context of the TSP.

- Gene: a city (represented as (x, y) coordinates).
- Individual: a single route satisfying the conditions above.
- Population: a collection of possible routes (i.e. collection of individuals).

1 Problem solving by uninformed and informed search

- Parents: two routes that are combined to create a new route.
- Mating pool: a collection of parents that are used to create our next population (thus creating the next generation of routes).
- Fitness: a function that tells us how good each route is (in our case, how short the distance is).
- Mutation: a way to introduce variation in our population by randomly swapping two cities in a route.
- Elitism: a way to carry the best individuals into the next generation.

As usual, our genetic algorithm will proceed through the following steps: (1) creation of the population; (2) determination of fitness; (3) selection of the mating pool; (4) breeding; (5) mutation; (6) repetition.

For our purposes, we first create a `City` class⁸ that will allow us to create and handle our cities in an easier way. These are simply our (x, y) coordinates. Within the `City` class we also add two methods, which are a distance calculation (making use of the Pythagorean theorem) and a cleaner way to output the cities as coordinates. For instance, if we define `u=city(10, 20)` and `v=city(20, 40)`, then `city.distance(u, v)` will give the distance between the two cities labeled `u` and `v`.

We'll also create a `Fitness` class in order to evaluate new candidate paths. In our case, we'll treat the fitness as the inverse of the route distance. Our goal is to minimize route distance, so a larger fitness score is better. Note also that, based on the definition of the problem, we also need to start and end at the same place. Practically, this means that we need to close the path when we have added together all the cities, and this is achieved through the lines `else: toCity = self.route[0]`. In fact, we are systematically adding together the various paths, but if our city index is larger than the total number of cities (meaning we have already connected all cities in some way), then we go back to the starting point.

We can now make our initial population (aka first generation). So we take our city list `cityList`, which is a collection of couples (x, y) , and we randomly reshuffle the order in which each city appear through `random.sample(cityList, len(cityList))`. In this way we obtain another list of cities having the same length but with the single elements selected randomly. This produces one individual, but we want a full population, so we loop the `random.sample` function until we have as many routes as we want for our population (`popSize`). We append new individuals to the `population` list.

Next, the evolutionary fun begins. To simulate our “survival of the fittest”, we can make use of `Fitness` to rank each individual in the population. Then the `sorted()` function sorts the elements of a given iterable in a specific order (either ascending or descending) and returns the sorted iterable as a list. Thus, our output will be an ordered list with the route IDs and each associated fitness score. In other words, we have ranked our sample according to their fitness.

There are a few options for how to select the parents that will be used to create the next generation. The most common approaches are:

⁸Roughly speaking, Python classes are structures that allows to organize objects according to common attributes and properties. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class. Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructor in object oriented programming. We normally use it to initialize all the variables. Within a class are usually defined also some “methods” which are functions acting within the class. See [→](#) for an extremely simple practical example.

- *Fitness proportionate selection*: the fitness of each individual relative to the population is used to assign a probability of selection. Think of this as the fitness-weighted probability of being selected.
- *Elitism*: with elitism, the best performing individuals from the population will automatically carry over to the next generation, ensuring that the most successful individuals persist.

For the purpose of clarity, we'll create the mating pool in two steps. First of all we re-scale all fitness scores of our sample in order to get for each path a probability between 0 and 1 of being chosen (`100*df.cum_sum/df.Fitness.sum()`; this is actually a cumulative percentage). Then we compare a randomly drawn number to these weights to select our mating pool (`100*random.random()`). Finally, we'll also want to hold on to our best routes, so we introduce elitism. The number of elite individuals is `eliteSize`, hence `len(popRanked) - eliteSize` is the size of the mating pool that will really undergo crossover.

With our mating pool created, we can create the next generation in the process called crossover (or “breeding”). If our individuals were strings of 0s and 1s and our initial constraints (each city visited only once and coinciding starting and end points) didn't apply, we could simply pick a crossover point and splice the two strings together to produce an offspring. However, the TSP is unique in that sense because we need to *include all locations exactly once*. To abide by this rule, we can use a special breeding function called *ordered crossover*. In ordered crossover, we randomly select a subset of the first parent string and then fill the remainder of the route with the genes from the second parent in the order in which they appear, without duplicating any genes in the selected subset from the first parent. Referring to Figure 1.25, we extract two random numbers indicating the chopping positions (`int(random.random() * len(parent1))`); then we append that portion to an empty list `childP1` in the same position that it was before (`childP1.append(parent1[i])`), and we fill the remaining empty slots with the genes that are not present in the portion just inserted (`childP2 = [item for item in parent2 if item not in childP1]` and `child = childP1 + childP2`). Next, we'll generalize this to create our offspring population. We also use elitism to retain the best routes from the current population.

Parents

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

9	8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---	---

Offspring

					6	7	8	
--	--	--	--	--	---	---	---	--

9	5	4	3	2	6	7	8	1
---	---	---	---	---	---	---	---	---

Figure 1.25: Numbers are the identifiers of the cities.

Finally, we consider the possibility of mutation. Also in this case we must pay attention that each city doesn't appear twice in the new individuals. So we can't just happily change a city as we described at the beginning of the section. In our case we use a *swap mutation*. This means that, with specified low probability `mutationRate`, two cities will swap places in our route. As usual we type `if random.random() < mutationRate:` to determine the occurrence of a mutation and in case we extract two random numbers indicating the two genes that must be swapped. Next, we can extend the `mutate` function to run through the new population.

At this points we just need to put all these things together to create a function that produces a new generation and run the code. All we need to do is to create the initial population, and then we can loop through as many generations as we desire. For our demonstration, we have created a list of 25 random cities. We set 100 individuals in each generation, kept 20 elite individuals, used a 1% mutation rate for a given gene and run through 500 generations. Of course we also want to see the best route (Figure 1.26) and how much we've improved (Figure 1.27) with new generations. Notice from the latter how the first random population is very far from the best solution, but after having proceeded of just some steps with new generations the length of the route approaches rapidly a short path.

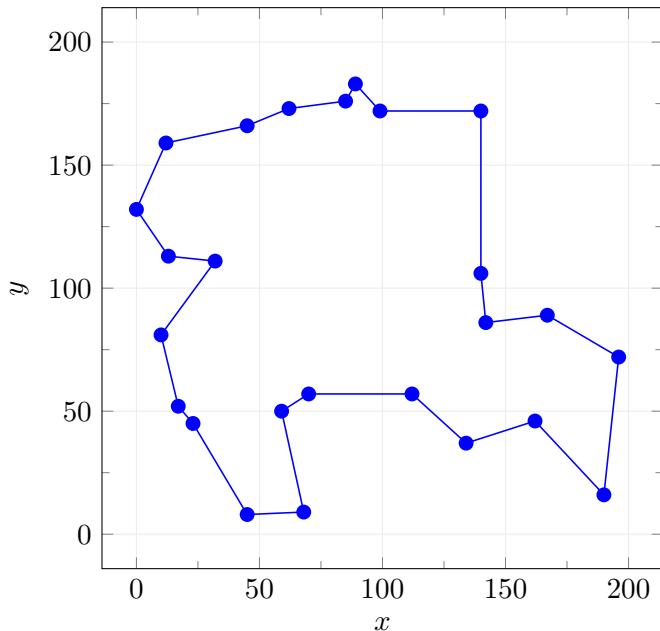


Figure 1.26: Best route found by the genetic algorithm after 500 generations.

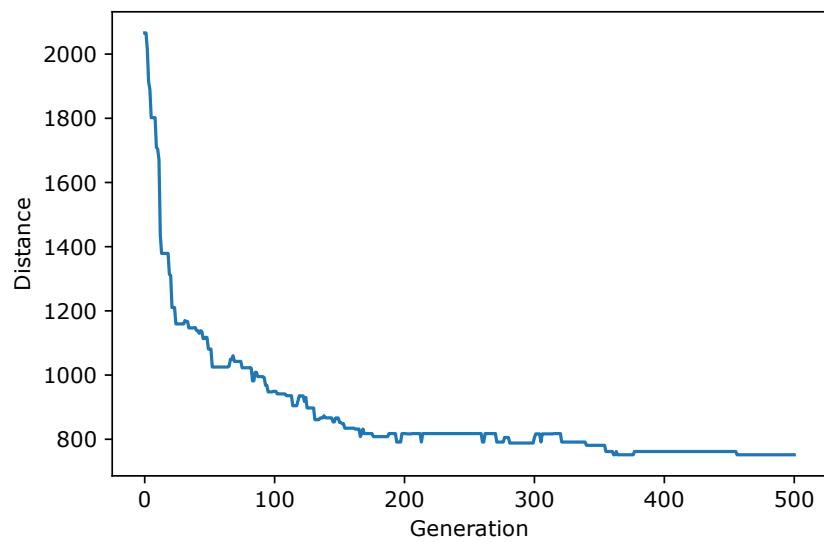


Figure 1.27

CHAPTER 2

Monte Carlo techniques

Contents

2.1	A short review of statistics	38
2.1.1	Sum of many variables	39
2.1.2	Central limit theorem	42
2.2	Pseudo-random number generation	44
2.2.1	Linear congruential generator	44
2.2.2	Combining generators	46
2.2.3	Transformation/inversion method	47
2.2.4	Rejection method	49
2.2.5	Special technique for Gaussian distributions	50
2.3	Metropolis algorithm	52
2.3.1	Simulated annealing	55
2.3.2	Traveling salesman problem	57
2.3.3	Monte Carlo integration	60
2.3.4	Correlations	62

2.1 A short review of statistics

A probability is a measure over a set of events that satisfies three axioms:

- The measure of each event is between 0 and 1. We write this as $0 \leq \Pr(X = x_i) \leq 1$, where X is a random variable representing an event and x_i are the possible values (outcomes) of X .
- The measure of the whole set is 1, that is $\sum_i \Pr(X = x_i) = 1$.
- The probability of a union of disjoint events is the sum of the probabilities of the individual events: $\Pr(X = x_1 \vee X = x_2) = \Pr(X = x_1) + \Pr(X = x_2)$.

This definition works very well for a discrete variable X , i.e. where the list of possible outcomes is finite. For *continuous variables*, there are an infinite number of values, and unless there are point spikes, the probability of any one value is 0 according to the previous definition. Therefore, we define a **probability density function** $f(x)$, but which has a slightly different meaning from the discrete probability function $\Pr(X)$. A random variable X has probability density f_X if

$$\Pr(X \in A) = \int_A f_X(x) dx \quad (2.1.1)$$

where f_X must be a non-negative Lebesgue-integrable function. Accordingly, the second axiom takes the form

$$\Pr(-\infty < X < \infty) = \int_{-\infty}^{\infty} f_X(x) dx = 1 \quad (2.1.2)$$

and we say that $f_X(x)$ is normalized to 1. Note that the probability density function has units, whereas the discrete probability function is unitless.

We can also define a **cumulative distribution function** (CDF) $F_X(x)$, which is the probability of a random variable being less than x :

$$F_X(x) = \Pr(X \leq x) = \int_{-\infty}^x f_X(u) du \quad (2.1.3)$$

For a discrete random variable, the *expected value*, mean or average value, $E(X) \equiv \mu$ is the sum of all possible outcomes weighted by the probability of each value, while for a continuous variable we replace the summation with an integral over the probability density function:

$$\mu = \sum_i x_i \Pr(X = x_i) \quad (2.1.4)$$

$$\mu = \int_{-\infty}^{\infty} x f(x) dx \quad (2.1.5)$$

In a similar fashion the *variance* $\text{Var}(X) \equiv \sigma^2$ is essentially the expected value of the squares of the rejects $\text{Var}(X) = E[(X - \mu)^2]$:

$$\sigma^2 = \sum_i (x_i - \mu)^2 \Pr(X = x_i) \quad (2.1.6)$$

$$\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x) dx \quad (2.1.7)$$

Anyway, from now on we will focus only on continuous variables.

2.1.1 Sum of many variables

Suppose now of having two independent random variables X_1 and X_2 and consider the sum $Y = X_1 + X_2$. What is the probability density function associated with the new variable Y ? What about its mean and variance? In order to answer these questions we need to introduce the concept of marginal probability.

If more than one random variable is defined in a random experiment, it is important to distinguish between the *joint probability distribution* $f_{X_1, X_2}(x_1, x_2)$ of X_1 and X_2 and the probability distribution $f_{X_1}(x_1)$ and $f_{X_2}(x_2)$ of each variable individually. The joint probability distribution is defined in such a way that

$$\Pr((X_1, X_2) \in D) = \iint_D f_{X_1, X_2}(x_1, x_2) dx_1 dx_2 \quad (2.1.8)$$

On the other hand, the individual probability distribution of a random variable is usually referred to as its *marginal probability distribution*. In general, the marginal probability

2 Monte Carlo techniques

distribution of X_i can be determined from the joint probability distribution of X_i and other random variables through the relations

$$f_{X_1}(x_1) = \int_{-\infty}^{\infty} f_{X_1, X_2}(x_1, x_2) dx_2 \quad (2.1.9)$$

$$f_{X_2}(x_2) = \int_{-\infty}^{\infty} f_{X_1, X_2}(x_1, x_2) dx_1 \quad (2.1.10)$$

If X_1 and X_2 are independent¹, then it holds the simple relation

$$f_{X_1, X_2}(x_1, x_2) = f_{X_1}(x_1) \cdot f_{X_2}(x_2) \quad (2.1.11)$$

However, note that $f_{X_1, X_2}(x_1, x_2)$ is *not* equivalent to $f_Y(y)$, i.e. the probability density function of the random variable obtained as $Y = X_1 + X_2$! In fact, it can be shown that $f_Y(y)$ is generally given by

$$f_Y(y) = \int_{-\infty}^{\infty} f_{X_1, X_2}(t, y-t) dt \quad (2.1.12)$$

which becomes the convolution of the individual distributions if and only if the variables are independent (2.1.11):

$$f_Y(y) = (f_{X_1} * f_{X_2})(y) = \int_{-\infty}^{\infty} f_{X_1}(t) f_{X_2}(y-t) dt \quad (2.1.13)$$

Returning to the initial question, we can now write:

$$\begin{aligned} \mu(Y) &= \iint (x_1 + x_2) f_{X_1, X_2}(x_1, x_2) dx_1 dx_2 \\ &= \int_{-\infty}^{\infty} dx_1 x_1 \int_{-\infty}^{\infty} dx_2 f_{X_1, X_2}(x_1, x_2) + \int_{-\infty}^{\infty} dx_2 x_2 \int_{-\infty}^{\infty} dx_1 f_{X_1, X_2}(x_1, x_2) \\ &= \int_{-\infty}^{\infty} x_1 f_{X_1}(x_1) dx_1 + \int_{-\infty}^{\infty} x_2 f_{X_2}(x_2) dx_2 \\ &= \mu(X_1) + \mu(X_2) \end{aligned} \quad (2.1.14)$$

Therefore, as we should expect, the expected value of Y is the sum of the individual expected values of X_1 and X_2 , since the mean is linear.

However, if we repeat the same type of calculation for the variance we obtain a more

¹Two events are said to be independent if $\Pr(AB) = \Pr(A)\Pr(B)$ or alternatively if $\Pr(A|B) = \Pr(A)$. The latter means that the probability to obtain A conditioned by having previously obtained B , is simply given by the probability of obtaining A (B does not influence our output in any way).

complicated expression because of the square inside the integral:

$$\begin{aligned}
 \sigma^2(Y) &= \iint (x_1 + x_2 - \mu(X_1) - \mu(X_2))^2 f_{X_1, X_2}(x_1, x_2) dx_1 dx_2 \\
 &= \int_{-\infty}^{\infty} dx_1 (x_1 - \mu(X_1))^2 \int_{-\infty}^{\infty} dx_2 f_{X_1, X_2}(x_1, x_2) + \\
 &\quad + \int_{-\infty}^{\infty} dx_2 (x_2 - \mu(X_2))^2 \int_{-\infty}^{\infty} dx_1 f_{X_1, X_2}(x_1, x_2) + \\
 &\quad + 2 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 (x_1 - \mu(X_1))(x_2 - \mu(X_2)) f_{X_1, X_2}(x_1, x_2) \\
 &= \int_{-\infty}^{\infty} (x_1 - \mu(X_1))^2 f_{X_1}(x_1) dx_1 + \int_{-\infty}^{\infty} (x_2 - \mu(X_2))^2 f_{X_2}(x_2) dx_2 + \\
 &\quad + 2 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 (x_1 - \mu(X_1))(x_2 - \mu(X_2)) f_{X_1, X_2}(x_1, x_2) \\
 &= \sigma^2(X_1) + \sigma^2(X_2) + 2 \text{Cov}(X_1, X_2)
 \end{aligned} \tag{2.1.15}$$

Notice now that we don't have anymore the same linear relation, but we have a spurious term:

$$\text{Cov}(X_1, X_2) := \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 (x_1 - \mu(X_1))(x_2 - \mu(X_2)) f_{X_1, X_2}(x_1, x_2) \tag{2.1.16}$$

called **covariance**. Essentially, the covariance gives some information about how X_1 and X_2 are statistically related. Hence, if X_1 and X_2 are independent variables, i.e. if it holds (2.1.11), then it is quite simple to convince yourselves that $\text{Cov}(X_1, X_2) = 0$. In that case, and only in that case, the variance of a sum is the sum of the variances:

$$\sigma^2(Y) = \sigma^2(X_1) + \sigma^2(X_2) \tag{2.1.17}$$

Of course, all this can also be extended for the general case of a variable Y obtained as the sum of an arbitrary number N of random variables X_1, X_2, \dots, X_N , with more or less complicated expressions of the covariances. As usual, if the variables are all independent the covariance terms vanish. Moreover, if the variables are also extracted from the same set, i.e. sharing the same probability density function $f_X(x)$, then we get the following immediate relations:

$$Y_N = X_1 + X_2 + \dots + X_N \quad \longrightarrow \quad \sigma_{Y_N}^2 = N\sigma_X^2 \tag{2.1.18}$$

$$Z_N = \frac{1}{N}(X_1 + X_2 + \dots + X_N) \quad \longrightarrow \quad \sigma_{Z_N}^2 = \frac{1}{N}\sigma_X^2 \tag{2.1.19}$$

$$W_N = \frac{1}{\sqrt{N}}(X_1 + X_2 + \dots + X_N) \quad \longrightarrow \quad \sigma_{W_N}^2 = \sigma_X^2 \tag{2.1.20}$$

where σ_X^2 is the common variance of the constituent deviates.

Another important result when is the famous **law of large numbers**. This law states that, if X_1, X_2, \dots, X_N are independent and identically distributed random variables with the same expected value $E(X_1) = E(X_2) = \dots = E(X_N) = \mu$, then the sample average

$$\bar{X}_N = \frac{1}{N}(X_1 + X_2 + \dots + X_N) \tag{2.1.21}$$

converges in probability to μ as $N \rightarrow \infty$.

2 Monte Carlo techniques

The proof makes use of the **Chebyshev inequality**:

$$\boxed{\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}} \quad (2.1.22)$$

where μ and σ are associated to the random variable X . Therefore, let's first demonstrate (2.1.22).

Proof (Chebyshev inequality). The demonstration of the Chebychev inequality starts from the definition of the variance (2.1.7). Exploiting the fact that $(x - \mu)^2$ is always positive and hence surely gives a positive contribution to the integral, we can restrict the integration domain and we get

$$\begin{aligned} \sigma^2 &= \int_{-\infty}^{\infty} (x - \mu)^2 f(x) dx \\ &\geq \int_{-\infty}^{\mu-k\sigma} (x - \mu)^2 f(x) dx + \int_{\mu+k\sigma}^{\infty} (x - \mu)^2 f(x) dx \\ &\geq k^2 \sigma^2 \left[\int_{-\infty}^{\mu-k\sigma} f(x) dx + \int_{\mu+k\sigma}^{\infty} f(x) dx \right] \\ &= k^2 \sigma^2 \Pr(|X - \mu| \geq k\sigma) \end{aligned} \quad (2.1.23)$$

as requested. The last inequality follows from the fact that $x \leq \mu - k\sigma$ for the first integral and $x \geq \mu + k\sigma$ for the second (remember that x is the integration variable), whence $|x - \mu| \geq k\sigma$ and $(x - \mu)^2 \geq k^2\sigma^2$ for both. \square

Proof (law of large numbers). From Chebyshev inequality applied to \bar{X}_N we can write

$$\Pr(|\bar{X}_N - \mu_{\bar{X}_N}| \geq k\sigma_{\bar{X}_N}) \leq \frac{1}{k^2} \quad (2.1.24)$$

Now, since all X_i are independent and identically distributed, from (2.1.19) it follows that $\sigma_{\bar{X}_N} = \sigma/\sqrt{N}$ and $\mu_{\bar{X}_N} = \mu$. Moreover, calling $\varepsilon = k\sigma_{\bar{X}_N}$ we get

$$\Pr(|\bar{X}_N - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{N\varepsilon^2} \quad (2.1.25)$$

In conclusion, as N approaches infinity, the right-hand side approaches 0. And by definition of convergence in probability, we have obtained that $\bar{X}_N \xrightarrow{P} \mu$. \square

2.1.2 Central limit theorem

We conclude this short review of statistics with a fundamental result. Let X_1, X_2, \dots, X_N be a set of variables which are independent and identically distributed according to a Gaussian given by

$$G_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} \quad (2.1.26)$$

Then the random variable $Y_N = X_1 + X_2 + \dots + X_N$ is still distributed according to a Gaussian, but naturally with different parameters:

$$G_{Y_N}(x) = \frac{1}{\sigma\sqrt{2\pi N}} e^{-\frac{x^2}{2N\sigma^2}} \quad (2.1.27)$$

But this is just a simple instance where we restricted to Gaussians. The more general case goes under the name of central limit theorem.

Central limit theorem. Let $\{X_1, X_2, \dots, X_N\}$ be a sample of mutually independent and identically distributed random variables drawn from a common and arbitrary distribution with expected value $E[X] = \mu$ and finite² variance $\text{Var}[X] = \sigma^2$. Let's define the aleatory variable

$$S_N^* := \sqrt{N} \left(\frac{\bar{X}_N - \mu}{\sigma} \right) = \frac{S_N - N\mu}{\sigma\sqrt{N}} \quad (2.1.29)$$

where

$$S_N = X_1 + X_2 + \dots + X_N = \sum_{i=1}^N X_i \quad (2.1.30)$$

is the sample sum and

$$\bar{X}_N = \frac{1}{N}(X_1 + X_2 + \dots + X_N) = \frac{1}{N} \sum_{i=1}^N X_i \quad (2.1.31)$$

is the sample mean. Then, as $N \rightarrow \infty$, the random variable S_N^* converges in distribution to a Gaussian having zero expected value and unit variance: $S_N^* \xrightarrow{d} \mathcal{N}(0, 1)$ ³.

We remember that convergence in distribution means that the cumulative distribution function of S_N^* converges pointwise to the cumulative distribution function of a normal distribution with expected value 0 and variance 1. In mathematical terms this is equivalent to say that

$$\lim_{N \rightarrow \infty} \Pr(S_N^* \leq x) = F(x) \quad (2.1.32)$$

$\forall x \in \mathbb{R}$, where

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt \quad (2.1.33)$$

So, there is the—quite surprising!—fact that the probability density function from which the variable S_N^* is sampled, that is the single random variable essentially obtained as the standardized average of many independent random variables, but characterized by the *same* and *arbitrary* distribution law, approaches the probability density function of the Gauss distribution

$$\mathcal{N}(x; 0, 1) \equiv \phi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (2.1.34)$$

whatever the law of the starting variables X_N (not necessarily normally distributed). Largely speaking, the central limit theorem establishes that, in many situations, when independent random variables are added, their properly normalized sum tends toward a normal distribution (informally a bell curve) even if the original variables themselves are not normally distributed.

Notice also the difference with respect to the law of large numbers. By the law of large numbers, the sample averages converge almost surely (and therefore also converge

²For instance, let's consider a Cauchy distribution (also known as Lorentz distribution, especially among physicists):

$$f(x; x_0, \gamma) = \frac{1}{\pi\gamma \left[1 + \left(\frac{x-x_0}{\gamma} \right)^2 \right]} \quad (2.1.28)$$

where x_0 is the location parameter, specifying the location of the peak of the distribution, and γ is the scale parameter, which specifies the half-width at half-maximum (HWHM). Since the Cauchy distribution has neither a mean nor a variance, being both undefined, the central limit theorem does not apply.

³Equivalently, we can also state that $\sqrt{N}(\bar{X}_N - \mu) \rightarrow \mathcal{N}(0, \sigma^2)$.

in probability) to the expected value μ as $N \rightarrow \infty$. Instead, the classical central limit theorem describes the size and the distributional form of the stochastic fluctuations around the deterministic number μ during this convergence.

In [→](#) we provided an example in Mathematica, where the distribution of average of a certain number of variables converges to a Gaussian. Also, you can take a look at [→](#).

2.2 Pseudo-random number generation

With random numbers we refer to a sequence of numbers in random or uncorrelated order, with each number having the same probability to occur next in the sequence. The algorithmic creation of random numbers is problematic since computers are deterministic, while the sequence should be non-deterministic. This means that once the characteristic of the generator and its initial values are known, one is completely able to predict the entire sequence (they are not truly random). Therefore, in real life one considers algorithms for the creation of *pseudo-random numbers*, which, although calculated in a deterministic way, should be *almost* homogeneously randomly distributed (to the best of possibilities). These numbers should follow a well-defined distribution and should have long periods. Furthermore, they should be calculated in a efficient and reproducible way.

2.2.1 Linear congruential generator

Linear congruential generators (LCG) are a class of pseudo-random number generator (RNG) algorithms used to produce sequences of random-like numbers within a specific range. The mathematical theory behind linear congruential generators is relatively easy to understand, which makes them easily implementable and fast. And indeed, it is not surprising that the linear congruential method is the most popular algorithm for random number generation in the field of computer simulations. This method can be defined by the following recurrence relation:

$$I_{n+1} = (aI_n + b) \bmod m \quad (2.2.1)$$

($a, m \in \mathbb{N} \setminus \{0\}$ and $b \in \mathbb{N}$) which returns a pseudo-random sequence of integer numbers between 0 and $m - 1$ starting from I_0 . In (2.2.1) a is called the *multiplier*, b is the *increment*, m is the *modulus*, I_0 is the *seed* which initializes the generator and I_n is the complete sequence of pseudo-random values obtained. Obviously, the sequence is perfectly deterministic since the value I_{n+1} is uniquely defined by the previous one I_n .

The operation $\bmod m$ returns the remainder of the integer division with respect to m . Accordingly, at each iteration the generator will always return numbers between 0 and $m - 1$. For this reason, it is reasonable to expect m to be closely related to the period of the generator. Remember that the period of a sequence of numbers is defined as the length of the smallest subgroup of the original sequence after which the numbers repeat identically, both in values and in order. So, when a LCG outputs a number that has already been generated, it means that the period of the generator has been reached, since from that moment on the whole sequence will be repeated the same as itself. In particular, m constitutes the *maximum obtainable period*, because at best the sequence repeats itself after having produced all the numbers between 0 and $m - 1$.

If we freely associate to a and b random numbers, the possibility that all the values between 0 and $m - 1$ are generated (thus obtaining the period with maximal length) will most likely be unverified, i.e. the period of the generator obviously depends on m but also on the choice of the other two parameters in equation (2.2.1), a and b (any initial

“seed” choice of I_0 is as good as any other: the sequence just takes off from that point). For example, adopting $a = 5$, $b = 1$ and $m = 100$ and starting with $I_0 = 1$ we produce the sequence 1, 6, 31, 56, 81, 6 and then it will repeat again because each number depends exclusively on the previous one. Hence, in this case we are not making good use of the stock of numbers that we could potentially extract. In conclusion, if a and b are chosen “correctly⁴”, the period will be of maximal length (i.e. m). Only in that case all possible integers between 0 and $m - 1$ appear at some point in the sequence, and this is desirable in order to increase the extrinsic quality of the generator. In fact, if we want to produce a certain sample of N numbers with a semblance of randomness, at the very least we should *avoid very short cycles*, for which the period is smaller than N ; choosing the suitable combination of parameters in order to obtain the maximum period is certainly a first step towards the solution.

So, the first problem with a linear congruential generator is the risk of producing short cycles, where “short” is to be referred with respect to the sample of N random numbers we want to generate for our goals. Apart from this, even if we have achieved a high period, we must face the fact that not all random number generators having long periods have the same quality. Not all sources of random numbers behave in the same way, and some are better than others, at least for different applications. This begs the question: how can we tell if a random number generator is “good” (or “good enough”)? We’ve already seen that there are ways to maximize the period of a LCG. Certainly, this is not the only requirement we would demand of a RNG. Not surprisingly, there are a lot of different quality *tests* for RNG’s and the sequences they produce. These tests can be divided into two distinct groups: empirical tests and theoretical tests. Empirical tests are conducted on a sequence generated by a RNG, and require no knowledge of how the RNG produces the sequence. Theoretical tests, which are better when they exist, are *a priori* tests, in the sense that they require a knowledge of the structure of the RNG but the sequence does not necessarily need to be generated. We will focus mainly on the empirical tests here⁵.

One of the most powerful empirical test known is the so-called **spectral test** and it deals with LCGs. It is widely considered by far the most powerful test known, because it can fail LCGs which pass most statistical tests. The test works in the following way. Let suppose of having generated our sequence of random numbers r_1, r_2, \dots, r_N from a specific LCG. Then we build pairs of consecutive numbers of the string $(r_1, r_2), (r_2, r_3), (r_3, r_4), \dots$ and we regard them as points in a 2-dimensional space, as shown in Figure 2.1a. As we can notice, the points will not tend to “fill up” the whole space, but rather an order pattern is observed; actually we say that they organize to form a family of parallel lines, and this is a symptom of the inaccuracy of the generator. The spectral test essentially compares the distance between these lines; the further apart they are, the worse the generator is. This test can also be generalized to higher dimensions by introducing the d -tuples $(r_i, r_{i+1}, r_{i+2}, \dots, r_{i+d-1})$ if we are not able to distinguish any ordered structure to the lowest level, since there are LCGs that fail the test in smaller dimensions⁶ (Fig.2.1b). But in any case, LCGs have a property that when plotted in 2 or more dimensions, lines or hyperplanes will *always* form, on which all possible outputs can be found. The spectral test compares the distance between these planes and the closer they are, the better the generator is. Note that with the spectral test we are able to examine the entire sequence without generating a full period,

⁴It is possible to demonstrate that the maximum period m is obtained if and only if: b and m have no common divisors (coprimes); $a - 1$ must be multiple of p , for each prime divisor p of m ; if m is multiple of 4, then also $a - 1$ must be so.

⁵Theoretical tests cannot be used for every RNG (unlike the empirical tests) and they are usually hard to find, and they tend to be considerably more complicated than empirical.

⁶The IBM subroutine `RANDU` LCG fails in this test for 3 dimensions and above.

2 Monte Carlo techniques

which allows us to search for global non-randomness, in addition to local non-randomness by means of empirical tests.

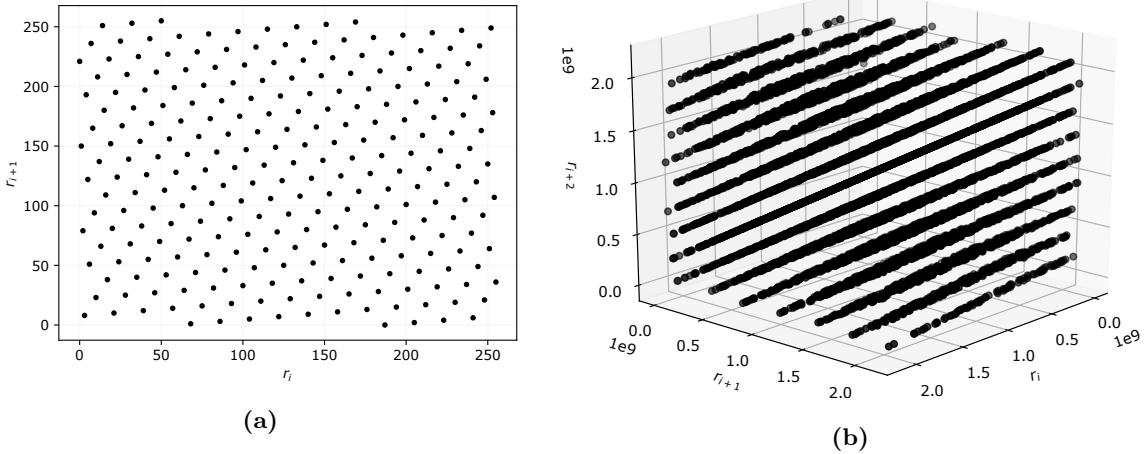


Figure 2.1: Examples of 2D (a) and 3D (b) spectral methods: (a) is set with $m = 256$, $a = 137$, $b = 187$ (these parameters give full period), while (b) has $m = 2^{31}$, $a = 65539$, $b = 0$ (this congruential generator is the (in)famous RANDU).

2.2.2 Combining generators

We analyze now another problem related to congruential generators. Let's consider the simple multiplicative congruential generator

$$I_{n+1} = aI_n \bmod m \quad (2.2.2)$$

i.e. with $b = 0$. Park and Miller proposed a “Minimal Standard” generator based on the choices

$$a = 7^5 = 16807, \quad m = 2^{31} - 1 = 2147483647 \quad (2.2.3)$$

This generator (called `ran0` in [4]) surely has the maximum period, because m is prime⁷, and indeed it is satisfactory for the majority of applications, but we do not recommend it as the final word on random number generators. Our reason is precisely the simplicity of the Minimal Standard. It is not hard to think of situations where successive random numbers might be used in a way that accidentally conflicts with the generation algorithm. For example, assume we extracted a very small number I_n compared to m ; then the next draw would be also rather small because $aI_n \gtrsim 10^4$ and we modulo-divide it by something of the order of 10^9 . And the same thing affects systematically the subsequent draws. In other words, since successive numbers differ by a multiple of only 10^4 out of a modulus of more than 10^9 very small random numbers will tend to be followed by smaller than average values. One time in 10^6 , for example, there will be a value $< 10^6$ returned (as there should be), but this will *always* be followed by a value less than about 0.0168. One can easily think of applications involving rare events where this property would lead to wrong results.

Usually, one way to create a more efficient generation algorithm is to combine two or more different generators in a purposeful way. In the specific case of `ran0`, one way to eliminate these correlations is to consider `ran1`, which uses the Minimal Standard for its random value, but it shuffles the output to remove low-order serial correlations. In practice,

⁷ $2^{31} - 1$ is the maximum value that a 32-bit (signed) integer can take.

once obtained a number I_n , we generate the next 10 (e.g.) according to (2.2.2) but we don't insert them directly in the final sequence. Instead, we put them in a buffer and we pick one of them randomly. This number selected randomly from the buffer will then be our I_{n+1} of the sequence. From I_{n+1} we generates other 10 numbers, we put them in a blank buffer, we choose one randomly, which will constitutes I_{n+2} , and so on completing the sequence. Clearly, the probability to obtain another very low number is extremely reduced.

2.2.3 Transformation/inversion method

In the previous section we have learned how to generate random numbers with a uniform probability between 0 and 1 (actually between 0 and $m - 1$, but we can normalize the interval by dividing by $m - 1$ the output), denoted $U(0, 1)$. Certainly LCGs produce a discrete distribution of numbers, but within the limit of very large periods⁸ we can imagine that it is substantially continuous. From the theoretical point of view, such continuous *uniform* probability of generating a number between x and $x + dx$ is defined as

$$p(x) dx = \begin{cases} dx, & \text{for } 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{cases} \quad (2.2.4)$$

where $p(x)$ is the probability density function.

Now we will learn how to generate random numbers drawn from other probability distributions. Let's first fix the following result. Suppose that we generate a random number x from a known distribution $p_x(x)$ and then take some prescribed function of it, $y(x)$. What is the probability density function describing the possible outcomes of $y(x)$? In general, the probability distribution of y , denoted $p_y(y) dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p_y(y) dy| = |p_x(x) dx| \quad (2.2.5)$$

or

$$p_y(y) = \left| \frac{dx}{dy} \right| p_x(x) \quad (2.2.6)$$

This follows from the fact that the probability contained in a differential area must be invariant under change of variables.

At the moment we only know how to extract random numbers from a uniform probability distribution $U(0, 1)$. In this particular case, the equation takes a very simple form:

$$p_y(y) = \left| \frac{dx}{dy} \right| \quad (2.2.7)$$

because $p_x(x) = 1$. Just to make an example, if $y(x) = -\ln x$ and x comes from a uniform distribution, then the variable $y = y(x)$ becomes distributed according

$$p(y) dy = \left| \frac{d}{dy} (e^{-y}) \right| dy = e^{-y} dy \quad (2.2.8)$$

because $x = e^{-y}$.

⁸There exist nice tricks in order to avoid problems of exceeding memory, arising from multiplication of very large numbers.

2 Monte Carlo techniques

We have seen that this procedure starts with a set of random variables x and through the relation $y = y(x)$, given at the beginning, we are able to generate random numbers distributed according to (2.2.6). Is then possible to reverse the reasoning? That is, how to determine the relation $y = y(x)$, i.e. the mapping between different variables, so that $p_y(y)$ is a given normalized $f(y)$? If x comes from $U(0, 1)$, then we just need to solve the differential equation (2.2.7):

$$f(y) = \left| \frac{dx}{dy} \right| \quad (2.2.9)$$

for y . But the solution of this is naturally $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$:

$$F(y) = \int_{-\infty}^y f(y') dy' \quad (2.2.10)$$

i.e. the cumulative distribution function. The desired transformation that takes a uniform deviate into one distributed as $f(y)$ is therefore

$$y = F^{-1}(x) \quad (2.2.11)$$

To summarize, the essence of the **transformation method** is the following. We first compute the cumulative distribution function F of the distribution we want to sample and we invert it; then we extract an x from a uniform distribution $U(0, 1)$ and we compute $y = F^{-1}(x)$. The resulting y will be surely distributed according to $f(y)$.

The transformation method has an immediate geometric interpretation (Fig. 2.2). The integral of $f(y)$ is the area under the probability curve; hence $F(y)$ indicates how the size of this area changes if we keep still one of the extremes of integration and we move forward the other, thus widening the domain of integration. Naturally, since $F(y)$ is a cumulative distribution function (remember that $F_Y(y) = \Pr(Y \leq y)$), this area ranges from 0 (zero range of integration) to 1 (all the real axis). In conclusion, $y = F^{-1}(x)$ is just the prescription: choose a uniform random x on the ordinate, then find the corresponding y on the abscissas that has that fraction x of probability area to its left.

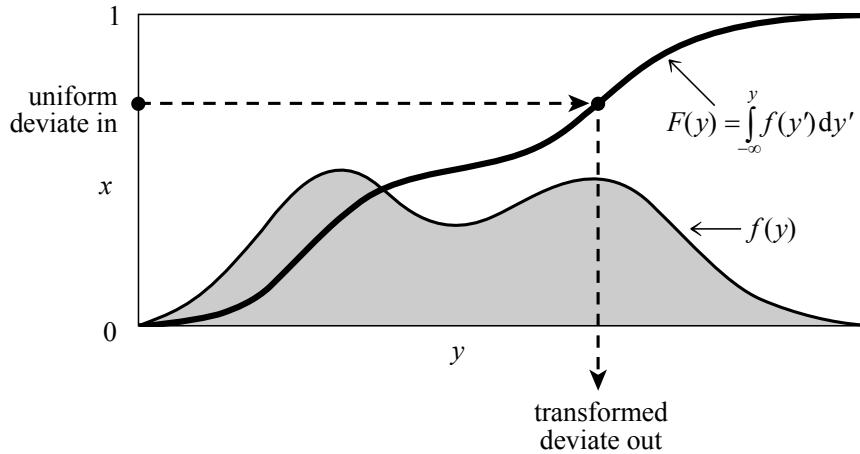


Figure 2.2

Notice also that $F(y)$ is the integral of $f(y)$. This means that at the values of y for which $f(y)$ is large, the function $F(y)$ is steep, and vice versa where $f(y)$ is small then $F(y)$ is flatter; in fact, the higher parts of the function $f(y)$ contribute more to the growth of the integral $F(y)$. By consequence, it follows that by mapping a *wide* range of values of x (vertically) through a *steep* function F^{-1} (i.e. $F(y)$), we obtain along the abscissa

axis (horizontally) a *narrow* interval of y . And this makes sense because it means that a large part of the uniform x extractions will produce very localized values of y distributed according to $f(y)$, in accordance with the fact that this probability density function is more piqued in that neighborhood. A similar reasoning also applies to the opposite case.

This technique is rather general and powerful but has a crucial problem: we must compute the inverse of $F(y)$ and this is not necessarily always possible.

2.2.4 Rejection method

The **rejection method** is a powerful, general technique for generating random numbers according to whatever distribution $p(x)$. The rejection method does not require that the cumulative distribution function (integral of $p(x)$) be readily computable, much less the inverse of that function—which was required for the transformation method in the previous section.

This method works as follows. First of all, from the knowledge of $p(x)$ we determine another probability density function $f(x)$ such that, once multiplied by a constant A , it always lies above $p(x)$ along the whole domain of definition, that is $Af(x) \geq p(x)$. Note that the constant is essential to ensure the inequality, because both $p(x)$ and $f(x)$ are probability density functions, and hence they have unitary areas. This majorant function, called the *comparison function*, must also be chosen in such a way that it could be used as a source of random numbers, that is we must be able (in any way) to generate random numbers distributed according to $f(x)$. For instance, if we want to extract numbers using the transformation method, then $f(x)$ must be integrable and its cumulative distribution function must be invertible. Once found the function, we generate random numbers x_i according to $f(x)$. Then we generates random numbers y_i uniformly distributed in the interval $0 \leq y_i \leq Af(x_i), \forall i$. If $y_i \leq p(x_i)$ then we accept x_i as outcome, otherwise we discard it. At the end, the sequence of x_i that passed the test constitutes our sample of random numbers distributed according to $p(x)$.

The rejection method is based on a simple geometrical argument (Fig. 2.3). Whenever that point (x_i, y_i) lies outside the area under the original probability distribution $p(x)$, we will reject it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will accept it. That's why this method is sometimes called *hit and miss*. It should also be obvious that the fraction of points rejected, and therefore the efficiency of our algorithm, just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. Therefore it would be advisable to choose an upper bound function that is as similar as possible to the starting distribution.

In we implemented an example of this procedure in the case of a $p(x)$ given by the sum of two Gaussians which is majored by another Gaussian $q(x)$. There we estimated the constant A (called k in the example) in a rough, and not very precise, way by computing the ratio $p(x)/q(x)$ for a discrete set of values and taking the maximum.

We have also given an example of **kernel density estimation** (through the `seaborn`'s function `distplot`), which is way to estimate the probability density function of a random variable if that is unknown. Kernel density estimation is a fundamental data smoothing problem where inferences about the population are made, based on a finite data sample. The method works as follows. Let $\{x_1, x_2, \dots, x_N\}$ be independent and identically distributed samples drawn from some univariate distribution with an unknown density $p(x)$. Its kernel

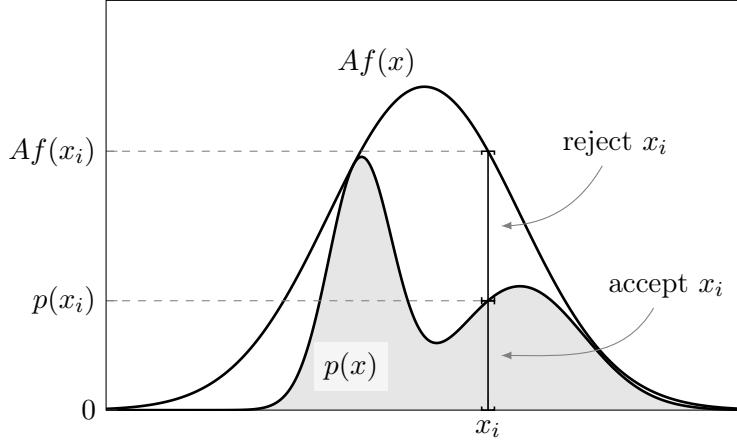


Figure 2.3

density estimator is defined as

$$\hat{f}_h(x) := \frac{1}{Nh} \sum_{i=1}^N K\left(\frac{x - x_i}{h}\right) \quad (2.2.12)$$

where K is the kernel and $h > 0$ is a smoothing parameter called the bandwidth. A range of kernel functions are commonly used—uniform, triangular, normal and others—but they all differ in efficiency. Due to its convenient mathematical properties, the normal kernel is often used, which means $K(x) = \mathcal{N}(x; 0, 1)$, where $\mathcal{N}(x; 0, 1) \equiv \phi(x)$ is the standard normal density function. Kernel density estimates are closely related to histograms, but can be endowed with properties such as smoothness or continuity by using a suitable kernel (Figure 2.4). For the histogram (left panel), first the horizontal axis is divided into sub-intervals or bins which cover the range of the data, and when one data point falls inside the same bin we increase the count of that bin; here the degree of freedom is the width of the bins. For the kernel density estimate (right panel), normal kernels, indicated by the red dashed lines, are placed on each of the data points x_i . Then the kernels are summed to make the kernel density estimate (solid blue curve). The smoothness of the kernel density estimate (compared to the discreteness of the histogram) illustrates how kernel density estimates converge faster to the true underlying density for continuous random variables. Intuitively one wants to choose h as small as the data will allow; however, there is always a trade-off between the bias of the estimator and its variance. In fact, if h is too small the curve is undersmoothed, while if h is too large the curve is oversmoothed, with respect to real distribution $p(x)$. An extreme situation is encountered in the limit $h \rightarrow 0$ (no smoothing), where the estimate is a sum of N delta functions centered at the coordinates of analyzed samples.

2.2.5 Special technique for Gaussian distributions

The previous methods are general and can, when possible, be applied to whatever distribution function. Instead, for Gaussians specific procedures have also been developed. One of these is based on classical physics considerations. Let's imagine of having a system of identical and independent particles; these particles are essentially free to move in the space at disposal, but we let them interact with each other only through 2-body elastic collisions, where total energy and momentum are notoriously conserved. At thermodynamic equilibrium, the

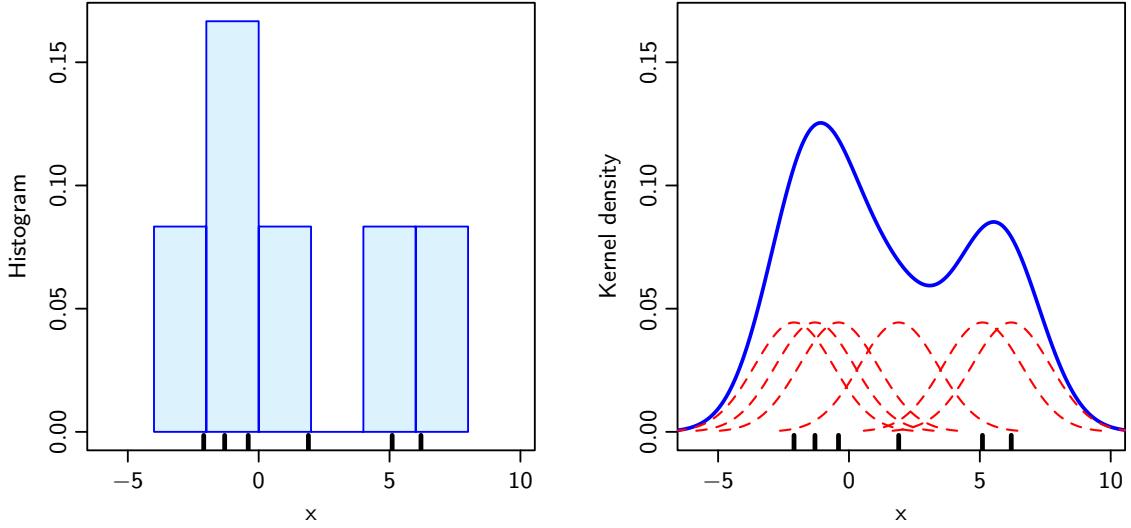


Figure 2.4: Comparison between histogram and kernel density estimation using the sample $\{-2.1, -1.3, -0.4, 1.9, 5.1, 6.2\}$. The red dashed curves represent normal kernels (actually, they are $\frac{1}{Nh} \mathcal{N}(x; x_i, h^2)$) with $h \simeq 1.5$. The data points are the rug plot on the horizontal axis.

entire system has the same temperature and the distribution of velocities (or momenta) is a Maxwell–Boltzmann distribution:

$$f(\mathbf{v}) \propto e^{-\frac{E}{k_B T}} = e^{-\frac{mv^2}{2k_B T}} \quad (2.2.13)$$

Here $f(\mathbf{v}) d\mathbf{v}$ essentially refers to the number of particles with velocity between \mathbf{v} and $\mathbf{v} + d\mathbf{v}$ in a phase space volume.

Our algorithm for random number generation starts with an initial situation in which our N particles are still not thermalized, i.e. where their distribution cannot yet be described in term of a Maxwell–Boltzmann distribution. The initial velocities can be chosen at random through whatever technique: for instance, we can extract random numbers from a uniform distribution between 0 and 1 and associate these values of velocity to our particles in this initial condition. Then, we spontaneously let the system thermalize through random elastic collisions. Practically, we choose two particles i and j and we replace their initial velocities with new values obtained through the following transformations:

$$\begin{cases} v_i(\text{old}) \rightarrow v_i(\text{new}) = \frac{v_i(\text{old}) + v_j(\text{old})}{\sqrt{2}} \\ v_j(\text{old}) \rightarrow v_j(\text{new}) = \frac{v_i(\text{old}) - v_j(\text{old})}{\sqrt{2}} \end{cases} \quad (2.2.14)$$

Note that these transformations leave unchanged the total energy of the couple ($v_i^2(\text{new}) + v_j^2(\text{new}) = v_i^2(\text{old}) + v_j^2(\text{old})$), but naturally the individual velocities will be modified. This is basically the essence of a 2-body elastic scattering, where the two particles interact and change their motion till they reach a thermal equilibrium. This procedure (i.e. choosing randomly a couple and applying (2.2.14)) has to be repeated a large number of times, meaning many more collisions. If theoretically we let the process evolve indefinitely, at an asymptotic time the system would reach complete thermalization, that is the particles will end up following the Maxwell–Boltzmann distribution (2.2.13). Since the MB distribution is a Gaussian, this means that our final sample of N particles having velocities v_k will be derived from a normal distribution. In other words, at the very end of the process

2 Monte Carlo techniques

we obtain a sequence of N numbers drawn from a normal distribution. However, it is possible to demonstrate that only after $2\text{--}3N$ rearrangements the system is essentially fully thermalized. See the little implementation [→](#), for instance.

2.3 Metropolis algorithm

In the previous sections we analyzed some of the random number generation methodologies, but they all have a weakness. The LCGs are restricted to uniform distributions and the transformation method can only be used as long as the CDF can be inverted. Even the rejection method, which certainly constitutes a step forward compared to the others, is not always “perfect” as it may seem, because the search for a majorant function is not always so immediate. For this reason, the famous *Monte Carlo techniques* have been introduced over time. Of these, we will focus on the **Metropolis–Hastings algorithm**, which is Markov chain Monte Carlo⁹ (MCMC) method for obtaining a sequence of random samples from a probability distribution from which direct sampling is usually difficult. This technique and other MCMC algorithms are indeed generally used for sampling from multi-dimensional distributions, especially when the number of dimensions is high.

The algorithm works as follows. Let $f(x)$ be a function that is proportional to the desired probability distribution $P(x)$, i.e. the target distribution.

1. Initialization: Choose an arbitrary point x_0 to be the first sample, and choose an *arbitrary* probability density $g(x|y)$ that suggests a candidate for the next sample value x , given the previous sample value y . In other words, $g(x|y)$ tells us the probability that, starting from y , x is selected as the new value to be taken into consideration. For the Metropolis algorithm g must be *symmetric*, that is it must satisfy $g(x|y) = g(y|x)$ (in this way we assure the possibility to explore all possible configurations). A usual choice is to let $g(x|y)$ be a Gaussian distribution centered at y , so that points closer to y are more likely to be visited next—making the sequence of samples into a random walk. The function g is referred to as the *proposal probability*.
2. For each iteration t ($t = 0, 1, 2, \dots$ is sometimes called “Markov time”):
 - Generate a candidate x' for the next sample by picking from the distribution $g(x'|x_t)$.
 - Calculate the *acceptance ratio* $\alpha = f(x')/f(x_t)$, which will be used to decide whether to accept or reject the candidate. Because f is proportional to the density of P , we have that $\alpha = f(x')/f(x_t) = P(x')/P(x_t)$.
 - If $\alpha \geq 1$, then the candidate is more likely than x_t ; automatically we accept it by setting $x_{t+1} = x'$. Otherwise, if $\alpha < 1$ we accept the candidate with probability α . To do this, we generate a uniform number $u \in [0, 1]$:
 - If $u \leq \alpha$, then we accept the candidate and set $x_{t+1} = x'$.
 - If $u > \alpha$, then we reject the candidate and set $x_{t+1} = x_t$, instead.

The procedure could seem so cryptic. Intuitively, we can imagine the following easier steps. Let x_t be the initial state and Δ the typical scale of variation of the probability distribution we want to sample. We are essentially free to choose Δ : the larger it is the better the final result, but it must be finite (besides, asymptotically the probability distribution goes to 0).

⁹Markov chain Monte Carlo methods comprise a class of algorithms for sampling from a probability distribution and based on Markov chains.

Then, we extract a random uniform number $r \in [-1, 1]$ and consider the value $x' = x_t + r\Delta$ as possible new candidate. Note that the previous recipe told us to generate the candidate through a symmetric proposal probability g , and here that's exactly what we're doing because the quantity $r\Delta$ is essentially a random variable drawn from a uniform distribution which is symmetric around 0, as required. The next step is the usual test for its acceptance or rejection.

The Metropolis–Hastings algorithm works by generating a sequence of sample values in such a way that, as more and more sample values are produced, the distribution of values more closely approximates the desired distribution $P(x)$. These sample values are produced iteratively, with the distribution of the next sample being dependent only on the current sample value (that's the meaning of $g(x|y)$), thus making the sequence of samples into a Markov chain. Specifically, at each iteration, the algorithm picks a candidate for the next sample value based on the current sample value. Then, with some probability, the candidate is either accepted (in which case the candidate value is used in the next iteration) or rejected (in which case the candidate value is discarded, and current value is reused in the next iteration)—the probability of acceptance is determined by comparing the values of the function $f(x)$ of the current and candidate sample values with respect to the desired distribution $P(x)$. Thus, we will tend to stay in (and return large numbers of samples from) high-density regions of $P(x)$, while only occasionally visiting low-density regions. Intuitively, this is why this algorithm works and returns samples that follow the desired distribution $P(x)$.

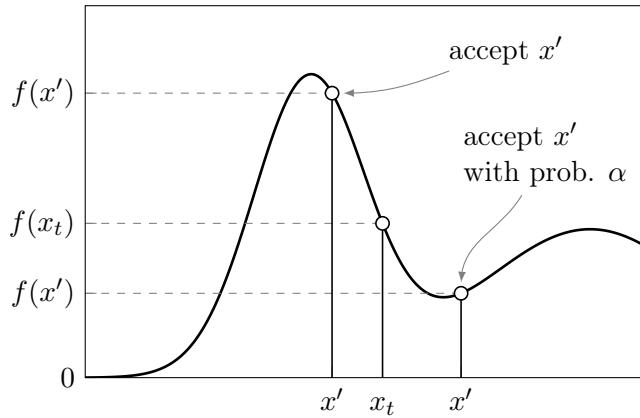


Figure 2.5

The algorithm we have just described starts from a single element x_0 and at each iteration it produces one at a time a new value to add to the sample, based on the previous value. However, there is also a different way to interpret it. In this “second way”, instead of starting with just one seed, we begin with an entire sequence of N values already extracted, but not necessarily from the desired distribution (it's just an arbitrary random sample). Then, for one of them chosen randomly we propose a new candidate which eventually replaces the initial value. Also in this case, the choice of the new candidate is made through g and the choice of acceptance or rejection is made according to the procedures already described above. This rearrangement of the initial set has to be done a large number of times, and at the very end we expect to reproduce the desired density function.

2 Monte Carlo techniques

Formal derivation

The purpose of the Metropolis–Hastings algorithm is to generate a collection of states according to a desired distribution $P(x)$. To accomplish this, the algorithm uses a *Markov process*, which asymptotically reaches a unique stationary distribution $\pi(x)$ such that $\pi(x) = P(x)$. A Markov process is uniquely defined by its transition probabilities $p_{i \rightarrow j} \equiv p(x_j|x_i)$, the probability of transitioning from any given state x_i to any other given state x_j . It has a unique stationary distribution $\pi(x)$ when the following two conditions are met:

- Existence of stationary distribution $\pi(x)$. A sufficient but not necessary condition is *detailed balance*, which requires that each transition $x_i \rightarrow x_j$ is reversible. This means that for every pair of states the probability of being in state x_i and transitioning to state x_j must be equal to the probability of being in state x_j and transitioning to state x_i :

$$P(x_i)p_{i \rightarrow j} = P(x_j)p_{j \rightarrow i} \quad (2.3.1)$$

- Uniqueness of stationary distribution, which is guaranteed by the ergodicity of the Markov chain.

The Metropolis–Hastings algorithm involves designing a Markov process (by constructing transition probabilities) that fulfills the two above conditions, such that its stationary distribution $\pi(x)$ is chosen to be $P(x)$. The formal derivation of the algorithm starts with the condition of detailed balance (2.3.1), which is re-written as

$$\frac{P(x_i)}{P(x_j)} = \frac{p_{j \rightarrow i}}{p_{i \rightarrow j}} \quad (2.3.2)$$

Now, as we mentioned at the beginning, we separate the transition in two sub-steps, the proposal and the acceptance–rejection, that is

$$p_{i \rightarrow j} = g_{i \rightarrow j} A_{i \rightarrow j} \quad (2.3.3)$$

where $A_{i \rightarrow j}$ is the *acceptance probability*. Accordingly (2.3.2) becomes

$$\frac{A_{j \rightarrow i}}{A_{i \rightarrow j}} = \frac{P(x_i)}{P(x_j)} \frac{g_{i \rightarrow j}}{g_{j \rightarrow i}} \quad (2.3.4)$$

The crucial point is that g is chosen to be symmetric; therefore the following ratios are all equal:

$$\frac{A_{j \rightarrow i}}{A_{i \rightarrow j}} = \frac{P(x_i)}{P(x_j)} = \frac{p_{j \rightarrow i}}{p_{i \rightarrow j}} \quad (2.3.5)$$

The next step in the derivation is to choose an acceptance probability that fulfills the condition above. The common choice in the Metropolis algorithm is to set

$$A_{i \rightarrow j} = \min \left(1, \frac{f_j}{f_i} \right) \quad (2.3.6)$$

(verify that this works) where $f_i \equiv f(x_i)$ and $f(x)$ is any function proportional to $P(x)$. So, in conclusion Metropolis algorithm consists in applying iteratively (2.3.3) and (2.3.6) to a given state in order to explore all the distribution, and this is exactly the procedure that we prescribed at the beginning of this section. In fact, if $f_j/f_i > 1$, then $A_{i \rightarrow j} = 1$ and we always accept the new candidate, whereas if $f_i/f_j < 1$ we accept it with probability $A_{i \rightarrow j} = f_j/f_i$.

The principle of detailed balance whence we started is essentially the result of an *equilibrium* which is reached. In fact, imagine to have different bins within which we organize an ensemble of particles; in particular, let n_i be the number of particles in the i -th bin. If we let the system evolve, at a later time we will find that the number of particles in the bin i is changed by a quantity

$$\Delta n_i = n_j p_{j \rightarrow i} - n_i p_{i \rightarrow j} = n_j p_{i \rightarrow j} \left(\frac{p_{j \rightarrow i}}{p_{i \rightarrow j}} - \frac{n_i}{n_j} \right) \quad (2.3.7)$$

Here $n_j p_{j \rightarrow i}$ is basically the incoming flux (number of particles in the j -th bin times the probability to go from j to i), which tends to increase the number, while $n_i p_{i \rightarrow j}$ represents the outgoing flux (number of particles in the i -th bin times the probability to migrate from i to j), which tends to decrease the number. Now, being all involved quantities positive definite, if $p_{j \rightarrow i}/p_{i \rightarrow j} > n_i/n_j$, then $\Delta n_i > 0$, whereas if $p_{j \rightarrow i}/p_{i \rightarrow j} < n_i/n_j$, then $\Delta n_i < 0$. In both cases, this means that the system evolves reducing or increasing the number of particles in the i -th bin. Asymptotically we expect the system to reach a stable configuration in which the number of particles coming in and out from the same bin is equal, and this corresponds to the realization of an equilibrium configuration. In that case we indeed have $\Delta n_i = 0$, that is

$$n_j p_{j \rightarrow i} = n_i p_{i \rightarrow j} \quad (2.3.8)$$

which is the principle of detailed balance.

In the attachments we also provided two practical examples showing how this technique works. The first example `π` is just a numerical evaluation of π , intended as 4 times the ratio between the area of a circle and a square circumscribed to it. For that purpose we generate randomly points (x_i, y_i) in two dimensions: if (x_i, y_i) falls within the circle, then we accept it and we increase the count of hits; if (x_i, y_i) falls outside, then we discard it. At the end, $\pi = 4R$, where R is the ratio of the points accepted to the total number of points analyzed.

The second example `Maxwell` consists in picking particles from the Maxwell distribution of velocities and exploits the second interpretation of the Metropolis algorithm. We start with a sequence `u` of `N` particles all with the same velocity `u0`. Then we choose randomly one of them (e.g. `n`, with velocity `ov=u[n]`), we extract a uniform number `du` between `-dv` and `dv` (which is equivalent to $r\Delta$) and we consider the new candidate with velocity `u[n]+du`. Then we compute the new Hamiltonian `h_new` with this possible change and we use it to test whether to accept the candidate or not. Notice that Maxwell distribution is just something of the form $\exp(-\frac{E}{k_B T})$ (or a Gaussian in terms of velocities); therefore, in the ratio $f(x_j)/f(x_i)$ we are essentially interested in the difference in energy between the old and the new configuration: `dh=0.5*m*(u[n]**2-ov**2)`. If `dh` is negative, then we accept the candidate. If `dh` is positive, then we accept it with probability `math.exp(-dh/(Kb*T))`. In case the candidate is rejected, we restore the old velocities and Hamiltonian that we have previously momentarily updated. The algorithm then proceeds by providing other attempts and at the very end we obtain a sample from a Gaussian distribution.

2.3.1 Simulated annealing

The method of **simulated annealing** is a technique that has attracted significant attention especially for problems where a desired global extremum (usually a minimum) is hidden among many local extrema. At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize or metals

2 Monte Carlo techniques

cool and anneal (the term indeed comes from metallurgy¹⁰). At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is progressively lost. The atoms are often able to line themselves up and form a pure crystal that is completely ordered over a large distance in all directions. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature *is* able to find this minimum energy state. Conversely, if a liquid metal is cooled quickly or “quenched”, it does not reach this state, but rather ends up in a polycrystalline or amorphous state having somewhat higher energy. So the essence of the process is slow cooling, allowing ample time for the redistribution of the atoms as they lose mobility. This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved.

The so-called Boltzmann probability distribution

$$P(E) \sim e^{-\frac{E}{k_B T}} \quad (2.3.9)$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E . Even at low temperature, there is a chance, albeit a very small one, of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum in favor of finding a better, more global one. In other words, the system sometimes goes uphill as well as downhill (in the case of Boltzmann distribution); but the lower the temperature, the less likely is any significant uphill excursion.

In 1953, Metropolis and coworkers incorporated these kinds of principles (annealing and Boltzmann distribution) into numerical calculations. Let’s start with our system at the initial temperature T and in the state of energy E_1 . At each step, the simulated annealing method considers some neighboring state E_2 of the current state E_1 , and probabilistically decides between moving the system to state E_2 or staying in state E_1 . If the move improves the situation, in the sense that minimize the energy, then the transition is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. In particular, the probability decreases exponentially with the “badness” of the move, i.e. the amount ΔE by which the evaluation is worsened: $P(E_1 \rightarrow E_2) \sim \exp(-\frac{E_2 - E_1}{k_B T})$. All the procedure is made by decreasing the initial temperature. Note that the probability of moving also decreases as the temperature goes down: “non convenient” moves are more likely to be allowed at the beginning when T is high, and they become more unlikely as T decreases. Thus, if the algorithm lowers T slowly enough, then the algorithm will ultimately find a global minimum with probability approaching 1. This general scheme, of always taking a downhill step while sometimes taking an uphill step, is the essence of the Metropolis algorithm. Here we spoke about “temperature” and “energy” by analogy with thermodynamics, but in reality they have to be identified as the characteristic parameters of the problem. For example, in the case of the traveling salesman problem, we want to minimize the distance covered.

But why do we use this algorithm, which seems so tortuous? The reason is that ill-climbing algorithms that never makes “downhill” moves toward states with lower value (or higher cost) are guaranteed to be incomplete, because they can get stuck on a *local* minimum. On the other hand, instead, a purely random walk—which is always moving to a successor chosen uniformly at random from the set of successors—is complete but

¹⁰Annealing is a heat treatment that alters the physical properties of a material to increase its ductility and reduce its hardness, making it more workable. It involves heating a material above its recrystallization temperature, maintaining a suitable temperature for an appropriate amount of time, and then gradually cooling.

extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness.

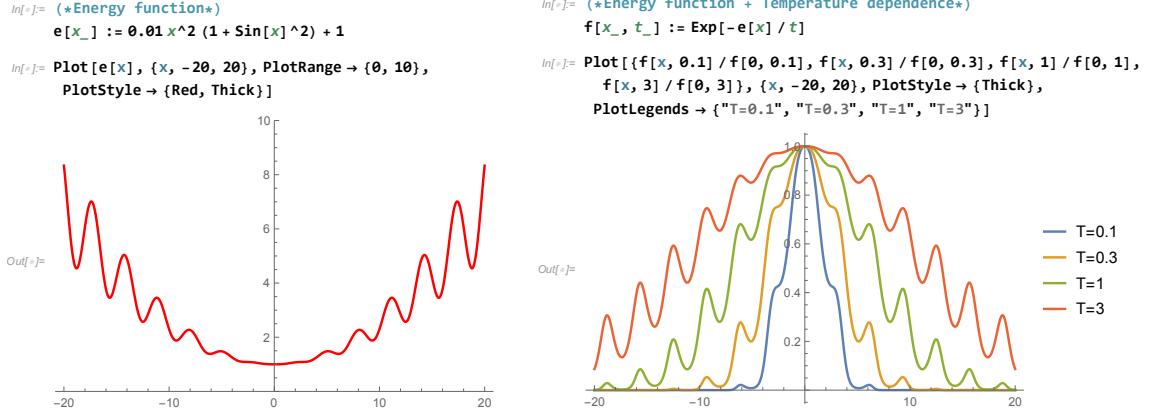


Figure 2.6: Example of energy profile. As we can notice there is one global minimum and many local minima. Clearly, if transitions to disadvantageous configurations were always forbidden, we would be trapped in those local minima. Note also that for large T (red curve), the system is more able to explore the entire distribution because it is flatter (the differences in energy between the minima are less marked).

2.3.2 Traveling salesman problem

For practical purposes, simulated annealing has effectively ‘‘solved’’ the famous traveling salesman problem of finding the shortest cyclical itinerary for a traveling salesman who must visit each of N cities in turn (other practical methods have also been found). As a problem in simulated annealing, the traveling salesman problem is handled as follows:

1. *Configuration.* The cities are numbered $i = 0, 1, \dots, N - 1$ and each has coordinates (x_i, y_i) . A configuration is a permutation of the number $0, 1, \dots, N - 1$ interpreted as the order in which the cities are visited.
2. *Rearrangements.* An efficient set of moves has been suggested by Lin [10]. The moves consist of two types: (i) A section of path is removed and then replaced with the same cities running in the opposite order; or (ii) a section of path is removed and then replaced in between two cities on another, randomly chosen, part of the path.
3. *Objective function.* In the simplest form of the problem, E is taken just as the total length of the journey

$$E \equiv L = \sum_{i=0}^{N-1} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \quad (2.3.10)$$

with the convention that point N is identified with point 0. To illustrate the flexibility of the method, however, we can add the following additional wrinkle. Suppose that the salesman has an irrational fear of flying over the Mississippi River. In that case, we would assign each city a parameter μ_i , equal to +1 if it is east of the Mississippi and -1 if it is west, and actually take the objective function to be

$$E = \sum_{i=0}^{N-1} \left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2 \right] \quad (2.3.11)$$

2 Monte Carlo techniques

A penalty 4λ is thereby assigned to any river crossing. The algorithm now finds the shortest path that avoids crossings. The relative importance that it assigns to length of path versus river crossings is determined by our choice of λ . Figure 2.8 shows the results obtained. Clearly, this technique can be generalized to include many conflicting goals in the minimization.

4. *Annealing schedule.* This requires experimentation. We first generate some random rearrangements, and use them to determine the range of values of ΔE that will be encountered from move to move. Choosing a starting value for the parameter T that is considerably larger than the largest ΔE normally encountered, we proceed downward in multiplicative steps each amounting to a 10 % decrease in T . We hold each new value of T constant for, say, $100N$ reconfigurations, or for $10N$ successful reconfigurations, whichever comes first. When efforts to reduce E further become sufficiently discouraging, we stop.

For certain aspects the analogy with genetic algorithms is strong. Here we are use something similar to crossover because first we extract two random numbers (corresponding to two cities) and we identify the part of the path that has those cities as extrema. Then, we can follow two different procedures: either exchange the order of the cities in that subsequence, or cut the entire piece and insert it somewhere else in route. The difference with respect to genetic algorithms is that here we are working with just *one* element, not an entire population. We then accept the new candidate through simulated annealing (i.e. Metropolis).

In [Figure 2.6](#) and [Figure 2.8](#) we can see another example of simulated annealing applied to a function which has a global minimum at $\mathbf{x} = 0$, but having many local minima (Fig. 2.7).

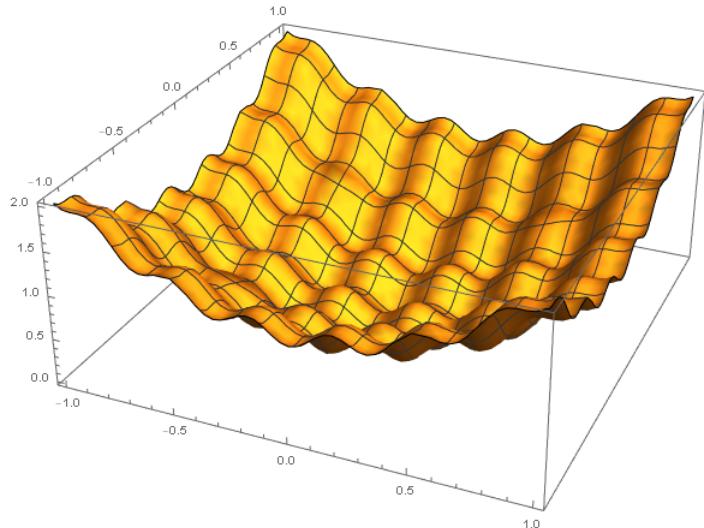


Figure 2.7

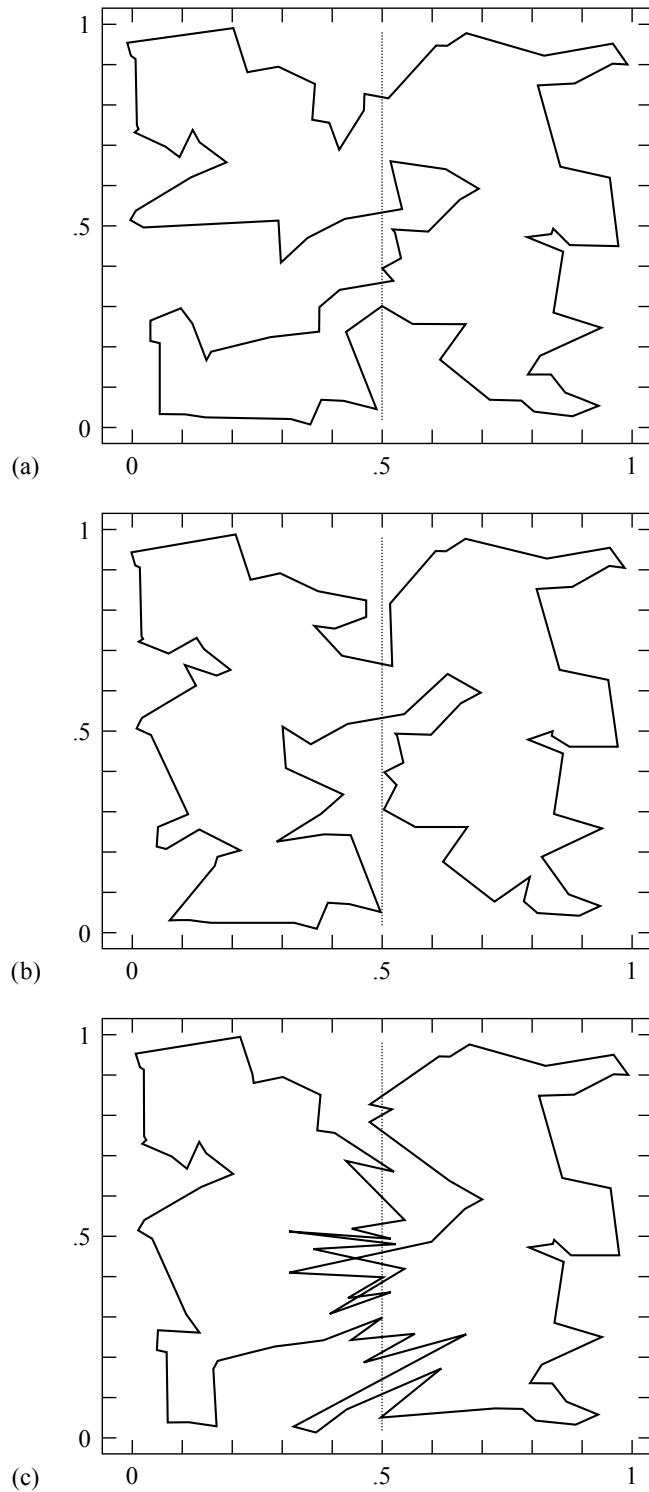


Figure 2.8: Traveling salesman problem solved by simulated annealing. The (nearly) shortest path among 100 randomly positioned cities is shown in (a). The dotted line is a river, but there is no penalty in crossing. In (b) the river-crossing penalty is made large, and the solution restricts itself to the minimum number of crossings, two. In (c) the penalty has been made negative: the salesman is actually a smuggler who crosses the river on the flimsiest excuse.

2.3.3 Monte Carlo integration

In the previous sections we have seen the guidelines of the Monte Carlo methods, of Metropolis in particular, and we have also studied some important cases such as simulated annealing applied to the traveling salesman problem. However, Monte Carlo methods can also be exploited for more “trivial” problems, such as estimating integrals.

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V ; call them $\mathbf{x}_0, \dots, \mathbf{x}_{N-1}$. Then the basic theorem of Monte Carlo integration estimates the integral of a function $f(\mathbf{x})$ over the multidimensional volume as¹¹

$$\int f dV \approx V\langle f \rangle \pm V\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \quad (2.3.12)$$

where the angle brackets denote taking the arithmetic mean over the N sample points

$$\langle f \rangle = \frac{1}{N} \sum_{i=0}^{N-1} f(\mathbf{x}_i) \quad (2.3.13)$$

$$\langle f^2 \rangle = \frac{1}{N} \sum_{i=0}^{N-1} f^2(\mathbf{x}_i) \quad (2.3.14)$$

The “plus-or-minus” term in (2.3.12) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error¹².

In numerical integration, methods such as the trapezoidal rule or Simpson’s rule use a deterministic approach. Monte Carlo integration, on the other hand, employs a non-deterministic approach: each realization provides a different outcome. In Monte Carlo, the final outcome is an approximation of the correct value with respective error bars, and the correct value is likely to be within those error bars. Based on this observation, one may wonder why should we ever need to resort to Monte Carlo integration when we know other simple ways to proceed. It happens that “classical” rules to solve integrals are simple indeed, but as the dimension of the integral increases, they become more and more expensive to use. In fact they suffer from the curse of dimensionality, where the convergence rate becomes exponentially worse as the dimension of the integral increases. On the other hand, the principle of the Monte Carlo integration can easily be extended to higher dimension and the efficiency of the method is *independent on the number of dimensions*. In fact, we have seen from (2.3.12) that the error scales as $1/\sqrt{N}$ and we don’t have any dependence on the dimensionality.

However, the basic form of the theorem written above is not always very efficient. In some cases we could have functions which are negligibly small in (large) fraction of the integration domain. In that case the uniform random number distribution is not very efficient: most of the time random numbers are generated in a part of the integration domain where the function is small, thus leading to an estimate which is smaller than the exact result. One way to overcome this problem is to increase the dimension of the sample, i.e. N , thus reducing the statistical error. But this is definitely not the most practical route, as it takes a great amount of time.

¹¹This derives essentially from $\langle\langle f \rangle\rangle = \frac{1}{V} \int f(\mathbf{x}) dV$. Here $\langle\langle f \rangle\rangle$ denote the *true* average of the function f over the volume V , while $\langle f \rangle$ denotes as before the simplest (uniformly sampled) Monte Carlo *estimator* of that average (2.3.13).

¹²We have that $\text{Var}(\langle f \rangle) = \text{Var}(f)/N$, where $\text{Var}(f) = \langle\langle f^2 \rangle\rangle - \langle\langle f \rangle\rangle^2$.

Moreover, this basic form gets even worse in high dimensions. To see this, imagine our multidimensional volume to be an hypercube (for simplicity) of linear size L . Naturally, its volume in n dimensions is given by $V(L, n) = L^n$. Subsequently, consider another cube, smaller than the previous one, and placed inside it (Figure 2.9). Calling $L - \delta$ the side of this inner cube, where δ is chosen to be significantly smaller than L , we can symbolically write its volume as $V(L - \delta, n)$. Thus, the volume of the gap between the two cubes goes approximately (Taylor expansion) as

$$\begin{aligned}\Delta V &= V(L, n) - V(L - \delta, n) \\ &= L^n - (L - \delta)^n \\ &= nL^{n-1}\delta + \dots\end{aligned}\tag{2.3.15}$$

whence¹³

$$\frac{\Delta V}{V} \simeq n \frac{\delta}{L}\tag{2.3.16}$$

The ratio $\Delta V/V$ indicates the fractional volume of the gap with respect to the volume of the biggest cube. As we can see, this fraction is extremely small when the number of dimensions n is near unit (because $\delta \ll L$). On the other hand, when n is large the empty space between the cubes can increase and become no more negligible. Now, let interpret the inner cube as the region in which the integrand function f gives larger contributions to the integral, while let the outer cube indicates the total domain of integration. The gap between them then represents the region in which f is negligibly small. Therefore it is clear from (2.3.16) that sampling random points from a uniform distribution is not efficient at all! If n is small, then we don't waste many random points to compute an estimate of the integral because only a very small fraction of them will fall in ΔV . But if n is large, the fractional volume of the gap can become very large, and as a consequence a huge number of uniformly sampled points is wasted in what becomes now an “underestimation” of the integral.

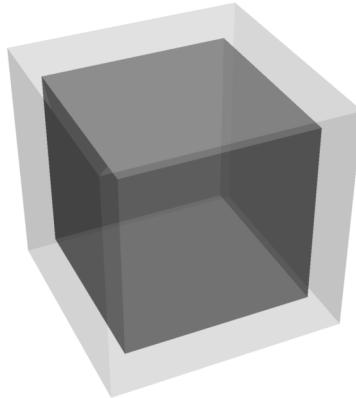


Figure 2.9

It is then more convenient to use a random number generator which samples more often the part of the integration domain where the function $f(\mathbf{x})$ has relatively large value. This procedure is called **importance sampling**. The use of importance sampling was already implicit in the previous reasoning. We now return to it in a slightly more formal way. Suppose that an integrand f can be written as the product of a function h that is almost

¹³From binomial theorem $(a + b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k$ with $b \equiv -\delta \ll a \equiv L$.

2 Monte Carlo techniques

constant times another, positive, function p . Then its integral over a multidimensional volume V is

$$\int f \, dV = \int \left(\frac{f}{p} \right) p \, dV = \int h \, p \, dV \quad (2.3.17)$$

A more general interpretation of equation (2.3.17) is that we can integrate f by instead sampling h —not, however, with uniform probability density dV (remember (2.2.4)), but rather with non-uniform density $p \, dV$. And at this point we just need to extract points from p through some known technique, like the Metropolis algorithm, for instance. Formally we extract random points according to the p , which must be normalized:

$$\int p \, dV = 1 \quad (2.3.18)$$

and the estimate of the integral becomes

$$\int f \, dV = \int \left(\frac{f}{p} \right) p \, dV \simeq \left\langle \frac{f}{p} \right\rangle \pm \sqrt{\frac{\langle f^2/p^2 \rangle - \langle f/p \rangle^2}{N}} \quad (2.3.19)$$

where

$$\left\langle \frac{f}{p} \right\rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)} \quad (2.3.20)$$

$$\left\langle \frac{f^2}{p^2} \right\rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f^2(\mathbf{x}_i)}{p^2(\mathbf{x}_i)} \quad (2.3.21)$$

The only difference is that now the points \mathbf{x}_i are drawn from p . In this second interpretation, the first interpretation follows as the special case when $p = \text{const.} = 1/V$.

With this method, we are free to choose the probability distribution $p(\mathbf{x})$ to sample the integration interval; in the limit $N \rightarrow \infty$ the method is independent on the particular choice of the distribution $p(\mathbf{x})$. This choice is very important, given that the error on the estimate in equation (2.3.19) depends on the function $p(\mathbf{x})$ we decide to use. The idea behind the importance sampling is then to choose the new probability density p such that it minimizes the variance of the ratio $f(\mathbf{x})/p(\mathbf{x})$. To do so we choose a form of $p(\mathbf{x})$ that mimics $f(\mathbf{x})$ as much as possible (particularly where $f(\mathbf{x})$ is large), so that their ratio becomes a slow varying function.

2.3.4 Correlations

Metropolis algorithm has many pros, but it has a delicate con: it originates *correlated* samples. Even though over the long term they do correctly follow $P(x)$, a set of nearby samples will be correlated with each other and does not correctly reflect the distribution. As a consequence, it's not true that the error in Monte Carlo integration scales as $1/\sqrt{N}$, because that formula (2.3.12) holds only for uncorrelated variables (remember what we said in Section 2.1.1 about covariance).

Formally speaking, the connection between couples of points sampled at a distance of k algorithm steps one from the other is furnished by the so-called **autocorrelation function**. For a discrete process with known theoretical mean and variance for which we get N observations $\{X_1, X_2, \dots, X_N\}$, an estimate of the autocorrelation function at lag k may be obtained as

$$\hat{R}(k) = \frac{1}{(N-k)\sigma^2} \sum_{t=1}^{N-k} (X_t - \mu)(X_{t+k} - \mu) \quad (2.3.22)$$

for any positive integer $k < N$. As we can see, when $k = 0$, the autocorrelation function is always equal to 1 because

$$\sigma^2 = \frac{1}{N} \sum_{t=1}^N (X_t - \mu)^2 \quad (2.3.23)$$

is the empirical variance of the sample. On the other hand, for $k > 0$ we should expect $\hat{R}(k)$ to be zero if the variables X_i were completely independent. In such a case the value of $(X_t - \mu)$ wouldn't have any connection to that of $(X_{t+k} - \mu)$: sometimes they have the same sign, sometimes not, and for this reason their product should average statistically to zero. However, Metropolis algorithm does not produce a sample of independent variables! Since it reads $X' = X_t + r\Delta$, with $r \in [-1, 1]$ and Δ typical scale of the distribution, then by taking Δ small the product $r\Delta$ doesn't allow to explore the whole distribution in just one step. This means that the point X_{t+2} is close to X_{t+1} , which in turn is close to X_t , etc. Accordingly, it is very probable that the two pieces $(X_t - \mu)$ and $(X_{t+k} - \mu)$ have the same sign because the algorithm allows only small steps. So for small k the autocorrelation function is not zero, but actually has a finite value. Only for k large the curve lowers until it enters a zone close to 0, which can be interpreted as statistical fluctuations around 0 (see Figure 2.10).

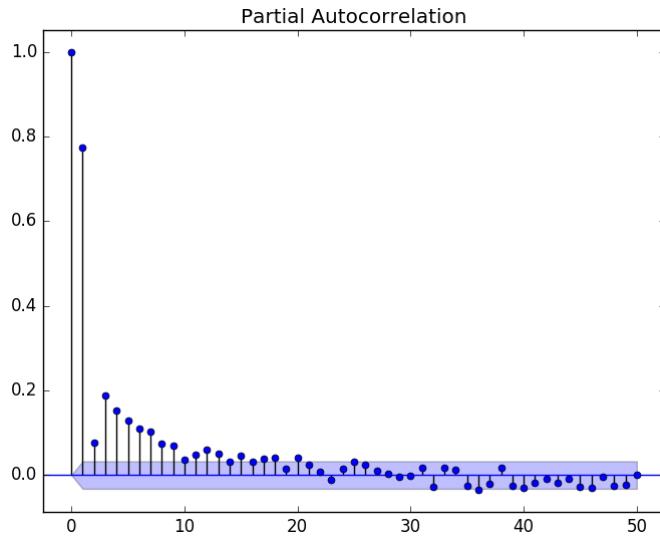


Figure 2.10

From the figure under consideration it can be seen that for $k \approx 10$ the autocorrelation function has essentially dropped in the narrow band, that is points generated at distance 10 can essentially be considered as uncorrelated. This means that if we want a set of independent samples generated from a Metropolis algorithm, we have to throw away the majority of samples and only take points at steps of \bar{k} , where \bar{k} is the time step at which points starts getting uncorrelated. In other words, from the initial sample

$$S = \{X_1, X_2, \dots, X_{\bar{k}}, X_{\bar{k}+1}, X_{\bar{k}+2}, \dots\} \quad (2.3.24)$$

which is correlated, we extract the uncorrelated subset

$$RS = \{X_1, X_{\bar{k}+1}, X_{2\bar{k}+1}, X_{3\bar{k}+1}, \dots\} \quad (2.3.25)$$

and only this set can be used to estimate the error as given in (2.3.12).

2 Monte Carlo techniques

Autocorrelation can also be reduced by increasing the jumping width Δ , but this will also increase the likelihood of rejection of the proposed jump. In fact, big jumps very often lead to regions in which the density function $P(x)$ we want to sample is negligible, thus making the Metropolis algorithm discard them almost every time. In both cases, too large or too small a jumping size will lead to a slow-mixing Markov chain, i.e. a highly correlated set of samples, so that a very large number of samples will be needed to get a reasonable estimate of any desired property of the distribution. Roughly speaking, Metropolis should work fine if we are able to keep half of the points being generated, even if we still have to make the analysis of the correlation function.

CHAPTER 3

Neural networks

Contents

3.1	Basic introduction to neuroscience	65
3.1.1	Neurons and brain	65
3.2	Artificial neural networks	69
3.2.1	Deterministic networks: associative memory	69
3.2.2	Learning by Hebb's rule	72
3.2.3	Diederich and Opper method	75
3.2.4	Spin glasses	78
3.2.4.1	Parallel versus sequential dynamics	79
3.2.5	Neural "motion pictures"	81
3.2.6	Stochastic neurons	83
3.2.6.1	Single pattern	86
3.2.6.2	Several patterns	87
3.2.7	Special learning rules	89
3.3	Simple perceptron	90
3.3.1	The exclusive-OR (XOR) gate	95
3.4	Multilayer perceptron	98
3.4.1	Solution of the XOR problem	98
3.4.2	Error back-propagation	100
3.4.2.1	Arbitrary number of hidden layers	101
3.4.3	Boolean functions	104
3.4.4	Continuous functions	107
3.4.5	Generalization and fitting	110
3.5	Boltzmann machines	113
3.5.1	Information theory	115
3.5.2	Mutual information	116
3.5.3	The "Boltzmann" learning rule	119
3.5.3.1	Applications	123
3.6	Restrictive Boltzmann machines	124

3.1 Basic introduction to neuroscience

3.1.1 Neurons and brain

Artificial neural networks are computing systems vaguely inspired by the biological neural networks that constitute animal brains. For this reason, it is worthwhile to illustrate even only briefly the characteristics of the neurons, the cells of our brain. Neurons are highly

3 Neural networks

specialized cells for the collection and conduction of nerve impulses, which represent the morphological, genetic and functional units of the nervous system. A neuron is made up of a bulbous cell body and two types of extensions: the *dendrites*, which are root-like extensions that conduct the incoming nerve stimulus from the periphery of the cell to the central body, and the *axon*, which conducts the stimulus to other neurons with which it enters in connection through specialized junctions, called *synapses* (Figure 3.1).

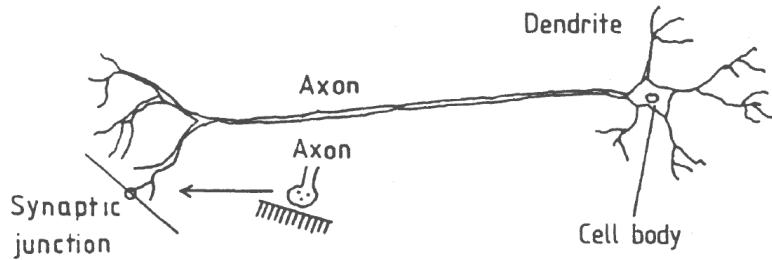


Figure 3.1: Schematic structure of a neuron.

This is the basic structure of a neuron, but they can have very different forms, depending on the type of task they have to perform (Fig. 3.2). They can be simply asymmetric, or can have deeply interconnected structures.

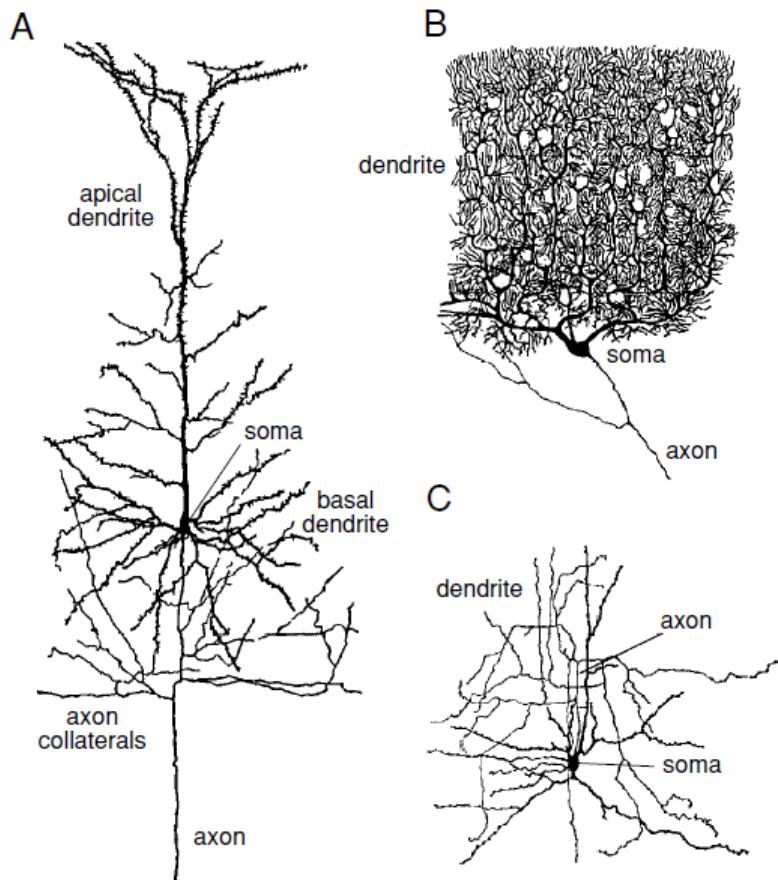


Figure 3.2: Diagrams of three neurons: (A) a cortical pyramidal cell; (B) a Purkinje cell of the cerebellum; (C) a stellate cell of the cerebral cortex.

In the human cortex there are about 150 000 neurons per mm², that corresponds to a total of about 3×10^{10} neurons. Each neuron has about 10^4 synaptic connections and

therefore the number of synaptic junctions in the human brain is about 10^{15} (so a very huge number), the majority of which develop within a few months after birth (Fig. 3.3).

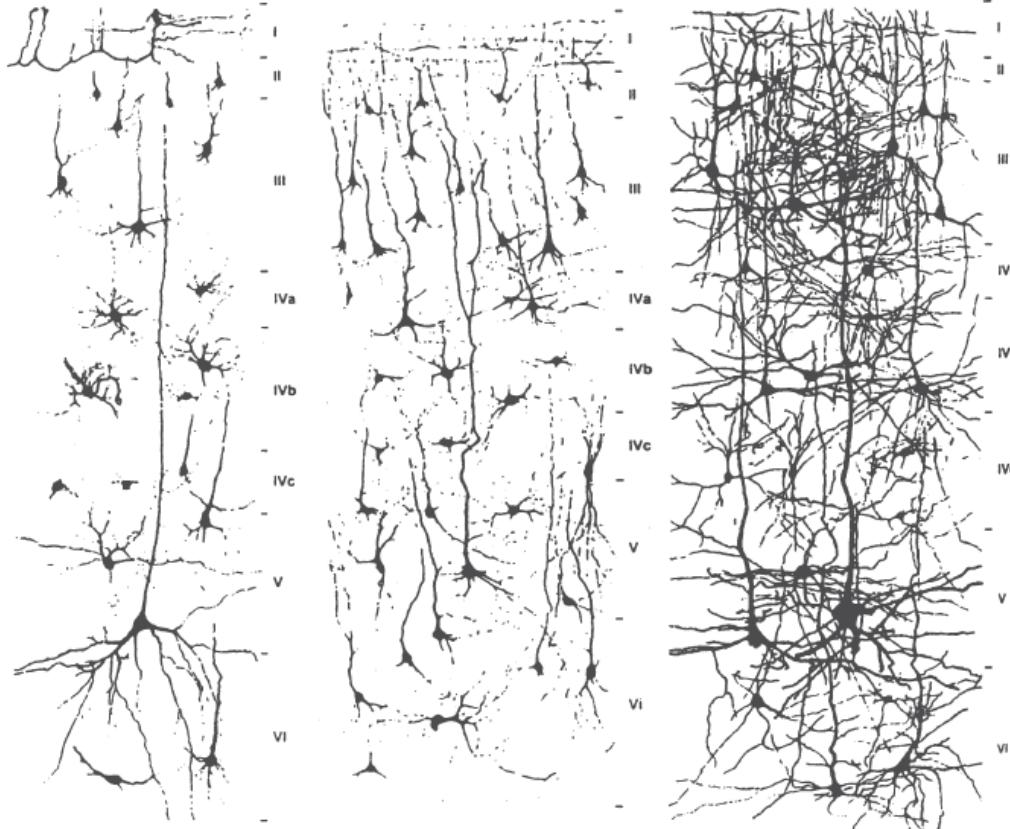


Figure 3.3: Development of dendritic arborization in the human visual cortex, from (left) newborn, (middle) three-month-old, and (right) two-year-old infants.

Nervous signals are transmitted either electrically or chemically. Electrical transmission prevails in the interior of a neuron, whereas chemical mechanisms operate between different neurons, i.e. at the synapses. The body of the neuron acts as a “summing device”. The effects of a stimulus decay after 5–10 ms, but if several signals arrive at the same synapse over such a period their excitatory effects accumulate. A high rate of repetition of firing of a neuron therefore expresses a large intensity of the signal (Fig. 3.4). When the total magnitude of the depolarization potential in the cell body exceeds the critical threshold (about 10 mV), the neuron fires. In this sense, the response of the neuron to the incoming information from dendrites is *non-linear*: the total signal is transmitted only if the input exceeds a certain threshold.

The influence of a given synapse therefore depends on several aspects: the inherent strength of its depolarizing effect, its location with respect to the cell body and the repetition rate of the arriving signals. There is a great deal of evidence that the inherent strength of a synapse is not fixed once and for all. As originally postulated by the psychologist D. Hebb, *the strength of a synaptic connection can be adjusted, if its level of activity changes. An active synapse, which repeatedly triggers the activation of its post-synaptic neuron, will grow in strength, while others will gradually weaken*. In other words, if we are systematically associating together many things, then the connection is strengthened (the neuron discharges easily), otherwise it is weakened. This mechanism, which is the essence

3 Neural networks

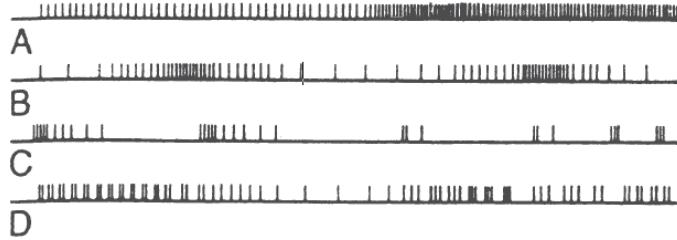


Figure 3.4: Neuron as pulse-coded analog device: spike trains of some typical neural transmission patterns. The microstructure of the successive intervals is increasingly important from top to bottom (A–D), indicating messages of growing complexity.

of the so-called **Hebb's law**, appears to play a dominant role in the complex process of learning.

It is an interesting question in what respect the human central nervous system differs from that of other higher animals. Here it is important to relate the size of the brain to the total size of the animal. Studies in vertebrate animals have shown that the weight of the brain E grows with body weight P for species at a comparable level of evolution. Quantitatively, a quite strict relation of the form

$$\ln E = a \ln P + c \quad (3.1.1)$$

with $a \approx 0.6$ was found (see Fig. 3.5). Now, animals are mainly made of water, i.e. they have approximately the same density ρ ; hence their weight is directly proportional to the body volume, $P \propto M \simeq \rho V$, which in turn is related to the surface through $S \sim V^{2/3} \sim P^{2/3}$. Since the surface of a body grows as $S \sim P^{2/3}$ (with $a \approx 0.6$ very close to the fraction $2/3$), one may suspect that the brain size grows in proportion to the body surface: $E \sim S$. This result is not too much of a surprise, because the amount of external sensory information that the brain has to process increases roughly in proportion to the surface of the body. The more extended is a body, the greater is the possibility to collect stimuli from the outside.

Vertebrates belonging to different levels of evolution differ, not so much in the density of neurons, but in the magnitude of the constant of proportionality c in the relation (3.1.1). For example

$$\frac{c_{\text{ape}}}{c_{\text{rodent}}} \simeq 11.3, \quad \frac{c_{\text{man}}}{c_{\text{rodent}}} \simeq 28.7 \quad (3.1.2)$$

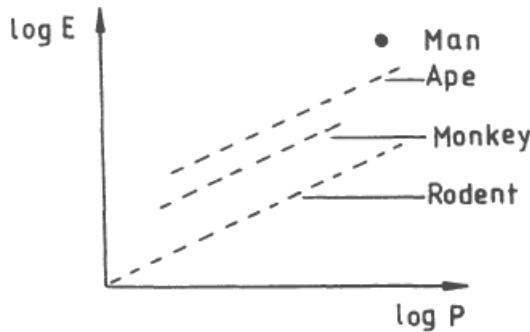


Figure 3.5: Relation between brain size (E) and body weight (P) for various animals in comparison to man.

3.2 Artificial neural networks

3.2.1 Deterministic networks: associative memory

Neural network models are algorithms for cognitive tasks, such as learning and optimization, which are in a loose sense based on concepts derived from research into the nature of the brain. In the standard terminology, the nodes are called “neurons” and the links are called “synapses”. The new state of a certain neural unit is determined by the influence of all other neurons, as expressed by a *linear* combination of their output values:

$$h_i(t) = \sum_{j=1}^N w_{ij} n_j(t) \quad (3.2.1)$$

Here the matrix w_{ij} represents the synaptic coupling strengths between neurons j and i , while $h_i(t)$ models the total post-synaptic polarization potential at neuron i caused by the action of all other neurons. h_i can be considered the *input* into the neural computing unit, and n_i the *output*. The properties of the neural network are completely determined by the functional relation between $h_i(t)$ and $n_i(t+1)$. In the simplest case, the neuron is assumed to become active if its input exceeds a certain threshold ϑ_i , which may well differ from one unit to the next. The evolution of the network is then governed by the law

$$n_i(t+1) = \Theta(h_i(t) - \vartheta_i) \quad (3.2.2)$$

where $\Theta(x)$ is the unit step function, i.e.

$$\Theta(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases} \quad (3.2.3)$$

This relation essentially accounts for the non-linearity in exit we talked about earlier.

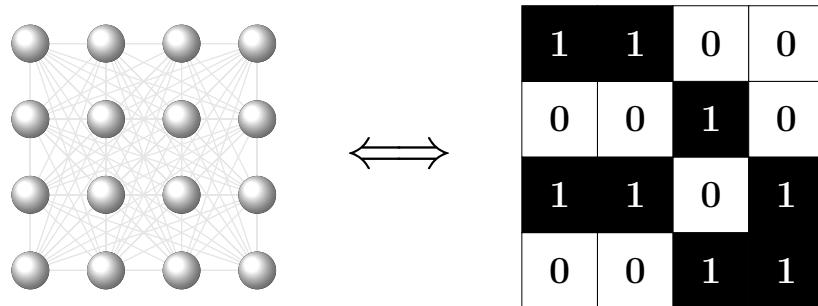


Figure 3.6: Schematic representation of a simple neural network, where the neurons n_i can be active ($n_i = 1$) or not ($n_i = 0$).

At this point one may wonder how the updating of the state of each neuron occurs. One possibility is to update all neurons synchronously according to the law (3.2.2), that is at a certain time t the functions $h_i(t)$ for all neurons are simultaneously computed. On the other hand, the second way is to update sequentially one at a time (either in a certain fixed order or randomly) the state of the neurons; for example, at time $t = 1$ we update one neuron, at time $t = 2$ we update another neuron, and so on. The first idea of working in *parallel* is called **Little model**, while the idea of working *sequentially* is called **Hopfield model**, from the names of the two scientists that first studied this matter. The Little and Hopfield models differ in the way in which the state of the system is updated. Sequential

3 Neural networks

updating has a considerable advantage when the network is simulated on a conventional digital computer, and also for theoretical analysis of the properties of the network. On the other hand, it holds the essential conceptual disadvantage that the basic feature of neural networks, namely the simultaneous operation of a large number of parallel units, is given up. Neurons in the human brain surely do *not* operate sequentially, this being precisely the reason for the brain's superiority in complex tasks to even the fastest existing electronic computer. In fact, the "brain clock" is not so fast, and so working sequentially would be very bad; but despite its slowness, we have a huge number of neurons which compensate this "defect" by working in parallel. In the following we will use both methodologies for instructing purposes.

Associative memory, i.e. storage and recall of information by association with other information, may be the simplest application of "collective" computation on a neural network. With traditional computers the recall of information requires *precise* knowledge of the memory address. However, it is obvious that our human memory cannot be organized in this way. For example, imagine that the memory has stored the idea of an apple. What does it happen if we see a "modified" image of an apple, for instance because it has bitten or is rotten? The apple is still an apple even under modification, but a traditional computer wouldn't recognize that it is still an apple because it has only stored the information of a healthy apple. Instead, human brain is very good in generalizing, and hence still recognizes the object as an apple. And this is what we want to achieve in our first simple model.

Here we define an associative memory as follows. Assume that p binary patterns containing N bits of information, ν_i^μ ($i = 1, \dots, N$; $\mu = 1, \dots, p$), are stored in the memory. In other words we have a certain number of "images" ν^μ ($\mu = 1, \dots, p$) and each of them is decomposed in N bits of information ($i = 1, \dots, N$); so ν_i^μ represents the i -th bit of the μ -th pattern. Each bit can be 0 or 1 and our patterns are something like what we see in Figure 3.7.

0	0	0	0	0
0	1	1	1	0
0	1	0	0	0
0	1	0	0	0
0	1	1	1	0
0	0	0	0	0

0	0	0	0	0
0	1	1	1	0
0	1	0	0	0
0	1	1	1	0
0	1	0	0	0
0	0	0	0	0

Figure 3.7: Letters C and F in bits.

If now a new pattern n_i is presented (i.e. a new configurations of 0s and 1s), we would have that the system recalls that stored pattern ν_i^λ which most strongly resembles the presented one. Here n_i (always $i = 1, \dots, N$) is the actual configuration of the net: essentially, each neuron contains a different bit of memory (so we have N neurons), and we are asking which stored pattern actually resembles the present configuration of the net. To do so, we define the so-called **Hamming distance**

$$H_\mu = \sum_{i=1}^N (n_i - \nu_i^\mu)^2 \quad (3.2.4)$$

and we want it to be minimal for $\mu = \lambda$. In principle, this problem is easily solved on a standard digital computer by computing all values H_μ and then searching for the smallest

one. However, if the memory contains many large patterns, this procedure can take quite a long time. We are therefore led to ask the question whether the patterns can be stored in a neural network of N elements (neurons) in such a way that the network evolves from the initial configuration n_i , corresponding to the presented pattern, into the desired configuration ν_i^λ under its own dynamics.

A very fruitful development comes by noticing the similarity between a neural network of the type we have proposed and systems of elementary magnetic moments or spins (see Figure 3.8). In these Ising systems the spin s_i at each lattice site i can take only two different

$$\downarrow \uparrow \uparrow \downarrow \downarrow \uparrow \uparrow \uparrow \downarrow \downarrow \uparrow$$

Figure 3.8

orientations, up or down, denoted by $s_i = +1$ (up) and $s_i = -1$ (down). The analogy to a neural network is realized by identifying each spin with a neuron and associating the upward orientation $s_i = +1$ with the active state $n_i = 1$ and the downward orientation $s_i = -1$ with the resting state $n_i = 0$. Therefore we introduce new variables s_i and σ_i , which are defined as

$$s_i := 2n_i - 1 = \begin{cases} +1, & \text{if } n_i = 1 \\ -1, & \text{if } n_i = 0 \end{cases} \quad (3.2.5)$$

$$\sigma_i := 2\nu_i - 1 = \begin{cases} +1, & \text{if } \nu_i = 1 \\ -1, & \text{if } \nu_i = 0 \end{cases} \quad (3.2.6)$$

By adopting these new variables, the addends in (3.2.4) can be rewritten in the following way:

$$(n_i - \nu_i^\mu)^2 = \frac{1}{4}(s_i - \sigma_i^\mu)^2 = \frac{1}{4}[s_i^2 - 2s_i\sigma_i^\mu + (\sigma_i^\mu)^2] = \frac{1}{2}(1 - s_i\sigma_i^\mu) \quad (3.2.7)$$

whence the Hamming distance

$$H_\mu = \frac{1}{2} \sum_{i=1}^N (1 - s_i\sigma_i^\mu) \quad (3.2.8)$$

Now, the search for the minimum of H_μ is then equivalent to the search for the maximal value of the function

$$A_\mu = \sum_{i=1}^N s_i\sigma_i^\mu \quad (3.2.9)$$

which a scalar product of the N -component vectors s_i and σ_i . Clearly, the optimal solution would be $s_i\sigma_i^\mu = 1$ for all i because now these new variables are unitary, and this corresponds to the vanishing of the Hamming distance. In fact, we know that $H_\mu = 0 \Leftrightarrow s_i = \sigma_i^\mu \forall i$ (or equivalently $s_i\sigma_i^\mu = 1$), because the Hamming distance is zero if there is a perfect overlap between the configuration of the net and the one of the stored pattern.

At this point all that remains is to consider how the state of the network evolves dynamically. According to Hopfield sequential model, the individual neurons i are assigned new values $s_i(t+1)$ in some randomly chosen sequence (i.e. we pick the neuron i randomly and at time $t+1$ we update only it). The new values are computed according to the non-linear rule

$$s_i(t+1) = \operatorname{sgn} [h_i(t)] \quad (3.2.10)$$

3 Neural networks

with

$$h_i(t) = \sum_{j=1}^N \omega_{ij} s_j(t) \quad (3.2.11)$$

where the instantaneous output values of all neurons are to be taken on the right-hand side. The temporal evolution proceeds, as already mentioned above, in finite steps, i.e. t takes only the discrete values $t = 0, 1, 2, \dots$. Note that the equations (3.2.10) corresponds to the rules (3.2.1) and (3.2.2), rewritten in the new variables (and without threshold).

3.2.2 Learning by Hebb's rule

The immediate problem consists in choosing the synaptic coupling strengths w_{ij} given the stored patterns σ_i^μ , such that the network evolves from the presented pattern s_i into the most similar stored pattern by virtue of its own inherent dynamics. We begin with the simple, but not trivial, case of a single stored pattern σ_i ($\mu = p = 1$) and with the simple choice

$$w_{ij} = \frac{1}{N} \sigma_i \sigma_j \quad (3.2.12)$$

Now let's show that using (3.2.12):

- each stored pattern corresponds to a stable configuration of the network;
- small deviations from the stored pattern will be automatically corrected by the network dynamics.

Proof 1. To see this, we calculate the synaptic potentials h_i , assuming that $s_i(t) = \sigma_i$ for all i . By virtue of the relation $(\sigma_i)^2 = (\pm 1)^2 = 1$ we have

$$h_i(t) = \sum_{j=1}^N \omega_{ij} \sigma_j = \frac{1}{N} \sum_{j=1}^N \sigma_i \underbrace{(\sigma_j)^2}_{=1} = \sigma_i \frac{1}{N} \sum_{j=1}^N 1 = \sigma_i \quad (3.2.13)$$

Inserting this result into the evolution law (3.2.10) we find, as required

$$s_i(t+1) = \operatorname{sgn}[h_i(t)] = \operatorname{sgn}(\sigma_i) = \sigma_i = s_i(t) \quad (3.2.14)$$

that is at the next time the status of the net is not changed; the net is in a stable configuration. \square

Proof 2. Now assume that the network would start its evolution not exactly in the memorized state σ_i , but some of the elements had the wrong value (-1 instead of $+1$, or vice versa), i.e. it is partially corrupted. Without loss of generality we can assume that these are the first n elements of the pattern, that is

$$s_i(t=0) = \begin{cases} -\sigma_i, & \text{for } i = 1, \dots, n \\ +\sigma_i, & \text{for } i = n+1, \dots, N \end{cases} \quad (3.2.15)$$

Then the synaptic potentials are

$$\begin{aligned}
 h_i(t=0) &= \sum_{j=1}^N w_{ij} s_j(t=0) \\
 &= \frac{1}{N} \sigma_i \sum_{j=1}^N \sigma_j s_j(t=0) \\
 &= \frac{1}{N} \sigma_i \left[-\sum_{j=1}^n (\sigma_j)^2 + \sum_{j=n+1}^N (\sigma_j)^2 \right] \\
 &= \frac{1}{N} \sigma_i [-n + (N-n)] \\
 &= \left(1 - \frac{2n}{N}\right) \sigma_i
 \end{aligned} \tag{3.2.16}$$

where in the last steps we broke in two the summation, distinguishing among right ($j = 1, \dots, n$) and wrong bits ($j = n+1, \dots, N$). In conclusion, for $n < \frac{N}{2}$ the network makes the transition into the correct stored pattern after a single update of all neurons:

$$s_i(t=1) = \text{sgn}[h_i(t=0)] = \text{sgn}\left[\left(1 - \frac{2n}{N}\right) \sigma_i\right] = \text{sgn}(\sigma_i) = \sigma_i \tag{3.2.17}$$

One also says that the stored pattern acts as an *attractor* for the network dynamics. \square

We are now ready to consider the general case, where p patterns $\sigma_i^1, \dots, \sigma_i^p$ are to be stored. Generalizing (3.2.12), we choose the synaptic strengths according to the rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \tag{3.2.18}$$

which is usually referred to as **Hebb's rule**. As we did before, we check again the two properties of stability and correction.

Proof 1. Let's select one specific pattern ν for the initial state of the neural network and study its stability. Assuming $s_i(t) = \sigma_i^\nu$, then the synaptic potentials are given by

$$\begin{aligned}
 h_i^{(\nu)}(t) &= \sum_{j=1}^N \omega_{ij} \sigma_j^\nu \\
 &= \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sum_{j=1}^N \sigma_j^\mu \sigma_j^\nu \\
 &= \frac{1}{N} \left[\sigma_i^\nu \sum_{j=1}^N \overbrace{\sigma_j^\nu \sigma_j^\nu}^{=1} + \sum_{\mu \neq \nu} \sigma_i^\mu \sum_{j=1}^N \sigma_j^\mu \sigma_j^\nu \right] \\
 &= \sigma_i^\nu + \frac{1}{N} \sum_{\mu \neq \nu} \sigma_i^\mu \sum_{j=1}^N \sigma_j^\mu \sigma_j^\nu
 \end{aligned} \tag{3.2.19}$$

where we split the summation in two parts: one for $\mu = \nu$ and the other for $\mu \neq \nu$. The first term is identical with the one in (3.2.13), obtained for a single stored pattern, but the

3 Neural networks

second term is not desirable if after we want to apply the sign function. Therefore, at this point we make the strong assumption that the memorized patterns are *uncorrelated*. In the case that the stored patterns are uncorrelated, one sees that the second term contains in all $N(p - 1)$ pieces randomly signed contributions (± 1), i.e. the bits themselves are randomized. Hence, according to the laws of statistics¹, for large N and p its value will typically be of size

$$\frac{1}{N} O\left(\sqrt{(p-1)N}\right) = O\left(\sqrt{\frac{p-1}{N}}\right) \quad (3.2.20)$$

leading to

$$h_i^{(\nu)}(t) = \sigma_i^\nu + O\left(\sqrt{\frac{p-1}{N}}\right) \quad (3.2.21)$$

As long as $p \ll N$, i.e. when the number of stored patterns is much smaller than the total number of neurons in the network, the additional term will most likely not affect the sign of the complete expression $h_i^{(\nu)}$, which alone determines the reaction of the i -th neuron. The smaller the ratio p/N , the greater the likelihood that the pattern ν is a stable configuration of the network:

$$s_i(t+1) = \text{sgn}\left[h_i^{(\nu)}(t)\right] \simeq \text{sgn}(\sigma_i^\nu) = \sigma_i^\nu = s_i(t) \quad (3.2.22)$$

□

Proof 2. Again, if n neurons start out in the “wrong” state

$$s_i(t=0) = \begin{cases} -\sigma_i^\nu, & \text{for } i = 1, \dots, n \\ +\sigma_i^\nu, & \text{for } i = n+1, \dots, N \end{cases} \quad (3.2.23)$$

a combination of the previous considerations yields²

$$\begin{aligned} h_i^{(\nu)}(t=0) &= \sum_{j=1}^N w_{ij} s_j(t=0) \\ &= \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sum_{j=1}^N \sigma_j^\mu s_j(t=0) \\ &= \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \left[-\sum_{j=1}^n \sigma_j^\mu \sigma_j^\nu + \sum_{j=n+1}^N \sigma_j^\mu \sigma_j^\nu \right] \\ &= \frac{1}{N} \sigma_i^\nu [-n + (N-n)] + \frac{1}{N} \sum_{\mu \neq \nu} \sigma_i^\mu \sum_j' \sigma_j^\mu \sigma_j^\nu \\ &= \left(1 - \frac{2n}{N}\right) \sigma_i^\nu + O\left(\sqrt{\frac{p-1}{N}}\right) \end{aligned} \quad (3.2.24)$$

¹This is a consequence of the central limit theorem: the sum of a large number, N , of independent random variables will obey a Gaussian distribution centered at N times the mean value and having a standard deviation which is \sqrt{N} times the standard deviation of the original probability distribution. For our special case this has been deduced from [18].

²The index j in the primed summation still runs from 1 to N , with the only difference that for $j = 1, \dots, n$ a minus sign in front of the corresponding terms must be understood. The “prime” is just a symbolic notation to indicate more compactly what it is written in the previous line. But if the patterns are randomly correlated, then it’s useless to make this distinction because in both cases we still have randomized ± 1 .

Under the conditions $n < \frac{N}{2}$ and $p \ll N$ we will still have $s_i(t = 1) = \text{sgn}(\sigma_i^\nu) = \sigma_i$, i.e. the network configuration will converge to the desired pattern within a single global update. However, if the number of stored patterns is comparable to the number N of neurons, the second, randomly distributed term in (3.2.24) becomes of order one, and the patterns can no longer be recalled reliably. It is possible to demonstrate that this undesirable case occurs when the number p of stored random patterns exceeds 14 % of the number N of neurons, i.e. if $p/N > 0.14$. \square

This limit becomes significantly higher if the various stored patterns happen to be orthogonal to each other, i.e. if their scalar products satisfy the conditions

$$\frac{1}{N} \sum_i \sigma_i^\mu \sigma_i^\nu = \delta_{\mu\nu} \quad (3.2.25)$$

Obviously, the disturbing ‘‘noise’’ term in (3.2.24) vanishes in this case, and hence it is possible to store and retrieve N patterns. In fact, having at disposal N bits for each pattern, we can have at most N patterns different from each other and satisfying the orthogonality condition. In the next section we will show that there exist ‘‘learning’’ protocols, i.e. rules for the choice of the synaptic connections, which are superior to Hebb’s rule and permit the storage of up to $2N$ retrievable patterns, and which even work in the presence of correlations.

3.2.3 Diederich and Opper method

The problem of discriminating between correlated patterns has a remarkably simple solution, which even permits the storage of $p = N$ arbitrarily correlated patterns, as long as they are linearly independent. To see how it works, we form the matrix of scalar products between all pairs of patterns ($\sigma_i^\mu = \pm 1$):

$$Q_{\mu\nu} := \frac{1}{N} \sum_i \sigma_i^\mu \sigma_i^\nu \quad (3.2.26)$$

For *linearly independent* patterns the matrix Q is invertible, and we can define the following improved synaptic coupling strengths:

$$\tilde{w}_{ij} = \frac{1}{N} \sum_{\mu,\nu} \sigma_i^\mu (Q^{-1})_{\mu\nu} \sigma_j^\nu \quad (3.2.27)$$

Mathematically, (3.2.27) corresponds to a projection technique that eliminates the existing correlations between patterns, hence this learning rule is often called the *projection rule*. With this choice the interaction of the stored patterns due to the fluctuating term in (3.2.19) vanishes exactly, as can be easily seen by computing the post-synaptic potentials in the

3 Neural networks

presence of one of the memorized patterns σ_i^λ :

$$\begin{aligned}
\tilde{h}_i^{(\lambda)} &= \sum_{j=1}^N \tilde{w}_{ij} \sigma_j^\lambda \\
&= \frac{1}{N} \sum_{\mu, \nu} \sigma_i^\mu (Q^{-1})_{\mu\nu} \sum_{j=1}^N \sigma_j^\nu \sigma_j^\lambda \\
&= \sum_{\mu, \nu} \sigma_i^\mu (Q^{-1})_{\mu\nu} Q_{\nu\lambda} \\
&= \sum_{\mu} \sigma_i^\mu \delta_{\mu\lambda} \\
&= \sigma_i^\lambda
\end{aligned} \tag{3.2.28}$$

We conclude that every stored pattern represents a stable network configuration, independent of correlations among the patterns. Of course, the condition $p < N$ continues to limit the memory capacity, since at most N linearly independent patterns can be formed from N units of information.

This procedure seems to work quite well, but there's a non-negligible problem. The practical application of the projection learning rule for large, memory saturated networks suffers from the *need to invert* the $p \times p$ matrix $Q_{\mu\nu}$, which poses a formidable numerical problem. In fact, apart for simple cases like exactly diagonal or quasi-diagonal matrices, the computation of an inverse can become very time-consuming. Fortunately, the matrix inversion needs to be performed only once, when the patterns are stored into the network. The ingrained memory can then be recalled as often as desired without additional effort. A practical method of implementing the projection rule is based on an iterative scheme, where the “correct” synaptic connections are strengthened in order to stabilize the correlated patterns against each other. This procedure goes under the name of **Diederich and Opper method**.

Because of the neuron evolution law $s_i(t+1) = \text{sgn}[h_i(t)]$ any presented pattern σ_i^μ ($s_i(t) = \sigma_i^\mu$) represents a stable network configuration if h_i has the same sign as σ_i , that is

$$\sigma_i^\mu h_i = \sum_{j=1}^N w_{ij} \sigma_i^\mu \sigma_j^\mu > 0 \tag{3.2.29}$$

for every neuron i . These quantities are sometimes called *stability coefficients* and are denoted as $\tilde{\gamma}_i^\mu := \sigma_i^\mu h_i$. If the previous expression is only slightly positive, any small perturbation $s_i(t) \neq \sigma_i^\mu$ for a few neurons can change its sign. But our goal is not only to provide a network which is stable, but also being able to correct eventual small corruptions (i.e. converging to the stored patterns). Then, in order to achieve greater stability of the desired memory patterns, we demand that the expression (3.2.29) be not only positive but also greater than a certain threshold $\kappa > 0$. For the single-pattern case, we have seen previously that the simplest form of the Hebb's rule

$$w_{ij} = \frac{1}{N} \sigma_i \sigma_j \tag{3.2.30}$$

yields $h_i = \sigma_i$ (3.2.13). As a consequence, the condition $\sigma_i h_i = 1$ was always satisfied in that case. Therefore it appears natural to take the stability threshold of the order of unit ($\kappa = 1$) also in the general case of several stored patterns, and to demand that the synaptic

connections w_{ij} be chosen such that

$$\sigma_i^\mu h_i = \sum_{j=1}^N w_{ij} \sigma_i^\mu \sigma_j^\mu = 1 \quad (3.2.31)$$

exactly for every neuron i .

An obvious method of achieving the desired result begins with choosing the synaptic connections initially according to the general Hebb's rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \quad (3.2.32)$$

In the next step we check, one after the other for all stored patterns, whether the condition (3.2.31) is fulfilled. If this is not the case, we modify the synapses according to the prescription

$$w_{ij} \rightarrow w'_{ij} = w_{ij} + \delta w_{ij} \quad (3.2.33)$$

with

$$\delta w_{ij} = \frac{1}{N} (1 - \sigma_i^\mu h_i) \sigma_i^\mu \sigma_j^\mu \quad (3.2.34)$$

where μ denotes the pattern just under consideration in the stability coefficient³. With these modified synaptic connections we obtain for the same pattern μ

$$\begin{aligned} \sigma_i^\mu h'_i &= \sum_{j=1}^N w'_{ij} \sigma_i^\mu \sigma_j^\mu \\ &= \sigma_i^\mu h_i + \sum_{j=1}^N \delta w_{ij} \sigma_i^\mu \sigma_j^\mu \\ &= \sigma_i^\mu h_i + \frac{1}{N} \sum_{j=1}^N (\sigma_i^\mu)^2 (\sigma_j^\mu)^2 (1 - \sigma_i^\mu h_i) \\ &= \sigma_i^\mu h_i + 1 - \sigma_i^\mu h_i \\ &= 1 \end{aligned} \quad (3.2.35)$$

since $(\sigma_i^\mu)^2 = 1$. Thus, after updating all synapses, the threshold stability condition (3.2.31) is satisfied for the considered pattern. When we proceed to the next pattern ($\mu + 1$), the synaptic couplings (already modified) will be modified again, so that (3.2.31) becomes valid for the pattern now under consideration. However, (3.2.31) may cease to be satisfied for the previous pattern μ . After a full cycle over all stored patterns the condition is therefore only fulfilled with certainty for the last pattern, $\mu = p$, but not necessarily for all other patterns. The crucial question is whether this updating process converges, or whether it may continue indefinitely without reaching a stationary state, in which the threshold condition is satisfied by all patterns.

Without going too far into the demonstration, it is possible to show that the iteration process is guaranteed to converge, yielding the synaptic connections

$$w_{ij} \rightarrow \bar{w}_{ij} = \frac{1}{N} \sum_{\mu,\nu} \sigma_i^\mu (Q^{-1})_{\mu\nu} \sigma_j^\nu \quad (3.2.36)$$

³In principle, we can do without the initial Hebbian choice of synapses. If we start with a completely disconnected network ($w_{ij} = 0$), or *tabula rasa*, the first application of the modification law (3.2.33) results in synaptic connections with precisely the values assigned by Hebb's rule!

3 Neural networks

But these are precisely the synaptic connections (3.2.27), \tilde{w}_{ij} , of the projection rule discussed at the beginning of this section, which solve the problem of storing correlated patterns! The only difference is that now, in the Diederich and Opper iterative procedure, we don't have to perform any inversion of the matrix.

3.2.4 Spin glasses

By denoting the two possible states of a neuron by the variables $s_i = +1$ (active neuron) and $s_i = -1$ (resting neuron), we have already exposed the analogy between a neural network and a system of atomic magnetic dipoles or spins, which can be oriented in two different directions. We shall now show how this analogy can be advantageously exploited in view of the remarkable progress made in studies of the physical properties of magnetic systems.

Atoms interact with each other by inducing a magnetic field at the location of another atom, which interacts with its spin. The total local magnetic field at the location of an atom i is given by something of the form $h_i = \sum_{j \neq i} w_{ij} s_j$, where w_{ij} is the dipole force, and the diagonal term $j = i$ (self-energy) is not included in the sum over j . Newton's law "action = reaction" ensures that the coupling strengths w_{ij} are symmetric: $w_{ij} = w_{ji}$. If all w_{ij} are positive, the material is *ferromagnetic*; if there is a regular change of sign between neighboring atoms, one has an *antiferromagnet*. If the signs and absolute values of the w_{ij} are distributed randomly, the material is called a **spin glass**⁴. The ferromagnetic case corresponds to a neural network that has stored a single pattern. The network which has been loaded with a large number of randomly composed patterns resembles a spin glass.

In order to describe the properties of a spin system it is useful to introduce an *energy functional* $E[s]$:

$$E[s] = -\frac{1}{2} \sum_{i,j, i \neq j} w_{ij} s_i s_j \quad (3.2.37)$$

which indicates the total energy of a certain configuration $[s] \equiv \{s_i\}$. This is always possible for symmetric couplings $w_{ij} = w_{ji}$; any antisymmetric contribution would cancel in the double sum. As usual, the one-half factor in front is necessary to prevent double counting of the terms corresponding to the same couples i, j . In principle we should exclude diagonal terms $i = j$ in (3.2.37) because they represent the self-energy contributions, but practically we can include them without prejudice since these contribute only a constant term to the energy, which does not depend on the orientation of the spins (or on the states of the neurons) because $(s_i)^2 = (\pm 1)^2 = 1: \frac{1}{2} \sum_i w_{ij}$. This additional constant does not influence the dynamical evolution of the system and can be dropped.

If the couplings w_{ij} are determined according to Hebb's rule (3.2.18), which surely ensure symmetry (Hebb's rule always leads to symmetric synaptic couplings, which is no longer true for more general learning rules), and if the configuration s_i of the system corresponds to a stored pattern σ_i^ν , the energy functional takes the value

$$E[\sigma^\nu] = -\frac{1}{2N} \sum_{i,j} \sum_\mu \sigma_i^\mu \sigma_j^\mu \sigma_i^\nu \sigma_j^\nu = -\frac{1}{2N} \left[N^2 + \sum_{\mu \neq \nu} \left(\sum_{i=1}^N \sigma_i^\mu \sigma_i^\nu \right)^2 \right] \quad (3.2.38)$$

Assuming again that the patterns are uncorrelated and the values $\sigma_i^\mu = \pm 1$ are equally likely, one finds that the term in round brackets is of order \sqrt{N} , and therefore the energy

⁴The term "glass" comes from an analogy between the magnetic disorder in a spin glass and the positional disorder of a conventional, chemical glass, e.g., a window glass. In window glass or any amorphous solid the atomic bond structure is highly irregular; in contrast, a crystal has a uniform pattern of atomic bonds.

becomes

$$E[\sigma^\nu] \simeq -\frac{1}{2N} \left[N^2 + O((p-1)N) \right] = -\frac{N}{2} + O\left(\frac{p-1}{2}\right) \quad (3.2.39)$$

This is the energy in the case in which the network presents one of the stored pattern. What happens now if the initial configuration of the net is not exactly equal to one of the stored patterns, but presents small deviations (notice that here we are again studying the two situations considered in Section 3.2.2)? If the spins of a few atoms have the wrong orientation, the fluctuating term is replaced by another fluctuating term, which does not result in an essential modification of its magnitude (is still a sum of random variables). However, the term with $\mu = \nu$ (i.e. the first one) suffers a substantial change: if the first n spins are incorrectly oriented:

$$s_i(t=0) = \begin{cases} -\sigma_i^\nu, & \text{for } i = 1, \dots, n \\ +\sigma_i^\nu, & \text{for } i = n+1, \dots, N \end{cases} \quad (3.2.40)$$

its value changes to

$$-\frac{1}{2N} \sum_{i,j} \sigma_i^\nu \sigma_j^\nu s_i s_j = -\frac{1}{2N} \left(\sum_i \sigma_i^\nu s_i \right)^2 = -\frac{1}{2N} (N-2n)^2 \quad (3.2.41)$$

because the index is dummy. Accordingly, the energy of the configuration becomes

$$E[s] \simeq -\frac{1}{2N} (N-2n)^2 + O\left(\frac{p-1}{2}\right) = E[\sigma^\nu] + 2n - \frac{2n^2}{N} \quad (3.2.42)$$

What have we just discovered from this analysis? We have demonstrated that $E[s] > E[\sigma^\nu]$, that is the memorized patterns are therefore (at least local) *minima* of the energy functional. Using Hebb's rule, the energy reaches a minimum in presence of a stored pattern.

A more detailed investigation shows that the energy functional has, in addition, infinitely many other local minima. However, all these spurious minima are less pronounced than those given by the stored patterns σ_i^μ ; hence these (the stored ones) correspond to *absolute* minima of the energy surface $E[s]$, at least for moderate values of the parameter p/N . However, this is not a good situation, because the network systematically tends to lower the energy (see below), and therefore we risk being trapped in local minima depending on the data flow (this procedure doesn't allow to jump over). Thus we risk of not being able to reach the true ground state because we get stuck in "false" minima.

Usually, these false minima are linear combinations of the stored patterns. Imagine, for instance, that our memorized patterns are the letters of the alphabet (Fig. 3.7). Clearly, a combination of the bits of the first figure with those of the other would lead to a pattern which is not one of the stored ones, i.e. a letter of the alphabet, but still a local minima.

3.2.4.1 Parallel versus sequential dynamics

The energy function of a physical system is closely related to its dynamics. It is easy to see that the energy function $E[s]$ continually decreases with time in the case of (random) *sequential* updating of the neuron states (the Hopfield model). Since we are now considering a sequential updating, it is sufficient to focus only on $E_i(t)$, the contribution of a given neuron i to the energy at time t , rather than over the entire energy $E[s]$ because at each instant only one neuron changes its energy. In particular, at time t the energy contribution

3 Neural networks

of neuron i is given by

$$E_i(t) = -s_i(t) \left[\sum_{j \neq i} w_{ij} s_j(t) \right] \equiv -s_i(t) h_i(t) \quad (3.2.43)$$

If the state of the neuron i is updated according to the law (3.2.10), its new contribution to the energy becomes

$$\begin{aligned} E_i(t+1) &= -s_i(t+1) \left[\sum_{j \neq i} w_{ij} s_j(t) \right] \\ &= -\operatorname{sgn}[h_i(t)] h_i(t) \\ &= -|h_i(t)| \\ &\leq -s_i(t) h_i(t) \\ &= E_i(t) \end{aligned} \quad (3.2.44)$$

Notice that in the first line we didn't modified the time of s_j inside the summation. In fact, in the Hopfield model we have kept fixed all neurons different from i , and those neurons (whose values at time $t+1$ have been determined at the previous step t) interact with the i -th neuron, which has now been updated. Note also that $|h_i(t)| \geq s_i(t)h_i(t)$ because on the right-hand side h_i is multiplied by a unitary factor s_i ; hence it can be either equal to $+h_i$ or to $-h_i$, which surely are smaller than the absolute value.

In conclusion, we have found that the energy contribution of a given neuron never increases with time. Since the energy functional $E[s]$ is bounded below, this implies that the network dynamics must reach a stationary point that corresponds to a (local) minimum of the energy functional. The network thus always ends up in a state of *equilibrium*, which is stable against changes in the state of any single neuron. But this, as mentioned before, is what we would like to avoid, in order not to get trapped in false minima.

The argument given above does not apply to the *synchronous* mode of operation of the neural network (the Little model). Since all neurons assume new states in parallel and at the same moment, the contribution of an individual neuron to the energy function cannot be considered in isolation. In this case it is more appropriate to consider the **Lyapunov stability function**

$$E_L(t) = - \sum_{i,j} w_{ij} s_i(t) s_j(t-1) \quad (3.2.45)$$

which depends on the state of the entire network at two subsequent moments of its dynamical evolution. $E_L[s(t), s(t-1)]$ cannot be regarded as an energy functional, since it depends on the states of the neurons at two different times, i.e. it is non-local in the time variable. However, for symmetric couplings $w_{ij} = w_{ji}$ it is again easy to show that $E_L(t)$ decreases

monotonously with time:

$$\begin{aligned}
 E_L(t+1) &= - \sum_{i,j} w_{ij} s_i(t+1) s_j(t) \\
 &= - \sum_i s_i(t+1) \sum_j w_{ij} s_j(t) \\
 &= - \sum_i \text{sgn}[h_i(t)] h_i(t) \\
 &= - \sum_i |h_i(t)| \\
 &\leq - \sum_i h_i(t) s_i(t-1) \\
 &= E_L(t)
 \end{aligned} \tag{3.2.46}$$

Furthermore, from the definition of Lyapunov function we can write that

$$E_L(t+1) - E_L(t) = - \sum_{i,j} w_{ij} s_i(t) [s_j(t+1) - s_j(t-1)] \tag{3.2.47}$$

Therefore, the network can eventually settle into a steady state ($E_L(t+1) - E_L(t) = 0$) in which the same networks configuration is repeated every other time step: $s_j(t+1) = s_j(t-1)$. In other words, if $s_j(t+1) - s_j(t-1) = 0$, then the net reaches a stable configuration because the terms in the summation cancel each other. In summary, this means that the network can reach a steady state (equilibrium) in two ways: either as in the Hopfield model through a “spontaneous” minimization, or permanently cycling between two different configurations.

The property of admitting cycling behavior as opposed to the unavoidable approach to a steady state is an attractive aspect of the parallel synchronous updating mode, especially if one wants to associate “trains of thought” with such reverberations of the neural network. Of course, symmetrical synaptic weights lead only to trivial cycling patterns (two-cycles), but networks with asymmetric synapses can exhibit very complex cycling behavior, with different cycle lengths possibly prevailing in separate regions of a large neural network.

3.2.5 Neural “motion pictures”

What we have seen so far is certainly a remarkable development of the theory. But we would like to create a system (a network) that works in a more sophisticated way than a simple hopping between two states. Is there any way to improve this situation? In this regard, let's think about the way the brain works. The brain not only recognizes a pattern, but also temporal sequences or chain of thoughts. Temporal sequences of recalled memories form a very important aspect of our mental capacity. They enable us to comprehend the course of events and actions, which otherwise would consist of meaningless single pictures. Periodic sequences of neural impulses are also of fundamental importance. The questions how neural networks can sustain highly periodic activity for a long period of time and what makes them fail under certain conditions are thus of vital interest.

It is clear from the very beginning that here we have to study the properties of networks with asymmetric synaptic connections, because periodic activity cannot occur in the presence of thermodynamic equilibrium, toward which all symmetric networks develop. It turns out that the transition from a Hopfield–Little network with its stable-equilibrium configurations

3 Neural networks

to a network exhibiting stationary periodic activity is surprisingly simple. One only has to modify Hebb's rule (3.2.18) in the following manner:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^{\mu+1} \sigma_j^\mu \quad (3.2.48)$$

where the patterns σ_i^{p+1} and σ_i^1 are identified in order to get temporal periodicity. In order to see how this synaptic rule works, we assume that the network configuration is just given by one of the stored patterns, $s_i(t) = \sigma_i^\nu$. Assuming that the patterns are orthogonal, i.e. $\sigma_i^\mu \sigma_i^\nu = \delta_{\mu\nu} \forall i$, the synaptic polarization potential for the next update is

$$h_i^{(\nu)}(t) = \sum_{j=1}^N w_{ij} \sigma_j^\nu = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^{\mu+1} \sum_{j=1}^N \sigma_j^\mu \sigma_j^\nu = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^{\mu+1} N \delta_{\mu\nu} = \sigma_i^{\nu+1} \quad (3.2.49)$$

As a consequence the network immediately makes a transition into the next pattern:

$$s_i(t+1) = \text{sgn} [h_i^{(\nu)}(t)] = \text{sgn} (\sigma_i^{\nu+1}) = \sigma_i^{\nu+1} \quad (3.2.50)$$

i.e. it performs “heteroassociation” instead of autoassociation. Note that this mechanism works only in the synchronous, parallel updating mode (since we update all neurons at the same time to produce a specific pattern).

This rapid, undelayed transition between patterns is not always desired. In the biological context it would imply that one activation pattern would replace another on the scale of the elementary time constant of neural processes, i.e. patterns would change every few milliseconds. This is much too short for many processes, such as motor actions or chains of thoughts, where the characteristic time constants lie in the range of tenths of seconds. How is it possible to stretch the temporal separation between different patterns, to obtain a kind of “slow-motion” effect?

One way to achieve this delay is to add a stabilizing term to the expression (3.2.48) for the synaptic connections. In particular, we have seen before many times that the original Hebb's rule (3.2.18) has the property that an initial configuration equal to a stored pattern produces a stable configuration $s_i(t+1) = s_i(t)$. This is exactly for us. So, why not trying to combine these two choices of the couplings, i.e. Hebb's rule and (3.2.48), in order to control the way in which the evolution of the network takes place? Hence we introduce different types of synaptic connections characterized by different lengths of their relaxation time. We denote the rapidly relaxing, stabilizing (“fast”) synapses as

$$w_{ij}^F = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \quad (3.2.51)$$

and the slowly relaxing, change-inducing (“slow”) synapses by

$$w_{ij}^S = \frac{\lambda}{N} \sum_{\mu=1}^p \sigma_i^{\mu+1} \sigma_j^\mu \quad (3.2.52)$$

where λ is a constant that characterizes the preponderance of one mode over the other. Then we define the total post-synaptic polarization potential at time t in the following way:

$$h_i(t) = \sum_{j=1}^N [w_{ij}^F s_j(t) + w_{ij}^S \bar{s}_j(t)] \quad (3.2.53)$$

with

$$\bar{s}_j(t) = \int_{-\infty}^t G(t-t') s_j(t') dt' \quad (3.2.54)$$

Here $G(t)$ plays the role of a delay function (like a causal Green function), which is normalized according to $\int_0^\infty G(t) dt = 1$. Standard choices for this function are $G(t) = \delta(t - \tau)$, representing a definite well-defined time delay, and $G(t) = \tau^{-1} e^{-\frac{t}{\tau}}$, describing a gradual decay of the post-synaptic potential with a characteristic time constant τ .

3.2.6 Stochastic neurons

We now consider a simple generalization of the neural networks discussed in the previous sections which permits a more powerful theoretical treatment. For this purpose we replace the deterministic evolution law

$$s_i(t+1) = \text{sgn}[h_i(t)] = \text{sgn}\left[\sum_{j=1}^N w_{ij} s_j(t)\right] \quad (3.2.55)$$

for the neural activity by a **stochastic law**, which does not assign a definite value to $s_i(t+1)$, but only gives the probabilities that $s_i(t+1)$ takes one of the values $+1$ or -1 . In particular we request that

$$\Pr[s_i(t+1) = +1] = f[h_i(t)] \quad (3.2.56)$$

where the activation function $f(h)$ must have the proper limiting values $f(h \rightarrow -\infty) = 0$, $f(h \rightarrow +\infty) = 1$. Between these limits the activation function must rise monotonously. A standard choice, depicted in Figure 3.9, is given by

$$f(h) = \frac{1}{1 + e^{-2\beta h}} \quad (3.2.57)$$

which satisfies the condition

$$f(h) + f(-h) = 1 \quad (3.2.58)$$

Accordingly, since $\Pr[s_i(t+1) = -1] = 1 - f[h_i(t)]$, we can explicit our stochastic law as

$$\Pr[s_i(t+1) = +1] = f[+h_i(t)] \quad (3.2.59)$$

$$\Pr[s_i(t+1) = -1] = f[-h_i(t)] \quad (3.2.60)$$

that is the values $s_i(t+1) = \pm 1$ will occur with probability $f(\pm h_i)$.

The shape of the function $f(h)$ is essentially that of a Fermi–Dirac distribution^{5,6}, which describes the thermal energy distribution in a system of identical fermions. In that case the parameter β has the meaning of an inverse temperature, $\beta = T^{-1}$ (imagine the thermodynamic beta $\beta = 1/(k_B T)$ in natural units). We shall also use this nomenclature in connection with neural networks, although this does not imply that the parameter β^{-1} should denote a physical temperature at which the network operates. The rule (3.2.57) should rather be considered as a model for a stochastically operating network which has been conveniently designed to permit application of the powerful formal methods of statistical physics. In other words, β^{-1} is essentially a pseudo-temperature, i.e. an artificial parameter that has been called “temperature” in analogy with statistical physics, but that has a

⁵The step is now placed at $h = 0$ and unlike the Fermi function it grows monotonously.

⁶Such functions are often called *sigmoidal* functions.

3 Neural networks

completely different meaning. Notice that in the limit $\beta \rightarrow \infty$, or $T \rightarrow 0$, the Fermi function (3.2.57) approaches the unit step function $\Theta(h)$. This means that at very low temperature the stochastic neural network goes over into our original deterministic network (3.2.55). Hence all results obtained for stochastic networks can be extrapolated to the deterministic case.

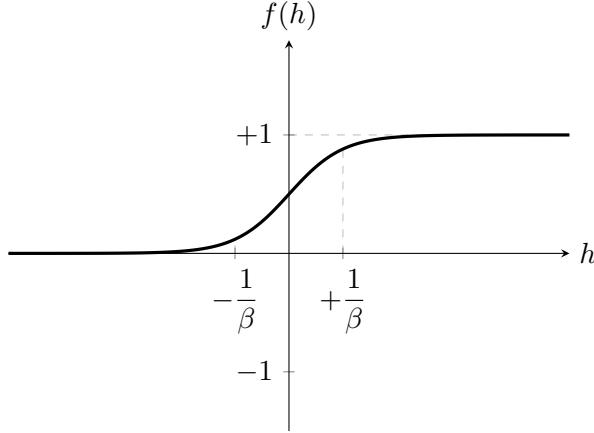


Figure 3.9

The exact state of a given neuron in a stochastic network is not of particular relevance, because it is determined randomly. However, the *mean activity* of a neuron (or the mean orientation of a spin in the magnetic analogy) is an interesting quantity. Let us first consider a single neuron; its mean activity is given by

$$\begin{aligned}
\langle s \rangle &= (+1)f(h) + (-1)f(-h) \\
&= \frac{1}{1 + e^{-2\beta h}} - \frac{1}{1 + e^{2\beta h}} \\
&= \frac{1}{1 + e^{-2\beta h}} - \frac{e^{-2\beta h}}{1 + e^{-2\beta h}} \\
&= \frac{1 - e^{-2\beta h}}{1 + e^{-2\beta h}} \\
&= \frac{e^{\beta h} - e^{-\beta h}}{e^{\beta h} + e^{-\beta h}} \\
&= \tanh(\beta h)
\end{aligned} \tag{3.2.61}$$

As already said, in the limit $\beta \rightarrow \infty$ one obtains $\langle s \rangle = \text{sgn}(h)$, i.e., depending on the sign of the local field h , the neuron is either permanently active or permanently dormant (again, we recover the deterministic result).

The evolution of a single neuron i in a network composed of many elements is difficult to describe, since its state depends on *all* other neurons, which continually fluctuate between $+1$ and -1 . That even remains true if we are only interested in the mean activity of the i -th neuron. This is determined by the value of the synaptic potential h_i , which depends, however, on the actual instantaneous states s_j of the other neurons and not on their mean activities. The difficulty is a result of the non-linearity of the probability function $f(h)$, which does not permit one to take the average in the argument of the function. In such cases one often resorts to a technique called the **mean-field approximation**.

We have already seen the mean-field approximation in many contexts: in solid state physics, statistical physics and nuclear physics also. In those cases we used to study

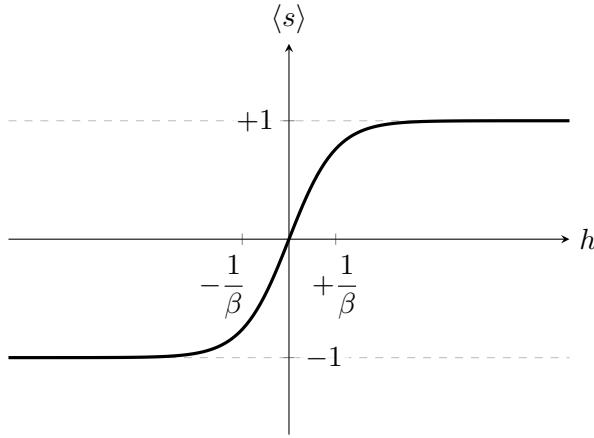


Figure 3.10: The function $\langle s \rangle = \tanh(\beta h)$.

many-body problems (for example systems of interacting fermions) of which we were given the total Lagrangian. By solving the Euler–Lagrange equations, one is able to construct a system of (often coupled) equations which then had to be solved to determine the evolution of the individual fields. The problem arises from the fact that usually we work in a quantum regime, and not in a classical one; so the fields to be determined are actually operators, and not numbers. An operator is a more complicated object because it contains a lot of information, although one usually takes the expectation value to get some measure of probability. Consequently, the interaction terms within the equations of motion become almost unmanageable for many-body problems, as each field depends on the local value of all the other fields. The idea of the mean-field approximation is therefore to systematically reduce quantum fields into classical fields, that is to see them no longer as operators, but as numbers. In practice, instead of quantum fields, their expectation value was substituted: $\hat{\psi} \rightarrow \langle \hat{\psi} \rangle$. In this way the many-body problem is simplified in a one-body problem, because the total intricate interaction is replaced with a *mean field* generated by all the others (i.e. no more many-body-dependent). In other words, this technique allows us to factor out the terms of interactions and to obtain, by adding them, a single interaction field that describes the average potential generated by all the particles together. The mean-field approximation becomes particularly useful when we have to deal simultaneously with many bodies (indeed, it is sometimes called large- N approximation).

Returning to our neural networks, here we basically want to do the same thing, and therefore we can define the mean-field approximation as the exchange operation between the synaptic potential with its average value

$$f(h_i) \xrightarrow{\text{MFA}} f(\langle h_i \rangle) = f\left(\sum_{j=1}^N w_{ij} \langle s_j \rangle\right) \quad (3.2.62)$$

The mean activity of a neuron is then computable as before:

$$\langle s_i \rangle = (+1)f(\langle h_i \rangle) + (-1)f(-\langle h_i \rangle) \quad (3.2.63)$$

whence

$$\langle s_i \rangle = \tanh\left(\beta \sum_{j=1}^N w_{ij} \langle s_j \rangle\right)$$

(3.2.64)

3 Neural networks

This is still a system of nonlinear equations with N unknowns $\langle s_i \rangle$, but these have become deterministic rather than stochastic variables.

3.2.6.1 Single pattern

The solution of the system of equations (3.2.64) depends, of course, on the choice of the synaptic strengths w_{ij} . We begin by considering the simplest case of a single pattern

$$w_{ij} = \frac{1}{N} \sigma_i \sigma_j \quad (3.2.65)$$

Assuming also that all bits are equal (+1 or -1, it doesn't matter) we can further simplify it as $w_{ij} = 1/N$ for all i, j . One could be skeptical about this assumption, but it is essentially a gauge transformation, since it corresponds to a reinterpretation of the meaning of "up" and "down" at each lattice site.

Anyway, in this case equation (3.2.64) becomes

$$\langle s_i \rangle = \tanh \left(\beta \frac{1}{N} \sum_{j=1}^N \langle s_j \rangle \right) \quad \text{for all } i \quad (3.2.66)$$

and because the right-hand side does not depend on i we can write

$$\langle s \rangle = \tanh (\beta \langle s \rangle) \quad (3.2.67)$$

(of course $\langle s_i \rangle = \langle s_j \rangle$ since we don't have any reason to suppose they are different). We have thus managed to reduce the evolution law to a single equation, but it is still a transcendental equation; therefore we must solve it graphically, i.e. we have to find graphically the possible intersections between the straight line $\langle s \rangle$ and the hyperbolic tangent $\tanh(\beta \langle s \rangle)$. Recall that the hyperbolic tangent is monotonously increasing and from 0 to ∞ its slope always diminishes (vice versa, from $-\infty$ to 0 it systematically increases, due to symmetry reasons). Based on this observation, we deduce that we have to compare the behavior of the two functions at the origin, where the slope has its maximum value. In particular, we have to distinguish two cases illustrated in Figure 3.11. For $\beta < 1$ ($T > 1$) equation (3.2.67) admits only the trivial solution at $\langle s \rangle = 0$ because the slope of the line is higher than that of the hyperbolic tangent (Fig. 3.11a). So at large temperature the bits continuously flip between +1 and -1, but with null expectation value, meaning that the network is essentially unable to memorize any information. Instead, for $\beta > 1$ ($T < 1$) equation (3.2.67) admits altogether three solutions, namely $\langle s \rangle = -\bar{s}, 0, \bar{s}$ because the slope of the line is smaller than that of the hyperbolic tangent (Fig. 3.11b). However, it is possible to show that $\langle s \rangle = \pm \bar{s}$ correspond to a stable minimum configurations of the system, while $\langle s \rangle = 0$ corresponds to a local maximum, which is unstable! In fact, the hyperbolic tangent is very stiff close to 0; hence any small perturbation from $\langle s \rangle = 0$ makes the system change rapidly either towards -1 or +1. Therefore, we neglect this possibility at low temperature. Finally, the limiting situation is that in which the straight line is exactly tangent to the hyperbolic tangent, corresponding to $\beta = T = 1$ (which takes the role of critical, or Curie, temperature separating the two different regimes). This analytical behavior at $T = 1$ is indeed very similar to a *phase transition* in which at high temperature the system is chaotic, whereas at low temperature it presents an oriented structure (like ferromagnetism).

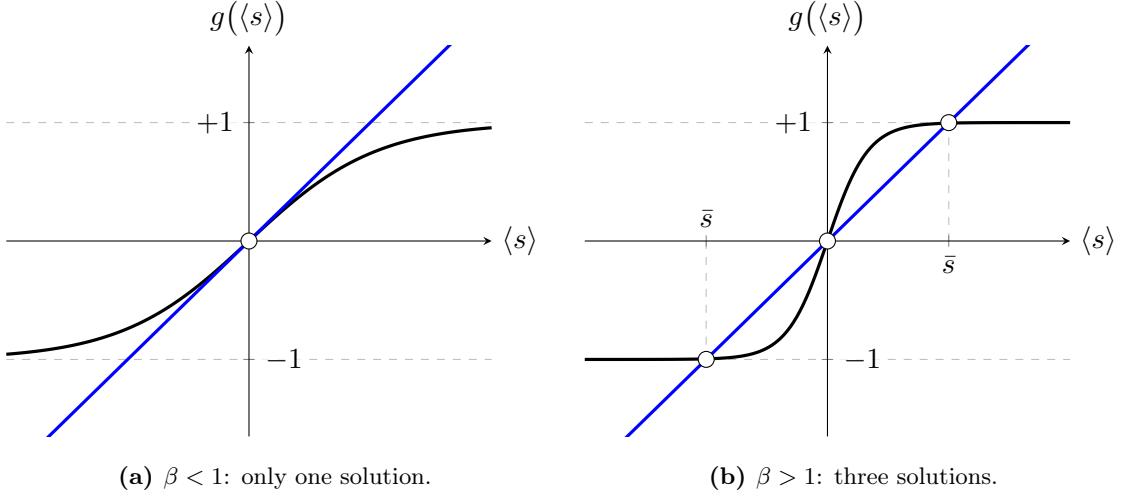


Figure 3.11: Intersections between $\langle s \rangle$ and $\tanh(\beta\langle s \rangle)$.

3.2.6.2 Several patterns

We now turn to the general situation, when the synaptic couplings are fixed according to Hebb's rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \quad (3.2.68)$$

for several patterns. The mean-field equation (3.2.64) then takes the form

$$\langle s_i \rangle = \tanh \left(\frac{\beta}{N} \sum_{j=1}^N \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \langle s_j \rangle \right) \quad (3.2.69)$$

This relation does not have an obvious solution. However, we recall that every single stored pattern represents a stable configuration for a deterministic network. It is, therefore, not unreasonable to make the *ansatz*⁷ that $\langle s_i \rangle$ resembles one of the stored patterns, except for a normalization factor:

$$\langle s_i \rangle = m \sigma_i^\nu \quad (3.2.70)$$

Inserting this into (3.2.69) and assuming as usual that the memorized patterns are completely uncorrelated we obtain

$$\begin{aligned} m \sigma_i^\nu &= \tanh \left(\frac{\beta m}{N} \sum_{j=1}^N \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \sigma_j^\nu \right) \\ &= \tanh \left(\frac{\beta m}{N} \sum_{j=1}^N \sigma_i^\nu \overbrace{\sigma_j^\nu \sigma_j^\nu}^{=1} + \frac{\beta m}{N} \sum_{\mu \neq \nu} \sigma_i^\mu \sum_{j=1}^N \sigma_j^\mu \sigma_j^\nu \right) \\ &= \tanh \left[\beta m \sigma_i^\nu + \beta m O \left(\sqrt{\frac{p-1}{N}} \right) \right] \end{aligned} \quad (3.2.71)$$

As long as $p \ll N$ the second term is clearly negligible:

$$m \sigma_i^\nu \simeq \tanh(\beta m \sigma_i^\nu) \quad (3.2.72)$$

⁷An ansatz (from German) is an educated guess or an additional assumption made to help solve a problem, and which is later verified to be part of the solution by its results.

3 Neural networks

Moreover, on account of $\sigma_i^\nu = \pm 1$, i.e. $\tanh(-x) = -\tanh x$ (it is an odd function), the normalization factor m (and consequently $\langle s_i \rangle$) is determined by the same equation

$$m = \tanh(\beta m) \quad (3.2.73)$$

Again we find the same fixed points as for a single stored pattern: for $T > 1$ one has $m = 0$, and the time-averaged network configuration does not resemble one of the stored patterns. One might say that the network is “amnesic”. For $T < 1$ one has $m = \pm \bar{s}$ and the average configuration of the network points toward one of the stored patterns. Since these take only the values ± 1 , the patterns can be uniquely (up to an overall sign) recovered from the averaged state $\langle s_i \rangle$ of the network.

Numerical simulations as well as statistical analysis show that the number of stored patterns relative to the number of neurons, $\alpha = p/N$, plays a similar role as the parameter T . When the storage utilization α is increased starting from zero, i.e. if more and more patterns are stored, the recall quality of the network deteriorates slightly. However, when α approaches the critical capacity $\alpha_c \approx 0.138$, the network suddenly and catastrophically fails to recall any of the memorized patterns. In other words the networks suddenly jumps from almost perfect memory into a state of complete confusion.

In order to obtain a complete description of the ability of a stochastic Hopfield network to recall memorized patterns one has to consider m as function of α as well as of T . One then obtains a complete phase diagram of the *order parameter* $m(T, \alpha)$. The phase boundary between “functioning memory” and total “confusion” or “amnesia” is given by a line in the α - T diagram, depicted in Figure 3.12. For $T = 0$ this line begins at $\alpha_c \approx 0.138$ and ends, as we just have found, at $T = 1$ for $\alpha = 0$. At the phase boundary m falls discontinuously to zero, except for $T = 0$, where the change is continuous, but not differentiable⁸.

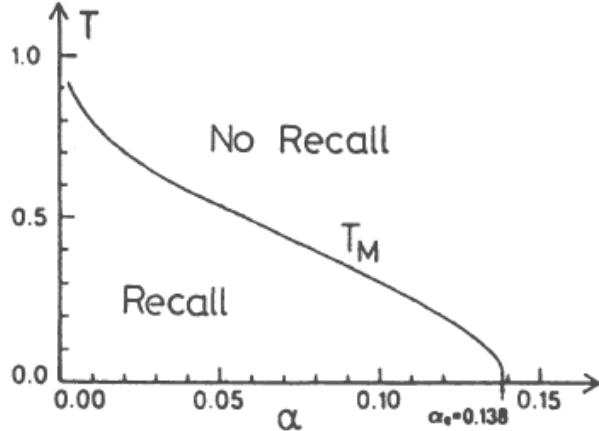


Figure 3.12: Memory recall is possible only in a finite region of temperatures T and storage density $\alpha = p/N$.

In addition to its usefulness in the formal treatment of neural-network properties the introduction of a stochastic neuron evolution law has important practical advantages. The thermal fluctuations reduce the probability that the network becomes caught in a spurious, undesired locally stable configuration.

⁸Note that these results strictly apply only in the thermodynamic limit $N \rightarrow \infty$, i.e. for systems with infinitely many neurons.

3.2.7 Special learning rules

As we discussed in previous sections, the standard learning rule (Hebb's rule) leads to the emergence of undesirable local minima in the “energy” functional $E[s]$. In practice, this means that the evolution of the network can be caught in spurious, locally stable configurations. Large networks usually contain a vast number of such spuriously stable states, many of which are not even linear combinations of the desired stability points. Even the learning rule for correlated patterns discussed in Section 3.2.3 does not guard against this problem. Thermal fluctuations do help to destabilize the spurious configurations, but at the expense of storage capacity. Moreover, the disappearance of all the spurious states at some finite T is not ensured.

A much better strategy is to eliminate the undesired stable configurations by appropriate modifications of the synaptic connections. Hopfield et al. [11] have proposed to make use of the fact that the spurious minima of the energy functional $E[s]$ are usually much shallower than the minima that correspond to the learned patterns. Borrowing ideas developed in the study of human dream sleep⁹, they suggested tracking these states by starting the network in some randomly chosen initial configuration and running it until it ends up in a stable equilibrium state s_i^∞ . This may be one of the regular learned patterns, or one of the many spurious states. Then, we build new synaptic connections according to the new Hebb's rule:

$$w_{ij} \rightarrow w_{ij} - \frac{\lambda}{N} s_i^\infty s_j^\infty \quad (3.2.74)$$

where $\lambda \ll 1$ is chosen (indeed we are weakening connections), and we let the network run again, until it reaches another equilibrium state s_i^∞ . After we build another w_{ij} and we let evolve, and so on and so forth. This iterative procedure must be repeated a large number of times, until we actually fall in one of the global minima. Note that whatever the resulting state is in each iteration, through (3.2.74) the synapses are partially weakened at each step. This procedure of unlearning has two favorable effects. Most spurious equilibrium states of the network are “forgotten”, since they are already destabilized by small changes in the synaptic connections w_{ij} . Moreover, the different regions of stability of the stored patterns become more homogeneous in size, since those with a larger range of stability occur more often as final configurations and are therefore weakened more than others.

The effect of this intentional forgetting is especially apparent in the sizes of the “basins of attraction”. This term denotes the set of all states, from which the network dynamics leads to a particular pattern (e.g. σ^μ). The change of the size of the basin of attraction of a given stored pattern, as the total memory load $\alpha = p/N$ is increased, is illustrated in Figure 3.13. The two axes labeled H_k and H_{N-k} in these figures represent a crude measure of the distance of an initial trial state s_i from the considered memory state σ_i^μ . They denote the partial Hamming distances between the trial state and the memory state, evaluated for

⁹The sleep is commonly divided in two phases, referred to as REM (rapid eye movement) and non-REMs, which is a collection of other stages. The REM phase is characterized by random rapid movement of the eyes, accompanied by low muscle tone throughout the body, and the propensity of the sleeper to dream vividly. Modern studies believe that the purpose of dream sleep (REM sleep) is the elimination of undesirable states of memory.

3 Neural networks

the first k and the last $N - k$ of all $N = 200$ neurons (for instance), respectively:

$$H_k = \frac{1}{4} \sum_{i=1}^k (s_i - \sigma_i^\mu)^2 \quad (3.2.75)$$

$$H_{N-k} = \frac{1}{4} \sum_{i=N-k+1}^N (s_i - \sigma_i^\mu)^2 \quad (3.2.76)$$

In this specific case $k = N/2$ was taken, i.e. the axes represent the Hamming distance for the first and the last half of the neurons of the network. If the trial state s_i developed into the stored pattern σ_i^μ , a black dot was plotted.

For a single memory state (Fig. 3.13a), half of the trial states are found to evolve into the stored pattern σ_i , while the other half ends up in the complementary pattern $-\sigma_i$. The basin of attraction thus represents a black triangle. This is essentially the result that we have discussed in Section 3.2.2: if we have only one pattern, a random configuration of the network with some altered bits converges to the stored pattern if the number of wrong bits n is smaller than $N/2$. And indeed, any set of s_i (i.e. any different trial configuration) which produces a couple (H_k, H_{N-k}) inside the black triangle satisfies this constraint, thus converging to the stored pattern σ_i . On the other hand, if those trial states are outside the triangle, meaning they are nearer to the complementary pattern $-\sigma_i$ than σ_i , then the net will converge to $-\sigma_i$ (which is not what we want). Just to make things clearer, if we take a trial state giving $(H_k, H_{N-k}) = (100, 0)$, then the network configuration will develop into the memorized pattern because it means that half of the initial bits are exact ($H_{N-k} = 0$), while the other half are wrong ($H_k = 100$).

What now happens if we increase the number of stored patterns? For more memory states the basin of attraction shrinks rapidly and takes on a highly ragged shape in the vicinity of $\alpha_c \approx 0.138$, as shown in Figures 3.13b, c for 28 and 32 uncorrelated memory states, respectively. In fact, when we are above the critical storage threshold α_c the network becomes no more able to retrieve memorized patterns.

Finally Figure 3.13d shows the result of applying the forgetting algorithm (3.2.74) 1000 times to the network loaded with 32 patterns and $\lambda = 0.01$. The basin of attraction grows strongly (by a factor of ten or more) and also takes on a more regular shape. The probability of retrieving the stored patterns is much improved (so “forgetting improves the memory”).

In a somewhat modified version of this method the network is allowed to develop from the stored patterns, deteriorated by random noise. One then not only weakens the synaptic connections by unlearning the final state s_i^∞ , but also simultaneously relearns the correct starting pattern ν :

$$w_{ij} \rightarrow w_{ij} - \frac{\lambda}{N} (s_i^\infty s_j^\infty - \sigma_i^\nu \sigma_j^\nu) \quad (3.2.77)$$

If the pattern was recalled without fault, the synapses remain unchanged according to this prescription. With this method the storage capacity can be increased to $\alpha = 1$, and the storage of strongly correlated patterns becomes possible.

3.3 Simple perceptron

The neural network models discussed so far were all aimed at storing and recalling given information. Memory is an important function of the brain, but far from the only one. Another important task of the central nervous system is to learn reactions and useful behavior that permit survival in an often hostile environment. In a highly simplified perspective one may identify this role of the brain with that of the supervising element

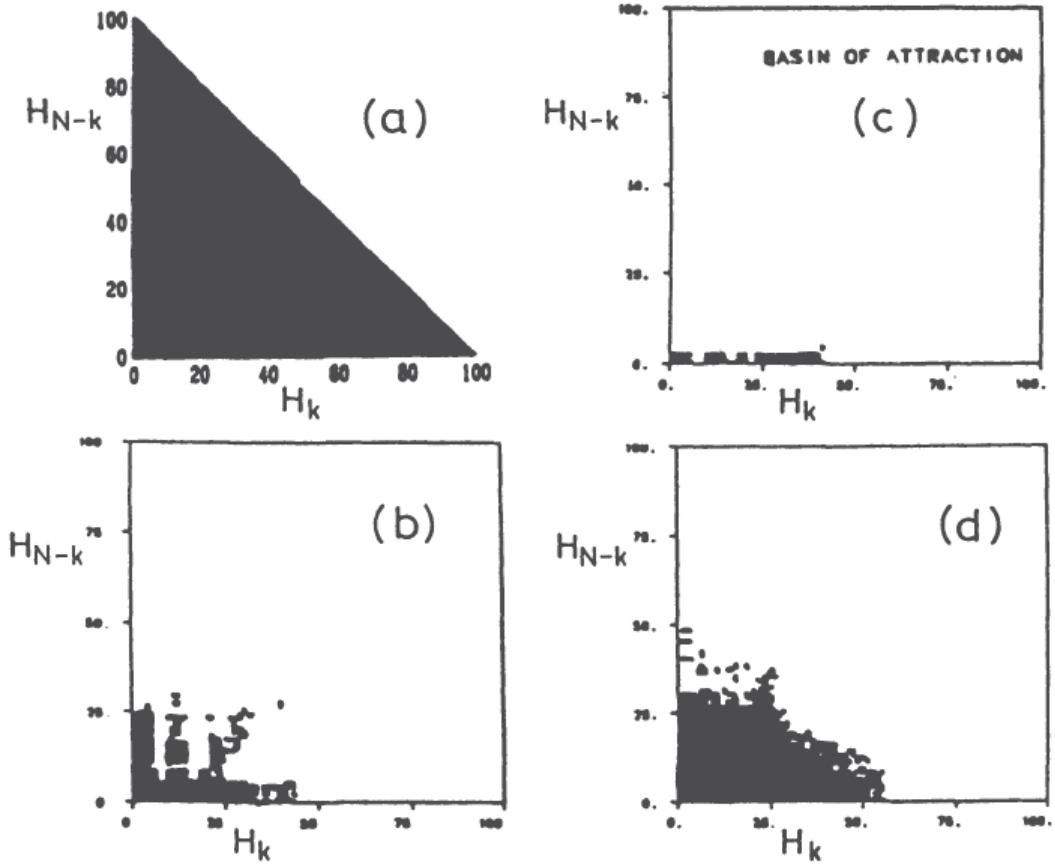


Figure 3.13: Basins of attraction in a Hopfield network with 200 neurons for (a) 1, (b) 28, (c) 32 memory states. After deliberate forgetting the basin expands strongly (d).

in a control circuit. We shall therefore introduce the term *cybernetic networks* for neural networks that provide an optimal reaction or answer to an external stimulus. Such networks usually have a structure that fundamentally differs from that of memory networks. In particular, the synaptic connections are generally *not symmetric* ($w_{ij} \neq w_{ji}$); often they are maximally asymmetric, i.e. unidirectional. As a result, the theories of thermodynamic equilibrium systems have no direct application to cybernetic networks, and much less general insights are known into their behavior.

The best-studied class of cybernetic networks are the so-called *feed-forward layered* neural networks, or simply **perceptrons**, where information flows in one direction between several distinct layers of neurons, as illustrated in Figure 3.14. At one end is the *input layer* composed of sensory neurons, which receive external stimuli, at the other end is an *output layer* often composed of motor neurons, which cause the desired reaction.

In principle a perceptron can contain an arbitrary number of layers of neurons in addition to the input and output layers¹⁰, but only rather simple cases have been studied in depth. Here we begin with the so-called *simple perceptron*, where the input layer feeds directly into the output layer without the intervention of inner, or “*hidden*”, layers of neurons. We shall denote the states of the input neurons by σ_k ($k = 1, \dots, N_{\text{in}}$); those of the output neurons are labeled by S_i ($i = 1, \dots, N_{\text{out}}$). The activation of the output neurons by the

¹⁰In this sense we talk about “deep learning” for algorithms that use multiple layers to progressively extract higher-level features from the raw input.

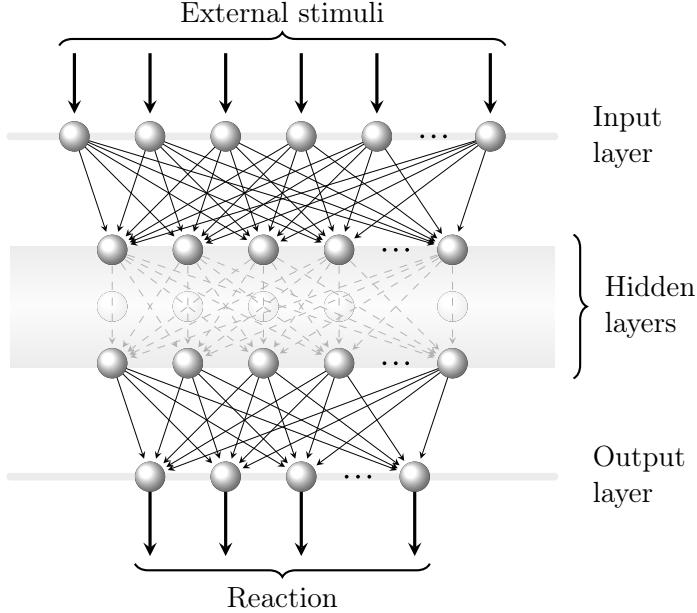


Figure 3.14: Schematic view of a feed-forward layered neural network (perceptron). The arrows represent the synaptic couplings w .

input layer may be determined by the nonlinear function f as¹¹

$$S_i = f(h_i) = f\left(\sum_{k=1}^{N_{\text{in}}} w_{ik}\sigma_k\right) \quad (3.3.1)$$

where h_i is the linear combination

$$h_i = \sum_{k=1}^{N_{\text{in}}} w_{ik}\sigma_k \quad (3.3.2)$$

The fact that the variables σ_k occur only on the right-hand side of this relation, while the S_i occur only on the left-hand side, is expression of the directedness of the networks: information is fed from the input layer into the neurons of the output layer, but not vice versa. The function $f(h)$ may be considered either a stochastic law, where it determines the probability of the values $S_i = \pm 1$, or a continuous function, if the neurons are assigned analog values (deterministic network). Here we concentrate on the latter case.

The task now is to choose the synaptic connections w_{ik} in such a way that a certain input σ_k leads to the desired reaction, specified by the correct states of the neurons in the output layer, which we denote by $S_i = \zeta_i$. Of course, this condition must not only be satisfied for a single input, but for a number of cases (i.e. different external stimuli) indicated by the superscript μ : $S_i^\mu \equiv S_i[\sigma_k^\mu] = \zeta_i^\mu$ with $\mu = 1, \dots, p$. In other words, we want to optimize the network through a suitable choice of the couplings w_{ik} such that it is able to get the desired output reaction for more than one input pattern at the same time. An explicit function allowing the calculation of the w_{ik} from the input σ_k^μ and the desired output ζ_i^μ is not known. However, it is possible to construct *iterative* procedures which converge to the desired values of synaptic connections, if those exist in principle.

For this purpose, it is more appropriate to introduce first a sort of performance measure, that is a notion of error that it is able to tell us at each iteration how far we are from

¹¹For simplicity, we momentarily neglect a possible activation threshold ϑ_i for each of the output neurons.

the optimal result, so we can adjust the algorithm accordingly for the next step. Here the performance measure is given by the general expression

$$D = \frac{1}{2} \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} (S_i^\mu - \zeta_i^\mu)^2 \quad (3.3.3)$$

which is a sort of Hamming distance between the actual and the correct outputs. Since we want the output layer S_i^μ to produce the correct reaction ζ_i^μ , for all μ simultaneously, our goal clearly becomes the *minimization* of D .

As said before, we do not have an explicit formula for the optimal tuning of the synaptic couplings; hence in the following we will describe a standard procedure for the search of the minima, guided by the previous notion of distance. The basic idea is to start with an initial random choice of the w_{ik} . Then, at each step we compute the distance D and we update the w_{ik} by adding corrections δw_{ik} which lead to a *lowering* of D .

Graphically speaking, we could imagine the following situation (Fig. 3.15). Suppose we have a function $f(x)$ describing the distance we are trying to minimize, in order to obtain the desired reaction. Then let's call x_0 the state of the synaptic connections corresponding to a minimum configuration for f (i.e. leading to $S_i^\mu = \zeta_i^\mu$) and suppose we are actually in a different state x ($S_i^\mu \neq \zeta_i^\mu$). The derivative of $f(x)$ in x has a certain sign: if $f'(x) > 0$ (positive slope), this means that we have to move leftwards, i.e. towards smaller values of x , in order to reduce the function f ; similarly, if $f'(x) < 0$ (negative slope), then we have to move rightwards towards larger values of x in order to lower f . Mathematically speaking, we can write this observation as

$$x_{n+1} = x_n - \varepsilon f'(x_n) \quad (3.3.4)$$

where x_{n+1} is the final point that we reach by moving of a quantity $\varepsilon f'(x)$, which constitutes one single step of our iterative procedure. The factor ε is an arbitrary positive constant that should be chosen sufficiently small to avoid overshooting the goal. Clearly, from (3.3.4) it is easy to convince yourself that the signs are correct: if $f'(x) > 0$ we move leftwards, and indeed $x_{n+1} < x_n$ (and vice versa). So at each step we evaluate the derivative and depending on its sign we move in the optimal direction to reduce f . In the limit of a large number of iterations we should expect to converge to x_0 , the desired minimum.

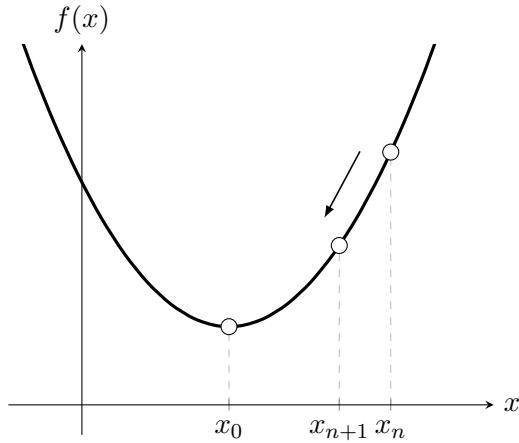


Figure 3.15

This reasoning has been made in one dimension, but in our case f , which must be identified with D , is a multivariate function since it depends on the various couplings w_{ik} .

3 Neural networks

Accordingly, we have to replace (3.3.4) with the more general form

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \varepsilon \nabla f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_n} \quad (3.3.5)$$

The method works because the gradient always points in the direction of strongest change of the function; hence the negative gradient follows the direction of steepest descent. For this reason, this method is sometimes referred to as “gradient learning” or method of steepest descent.

At this point all that remains is to translate these observations for our specific case of simple perceptron. First of all, with the help of (3.3.1) we rewrite the deviation D as an explicit function of the synaptic connections:

$$D[w_{ik}] = \frac{1}{2} \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} [\zeta_i^\mu - f(h_i^\mu)]^2 = \frac{1}{2} \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} \left[\zeta_i^\mu - f\left(\sum_{k=1}^{N_{\text{in}}} w_{ik} \sigma_k^\mu \right) \right]^2 \quad (3.3.6)$$

where $h_i \equiv h_i^\mu$ is itself a function of μ . In order to lower the value of D , we now compute the gradient with respect to the synaptic couplings¹²:

$$\frac{\partial D}{\partial w_{ik}} = - \sum_{\mu=1}^p [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \frac{\partial h_i^\mu}{\partial w_{ik}} = - \sum_{\mu=1}^p \Delta_i^\mu \sigma_k^\mu \quad (3.3.7)$$

with the abbreviation

$$\Delta_i^\mu := [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \quad (3.3.8)$$

Here $f'(h_i^\mu)$ denotes the derivative of the function f evaluated at the point h_i^μ . Moreover we have exploited the usual chain rule for differential calculus¹³. Finally, if we apply the gradient expression (3.3.5) we obtain the small deviation

$$\delta w_{ik} = -\varepsilon \frac{\partial D}{\partial w_{ik}} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu \sigma_k^\mu \quad (3.3.9)$$

In conclusion, our iterative technique to obtain the desired reactions $S_i^\mu = \zeta_i^\mu$ can be summarized as follows:

$$\boxed{\begin{cases} w_{ik} \rightarrow w_{ik} + \delta w_{ik} \\ \delta w_{ik} = -\varepsilon \frac{\partial D}{\partial w_{ik}} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu \sigma_k^\mu \end{cases}} \quad (3.3.10)$$

$$\boxed{\begin{cases} w_{ik} \rightarrow w_{ik} + \delta w_{ik} \\ \delta w_{ik} = -\varepsilon \frac{\partial D}{\partial w_{ik}} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu \sigma_k^\mu \end{cases}} \quad (3.3.11)$$

where the first equation indicates the small modifications we make to the synaptic couplings at each step, while the second one explicitly gives us a recipe for how to calculate them. We call this procedure the **training phase** (or routine) of the network.

At the beginning of this section we have said that the function $f(h)$ may be considered either as a stochastic law or a deterministic law, for which a standard choice is

$$f(h) = \tanh(\beta h) \quad (3.3.12)$$

¹²Pay attention to the notation, taken from [6], which can be misleading! The derivatives are made each time with respect to a specific weight w_{ik} . Obviously these indices have nothing to do with those in the summation defining D (see equation (3.3.6)), which for clarity should have been renamed as i' and k' , for example. The same notation is kept until the end of the chapter.

¹³If $f(x_1, \dots, x_n)$, then $\frac{df}{dx_k} = \sum_{i=1}^n \frac{\partial f}{\partial x_i} \frac{\partial x_i}{\partial x_k}$

An important remark is that $f(h)$ does not necessarily need to be continuous if the right precautions are taken. For example, if we chose $f(h) = \text{sgn}(h)$ we would have a potential problem in computing the derivative (as prescribed by the perceptron learning rule), since the sign function presents a jump at $h = 0$. Hence we would have $f'(h) = 2\delta(h)$, which is not a good situation because we don't want to deal with Dirac delta functions. However, this is an apparent problem: in practice, where the derivative is not well-defined (in the sense of "standard" derivative) we can substitute $f'(h)$ with any arbitrary number because so much the arbitrariness of this choice is automatically absorbed in the arbitrariness of the constant ε . So we can choose the combination $\varepsilon f'(h)$ freely in order to obtain the best result. This is to say that having a continuous function is not strictly binding.

This *perceptron learning rule* is rather general and works also for more complex neural networks with hidden layers of neurons. But not all that glitters is gold. This technique has also some weaknesses. First of all, the usual problem of local minima where the system can get stuck is still present because the algorithm is essentially an hill-climbing algorithm in which the steps always point to the nearest minimum. Therefore, if the distance notion that we have chosen to adopt has more than one local minimum, this procedure could bring us towards one of these false minima, which are not our desired output.

Second, the procedure may have a very slow convergence, if not completely absent. In fact, if we are in a point x very close to the minimum x_0 and the step (which is a combination of the free parameter ε and the derivative) goes beyond the minimum, we end up in a configuration where the sign of the derivative is changed. Then, if we iterate and the next step is still larger than necessary, we exceed again the minimum in going in the opposite direction (with respect to the previous step). It is obvious that if this situation does not halt, the system could be found in a configuration in which it alternatively makes one step to the right and one to left of the minimum. The result would be a continuous jumping between the two sides which can greatly slow down the convergence towards the desired state, or even make it impossible. A possible way out is to combine the step $\Delta_n := -\varepsilon f'(x_n)$ we would take at the present time with the step $\Delta_{n-1} := -\varepsilon f'(x_{n-1})$ taken at the previous iteration (which is usually discarded):

$$\bar{\Delta}_n = \alpha \Delta_n + (1 - \alpha) \Delta_{n-1} \quad (3.3.13)$$

leading to $x_{n+1} = x_n + \bar{\Delta}_n$. Since the two combined steps have different signs close to the minimum their contributions are mutually canceled, thus avoiding the undesired bouncing. Of course, if we are far from the minimum, subsequent steps all point towards it and this corrective step is not so important. In the language of numerical analysis this is called a *relaxation* procedure.

3.3.1 The exclusive-OR (XOR) gate

The existence of a simple but effective learning algorithm for perceptrons without hidden neurons makes these very attractive as neural-network models. Unfortunately, there exist elementary problems that cannot be handled by such a system. Let's consider for simplicity a deterministic perceptron with just two input neurons and one output neuron (Fig. 3.16). Is this network architecture able to reproduce any type of function, in particular logical functions (for example) like AND, OR, XOR gates?

Before answering, let's recall how they are defined. The result of the logical AND (symbol \wedge) between a certain number of variables is 1 if all the variables are 1, while it is 0 if even just one variable is 0. Instead, the result of the logical OR (symbol \vee) between a certain number of variables is 1 if at least one variable is 1, while it is 0 if all the variables are

3 Neural networks

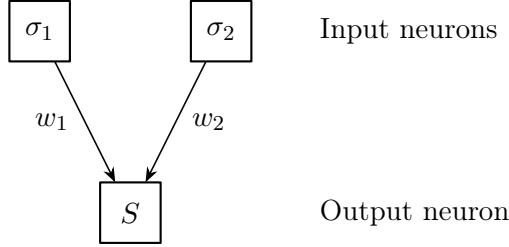


Figure 3.16

equal to 0. If more than one variable is 1, the result is always 1. The possible results of these two functions between our two input neurons are shown in Tables 3.1a and 3.1b (they are called truth tables), respectively. Notice that the definitions are given in terms of the Boolean variables 0 and 1, but we can easily extend them for -1 and $+1$, which are the possible outcomes of our neurons (we identify 0 with -1 and 1 with $+1$).

σ_1	σ_2	AND
+	+	+
+	-	-
-	+	-
-	-	-

(a) Logical AND(σ_1, σ_2).

σ_1	σ_2	OR
+	+	+
+	-	+
-	+	+
-	-	-

(b) Logical OR(σ_1, σ_2).

Table 3.1

On the other hand, the exclusive-OR, or simply XOR, is a logical operation that gives 1 when the number of inputs equal to 1 is odd, and 0 otherwise. For just two inputs, we say that the XOR is 1 if and only if its arguments differ (one is 0, the other is 1). It is called exclusive-OR because it has essentially the same behavior of the inclusive OR, with the only difference of excluding all combinations in which the two variables are equal (see Table 3.2).

σ_1	σ_2	XOR
+	+	-
+	-	+
-	+	+
-	-	-

Table 3.2: Logical XOR(σ_1, σ_2).

It is possible to show that AND and OR functions can actually be represented by our simple perceptron of Figure 3.16 with a suitable choice of the synaptic connections. Instead, the XOR operation cannot, independently of the values we decide to assign to the synaptic connections! In fact, when this function is to be represented by a deterministic perceptron with two input neurons and one output neuron, the network architecture is explicitly given by

$$S = \text{sgn}(w_1\sigma_1 + w_2\sigma_2 - \vartheta) \quad (3.3.14)$$

which is nothing but (3.3.1) where we used the deterministic law $f(h) = \text{sgn}(h)$. For completeness we have also added the threshold polarization potential ϑ of the output

neuron, which until now we had always neglected for simplicity. Our goal is to obtain an output S which coincides with the desired output ζ typical of a XOR gate. Therefore, in order for equation (3.3.14) to produce the same outcome as the $\text{XOR}(\sigma_1, \sigma_2)$ operation, it is necessary that the following conditions for the argument of the sign function are met:

σ_1	σ_2	$\zeta = \text{XOR}$	Condition
+	+	-	$+w_1 + w_2 - \vartheta < 0$
+	-	+	$+w_1 - w_2 - \vartheta > 0$
-	+	+	$-w_1 + w_2 - \vartheta > 0$
-	-	-	$-w_1 - w_2 - \vartheta < 0$

Table 3.3

Now if we multiply the second inequality by -1 and we add it to the first we obtain

$$\begin{cases} +\textcircled{1}: +w_1 + w_2 - \vartheta < 0 \\ -\textcircled{2}: -w_1 + w_2 + \vartheta < 0 \end{cases} \implies w_2 < 0 \quad (3.3.15)$$

Analogously, if we multiply the fourth inequality by -1 and we add it to the third we obtain

$$\begin{cases} +\textcircled{3}: -w_1 + w_2 - \vartheta > 0 \\ -\textcircled{4}: +w_1 + w_2 + \vartheta > 0 \end{cases} \implies w_2 > 0 \quad (3.3.16)$$

The two conditions are clearly contradictory and cannot be satisfied simultaneously, i.e. the desired task cannot be performed by the network for *any* choice of the synaptic parameters w_1 , w_2 and ϑ . This result is not entirely surprising, because there are only three adjustable parameters but a total of four conditions. Although these are only inequalities, the example shows that the available freedom to choose the synapses may not be sufficient to solve the problem.

This condition has also a simple geometrical interpretation. The space of input variables $\sigma_i = \pm 1$ consists of the corners of a square in our simple perceptron (or a N_{in} -dimensional hypercube if we consider the general case with more inputs), as shown in Figure 3.17. If now we consider the equation $w_1\sigma_1 + w_2\sigma_2 - \vartheta = 0$, this is the equation of a straight line in that space. As said by the equation itself, along this line the output is zero, whereas we expect it to be positive on one side and negative on the other side. It is obvious that for the AND case (Fig. 3.17a) we are able to choose the parameters in such a way that the line is in accordance with the desired underlying output. Instead, for the XOR case (Fig. 3.17b) this is not possible by using just one line because on one of its sides the desired output is both $+1$ and -1 . For the XOR operation two such lines are required. For these reasons the XOR function is said to be *linearly inseparable*, since there is no way of dividing the space of input variables into regions of equal output by a single linear condition. Instead, AND and OR functions are called *linearly separable* because a line (or more generally an $(N_{\text{in}} - 1)$ -dimensional hyperplane) can be found which separates the regions of function values $+1$ and -1 . Accordingly, the class of linearly inseparable functions can not be represented by a simple perceptron (we need to include hidden layers).

This kind of criticism of the concept of simple perceptrons, which was particularly emphasized by the MIT school, almost put an end to the study of neural networks in the late 1960s, at least concerning their possible use as general information processing machines.

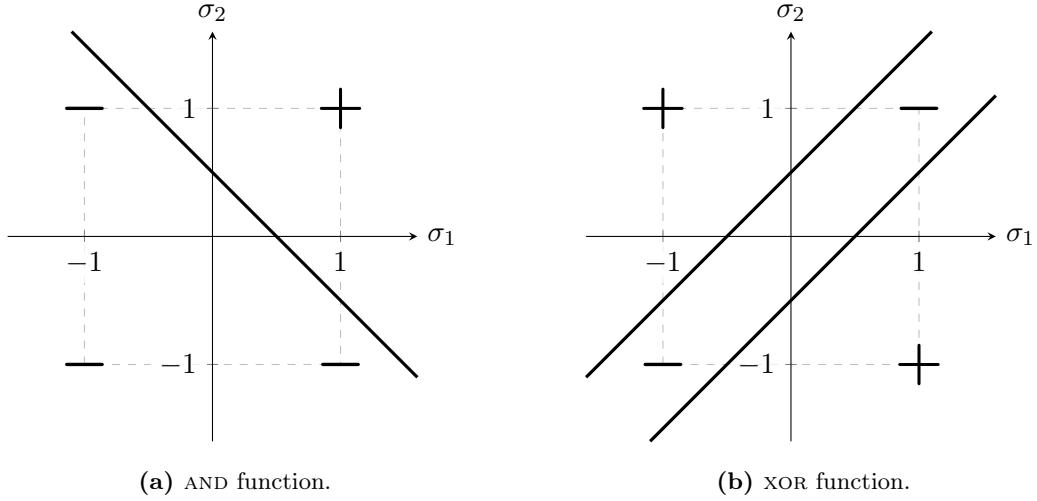


Figure 3.17

3.4 Multilayer perceptron

3.4.1 Solution of the XOR problem

Before we enter into the discussion of how one can derive a learning rule for multilayered perceptrons, it is useful to consider first a simple example. Let's consider the exclusive-XOR (XOR) function, with which we are already familiar from the previous section. In order to circumvent the "no-go" theorem derived for simple perceptrons in the previous section, we add a *hidden layer* containing two neurons which receive signals from the input neurons and feed the output neuron (see Fig. 3.18). We denote the states of the hidden neurons by the variables s_j ($j = 1, \dots, N_{\text{hid}}$). Capital letters S will henceforth symbolize output neurons, while hidden neurons are denoted by lower-case letters s . The synaptic connections between the hidden neurons and the output neurons are denoted by w_{ij} ; those between the input layer and the hidden layer by \bar{w}_{jk} . The threshold potentials of the output neurons are called ϑ_i ; those of the hidden neurons are called $\bar{\vartheta}_j$.

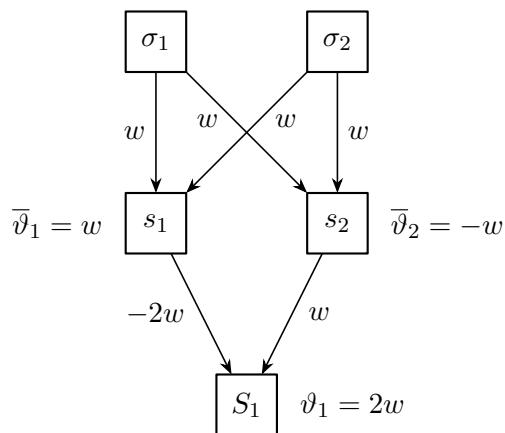


Figure 3.18

The state of our network with just one hidden layer is then governed by the following

equations:

$$s_j = \text{sgn}(\bar{h}_j) = \text{sgn}\left(\sum_{k=1}^{N_{\text{in}}} \bar{w}_{jk} \sigma_k - \bar{\vartheta}_j\right) \quad (3.4.1)$$

$$S_i = \text{sgn}(h_i) = \text{sgn}\left(\sum_{j=1}^{N_{\text{hid}}} w_{ij} s_j - \vartheta_i\right) \quad (3.4.2)$$

It is clear that we have chosen a deterministic perceptron in which the nonlinear function f is given by $f(h) = \text{sgn}(h)$, as before. The previous equations hold in general for an arbitrary number of neurons in the input, hidden and output layers; hence we wrote them by leaving the extrema of the summations as general as possible (in case of future use). However, in the case of our specific example on the XOR function, the indices j and k (hidden and input neurons, respectively) run from 1 to 2, whereas the index i takes only the value 1 (single output neuron). So the state of the only output neuron we have is just S_1 , or simply S .

As the table below shows, the following choice of synaptic couplings and threshold potentials, also indicated in Figure 3.18, provides one of the possible solutions to our problem:

$$\bar{w}_{jk} = w \quad (\forall j, k) \quad (3.4.3)$$

$$w_{11} = -2w \quad (3.4.4)$$

$$w_{12} = w \quad (3.4.5)$$

$$\bar{\vartheta}_1 = w \quad (3.4.6)$$

$$\bar{\vartheta}_2 = -w \quad (3.4.7)$$

$$\vartheta_1 = 2w \quad (3.4.8)$$

We check the validity of this solution by explicit evaluation of the XOR function for all four possible input combinations:

σ_1	σ_2	ζ_1	\bar{h}_1	\bar{h}_2	s_1	s_2	h_1	S_1
+1	+1	-1	w	3w	+1	+1	-3w	-1
+1	-1	+1	-w	w	-1	+1	w	+1
-1	+1	+1	-w	w	-1	+1	w	+1
-1	-1	-1	-3w	-w	-1	-1	-w	-1

Table 3.4

As one can see, the hidden neuron s_1 plays the role of a logical element representing the AND function, while the other hidden neuron s_2 emulates a logical OR element. The combination of these two elements allows the generation of the XOR function. This is not an accident: there are indeed two distinct classes of representation of XOR where the output neuron acts either as an OR gate or as an AND gate. So in general a XOR operation between two input variables can be expressed using just AND, OR and NOT (logical negation) gates according to

$$\text{XOR}(A, B) = \begin{cases} (A \vee B) \wedge \overline{A \wedge B} \\ (A \wedge \overline{B}) \vee (\overline{A} \wedge B) \end{cases} \quad (3.4.9)$$

We shall show in the next sections that any Boolean function can be represented by a feed-forward network with one hidden layer and appropriate architecture.

3.4.2 Error back-propagation

To use multilayered networks efficiently, one needs a method to determine their synaptic connections and threshold potentials. A very successful method, usually called **error back-propagation**, was developed independently around 1985 by several research groups. It is based on a generalization of the gradient method discussed in Sec. 3.3. Here we formulate this algorithm for a generic three-layered network (input, one hidden layer and output) with analog-valued neurons. In particular, we demand that the synapses and threshold potentials be chosen such that the output deviation function

$$D[w_{ij}, \vartheta_i, \bar{w}_{ij}, \bar{\vartheta}_i] = \frac{1}{2} \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} [\zeta_i^\mu - f(h_i^\mu)]^2 \quad (3.4.10)$$

becomes as small as possible. Thus we search for a minimum (which ideally should be a global minimum) of the error surface defined by (3.4.10). For this purpose we compute the gradient of D with respect to every parameter and then change the value of the parameters accordingly. Note the subtle difference with respect to the expression for the simple perceptron. In the case of simple perceptron D was a function of just w_{ik} because h_i itself depended only on the couplings w_{ik} between the input and the output neurons (3.3.2). Now D depends on w_{ij} and ϑ_i through $f(h_i)$, and on \bar{w}_{ij} and $\bar{\vartheta}_i$ through s_j and \bar{h}_j . In other words, there is a series of hidden dependencies due to the shapes of the hidden neurons s_j :

$$s_j = f(\bar{h}_j) \quad (3.4.11)$$

$$S_i = f(h_i) \quad (3.4.12)$$

where

$$\bar{h}_j = \sum_{k=1}^{N_{\text{in}}} \bar{w}_{jk} s_k - \bar{\vartheta}_j \quad (3.4.13)$$

$$h_i = \sum_{j=1}^{N_{\text{hid}}} w_{ij} s_j - \vartheta_i \quad (3.4.14)$$

Let's now derive the steps for this error back-propagation. In the first step we consider only synaptic connections at the output neurons

$$\delta w_{ij} = -\varepsilon \frac{\partial D}{\partial w_{ij}} = \varepsilon \sum_{\mu=1}^p [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \frac{\partial h_i^\mu}{\partial w_{ij}} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu s_j^\mu \quad (3.4.15)$$

$$\delta \vartheta_i = -\varepsilon \frac{\partial D}{\partial \vartheta_i} = \varepsilon \sum_{\mu=1}^p [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \frac{\partial h_i^\mu}{\partial \vartheta_i} = -\varepsilon \sum_{\mu=1}^p \Delta_i^\mu \quad (3.4.16)$$

where as usual

$$\Delta_i^\mu := [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \quad (3.4.17)$$

In the next step we consider the parameters associated with synaptic connections between the input and hidden layer. The procedure is exactly the same, except that we have to

apply the substitution rule of differentiation once more:

$$\begin{aligned}
 \delta\bar{w}_{jk} &= -\varepsilon \frac{\partial D}{\partial \bar{w}_{ij}} \\
 &= \varepsilon \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \frac{\partial h_i^\mu}{\partial s_j} \frac{\partial s_j}{\partial \bar{w}_{jk}} \\
 &= \varepsilon \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} \Delta_i^\mu w_{ij} f'(\bar{h}_j^\mu) \frac{\partial \bar{h}_j^\mu}{\partial \bar{w}_{jk}} \\
 &= \varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu \sigma_k^\mu
 \end{aligned} \tag{3.4.18}$$

and similarly

$$\begin{aligned}
 \delta\bar{\vartheta}_j &= -\varepsilon \frac{\partial D}{\partial \bar{\vartheta}_j} \\
 &= \varepsilon \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \frac{\partial h_i^\mu}{\partial s_j} \frac{\partial s_j}{\partial \bar{\vartheta}_j} \\
 &= \varepsilon \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} \Delta_i^\mu w_{ij} f'(\bar{h}_j^\mu) \frac{\partial \bar{h}_j^\mu}{\partial \bar{\vartheta}_j} \\
 &= -\varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu
 \end{aligned} \tag{3.4.19}$$

with the new abbreviation

$$\bar{\Delta}_j^\mu := \left(\sum_{i=1}^{N_{\text{out}}} \Delta_i^\mu w_{ij} \right) f'(\bar{h}_j^\mu) \tag{3.4.20}$$

One should note that the equations (3.4.18) and (3.4.19) determining the synaptic adjustments have the same form as the synaptic equations (3.4.15) and (3.4.16) derived earlier. Only the expression for $\bar{\Delta}_j^\mu$ differs from that for Δ_i^μ , from which it may be obtained recursively. In conclusion, at each iteration one updates the parameters in the following way:

$$w_{ij} \rightarrow w_{ij} + \delta w_{ij} \tag{3.4.21}$$

$$\vartheta_i \rightarrow \vartheta_i + \delta \vartheta_i \tag{3.4.22}$$

$$\bar{w}_{jk} \rightarrow \bar{w}_{jk} + \delta \bar{w}_{jk} \tag{3.4.23}$$

$$\bar{\vartheta}_j \rightarrow \bar{\vartheta}_j + \delta \bar{\vartheta}_j \tag{3.4.24}$$

with the values given above.

One interesting aspect of this recursion relation is that it resembles (3.4.13), which determines the state of the two final layers of the neural network. In a sense, therefore, the error-correction scheme works by propagating the information about the deviation from the desired output “backward” through the network, against the direction of synaptic connections.

3.4.2.1 Arbitrary number of hidden layers

The method is easily generalized to neural networks with more than one hidden layer of neurons. In fact, an equation of the form (3.4.20) always expresses the parameters $\bar{\Delta}$ in

3 Neural networks

terms of those obtained for the previous layer (in the backward propagating sense). As an exercise we explicitly write down the full set of equations for a multilayer perceptron with two layers of hidden neurons (see Fig. 3.19). The variables pertaining to the additional layer, here taken to be the one directly connected to the input layer, are denoted by an additional “bar”, e.g. \bar{s}_k , $\bar{\bar{w}}_{jk}$, $\bar{\vartheta}_k$ and $\bar{\bar{\vartheta}}_k$.

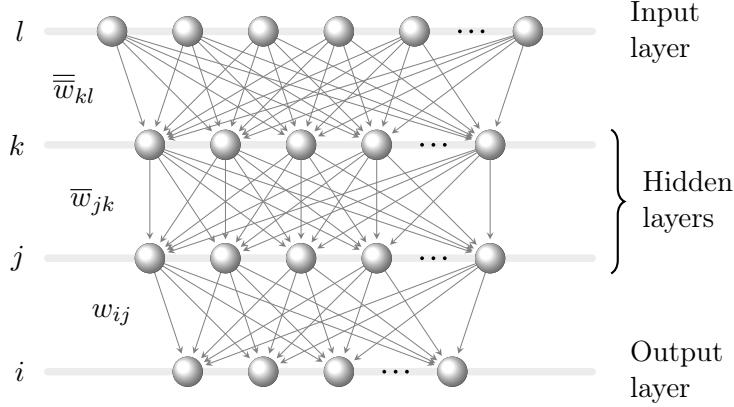


Figure 3.19: General architecture of a multilayer perceptron with two hidden layers of neurons.

The states of the neurons in the various layers are then determined from the three equations

$$\bar{s}_k = f(\bar{\bar{h}}_k), \quad \bar{\bar{h}}_k = \sum_{l=1}^{N_{\text{in}}} \bar{\bar{w}}_{kl} \sigma_l - \bar{\bar{\vartheta}}_k \quad (3.4.25)$$

$$s_j = f(\bar{h}_j), \quad \bar{h}_j = \sum_{k=1}^{N_{\text{hid},1}} \bar{w}_{jk} \bar{s}_k - \bar{\vartheta}_j \quad (3.4.26)$$

$$S_i = f(h_i), \quad h_i = \sum_{j=1}^{N_{\text{hid},2}} w_{ij} s_j - \vartheta_i \quad (3.4.27)$$

The equations defining the adjustment of synapses are

$$\delta w_{ij} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu s_j^\mu, \quad \delta \vartheta_i = -\varepsilon \sum_{\mu=1}^p \Delta_i^\mu, \quad \Delta_i^\mu = [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \quad (3.4.28)$$

$$\delta \bar{w}_{jk} = \varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu \bar{s}_k^\mu, \quad \delta \bar{\vartheta}_j = -\varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu, \quad \bar{\Delta}_j^\mu = \left(\sum_{i=1}^{N_{\text{out}}} \Delta_i^\mu w_{ij} \right) f'(\bar{h}_j^\mu) \quad (3.4.29)$$

$$\delta \bar{\bar{w}}_{kl} = \varepsilon \sum_{\mu=1}^p \bar{\bar{\Delta}}_k^\mu \sigma_l^\mu, \quad \delta \bar{\bar{\vartheta}}_k = -\varepsilon \sum_{\mu=1}^p \bar{\bar{\Delta}}_k^\mu, \quad \bar{\bar{\Delta}}_k^\mu = \left(\sum_{i=1}^{N_{\text{out}}} \bar{\Delta}_i^\mu \bar{w}_{ik} \right) f'(\bar{\bar{h}}_k^\mu) \quad (3.4.30)$$

Learning in networks with hidden layers is, therefore, easily implemented. However, no general convergence theorem for the learning process is known for such networks.

Example

As an example, we implemented the error back-propagation technique in the case of simple logic functions \rightarrow . First we have defined some functions.

In[2]: `net_f_df(z)` is actually our $f(h)$, but it also furnishes the derivative $f'(h)$, since it will be used many times in computing the synaptic connections. In this case we have chosen the sigmoidal function $f(h) = 1/(1 + e^{-h})$ without any β parameter.

In[3]: `forward_step(y,w,b)` indicates the general structure of any of the h -functions (3.4.25), (3.4.26) or (3.4.27) and the corresponding state of the neuron. In fact, the potential of a layer is computed through the synaptic connections and the states of the neurons of the previous input layer, which in the code has been written as the matrix product `dot(y,w)` between the array `y` of the input neurons and the matrix `w` of the couplings. `b`, called bias in the code, is actually what we have called so far the threshold potential ϑ . Naturally, `forward_step(y,w,b)` also gives $f'(h)$ because it is based on the previous function `net_f_df(z)`, which does so.

In[4]: `apply_net(y_in)` represents the setting of the perceptron at each step. The array `y_layer` contains the states of the neurons in the various layers at the specific step of the overall iterative procedure (i.e. of the error back-propagation) in which we are. In particular, the first row `y_layer[0]` always contains the input neurons `y_in`, that is our σ . Then, a loop over the remaining layers (hidden and output) progressively determines the values of the corresponding neurons through (3.4.25–3.4.27).

In[6]: `backward_step(delta,w,df)` returns the values `dot(delta,transpose(w))*df`, which correspond exactly to equations (3.4.29) and (3.4.30). In practice we are using the derivatives and the previous Δ to calculate the barred ones.

In[7]: `backprop(y_target)` instead computes the first Δ , whose definition given in equation (3.4.28) is slightly different from the barred ones. Furthermore, within the same command this function exploits these Δ to obtain the synaptic and threshold modifications for all the layers (3.4.25–3.4.27). Notice also that we divided by the number of patterns in the training (`batchsize`) to avoid having to change the ε value if we change the number of patterns. In other words, this compensates for the fact that we should sum over all the patterns, a procedure that would increase the value of D .

In[8]: Notice that in the previous input we have not included the small parameter ε of the gradient technique. This is done after in the function `gradient_step(eta)`, which finally updates all the synaptic couplings and threshold potentials (here ε is called `eta`).

In[9]: Finally, we use the function `train_net(y_in,y_target,eta)`, which summarizes all the previous functions, in order to make a single update of all the perceptron architecture. This is the function to be used for the training procedure of the net.

At this point we just need to set the involved variables, like number of layers, their size, the starting input, etc., and to run the code! In particular we have chosen a random start (i.e. random inputs) and random connections and couplings. For the moment let's forget things about plotting and animations, and go to the “Logic” section. It is interesting to notice that there have been defined several activation functions $f(z)$ for the neurons:

- `sigmoid` is the classical Fermi function $f(z) = 1/(1 + e^{-z})$;
- `jump` is the Heaviside step function $f(z) = \Theta(z)$;

- **linear** is the line $f(z) = z$;
- **ReLU**, from rectified linear unit, also called ramp function, is given by $f(z) = z\Theta(z)$.

In fact, when the perceptron that we are training has many layers sometimes is better to use different functions in different layers in order to increase the efficiency.

Scrolling down in the code are shown the desired outputs for the exact AND and for the exact XOR for different layers size, i.e. with synaptic values and thresholds fixed exactly from the beginning in order to give those specific results. In other words, these are the theoretical outputs and the form of the perceptron if we already knew from the beginning the perfect couplings to be used. For the former it is sufficient a simple perceptron, i.e. without hidden layers; for the latter we have showed both a neural network with a single hidden layer, which is the situation that we have studied before (see Fig. 3.18), and with two hidden layers, which is the more general (but also more time-consuming) treatment for logical functions, as we will demonstrate in the next section.

In the last part of the code we have also implemented the complete training routine for reproducing logical functions (with arbitrary choice of layers size). The animations show the various attempts made by the neural network to try to produce the desired output (the function `my_target`) by changing iteratively the synaptic connections and thresholds. Since we have chosen a random start, the learning protocol takes a while. Moreover, we note that the `cost` function, i.e. our D function we want to minimize, actually reaches a minimum at the end of the procedure.

3.4.3 Boolean functions

Deterministic neural networks with binary neurons (i.e. assuming just two values) are ideally suited to represent logical, or Boolean, functions. These are functions whose arguments and output values are logical variables taking only the two values “True” and “False” (“T” and “F”), sometimes also denoted by “1” and “0”, or “+1” and “−1” in the Boolean representation. The range of definition of a Boolean function with N arguments covers exactly 2^N elements, that is we can create at most 2^N function arguments $(\sigma_1, \dots, \sigma_N)$ having different values of the input variables σ_k . Moreover, since we have 2^N different ways to organize the inputs and the output (the function value) has only two possible values, then we can build at most 2^{2^N} different logical functions. Hence there exist 2^{2^N} different Boolean functions with N arguments, a vast number even for moderately large N . Nonetheless, one can prove the following statement.

Every Boolean function can be represented by a perceptron with just a single, albeit very large, hidden layer.

Proof. We take an input layer of N binary neurons σ_k ($k = 1, \dots, N$) to represent the function arguments and a single output neuron S to represent the function values:

$$S = F(\sigma_1, \dots, \sigma_N) \tag{3.4.31}$$

The logical truth values “T” and “F” are represented by the neuron states +1 and −1, and 2^N hidden neurons denoted by s_j ($j = 0, \dots, 2^N - 1$) form the hidden layer. Since there are exactly 2^N possible input patterns, with this architecture we are dedicating one hidden unit to each input pattern. It will become clear in a moment the motivation for this statement and also why the index j runs from 0 to $2^N - 1$, rather than from 1 to 2^N , as we were used to see. Finally, we assume that the network is fully connected in the forward direction (see

Fig. 3.20). We also consider the sign function $f(h) = \text{sgn}(h)$ as nonlinear deterministic law for the discharge phase of neurons.

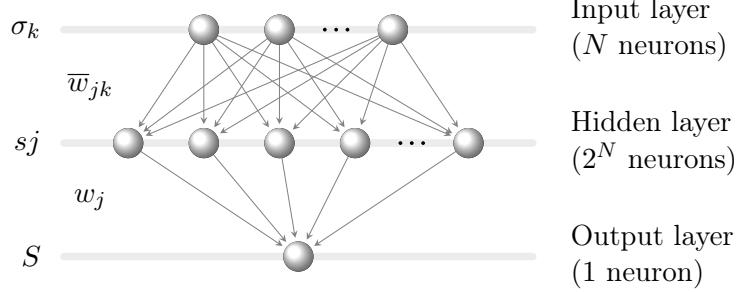


Figure 3.20

We start the demonstration by fixing the strengths of the synaptic connections between the input and hidden neurons uniformly, except for their sign: $|\bar{w}_{jk}| = w = 1$. The sign is determined as follows. Take any neuron from the hidden layer and write its index j in binary representation. Owing to the above choice of counting j from 0, this binary number has at most N digits, or exactly N digits, if leading zeros are added:

$$j^{\text{bin}} = (\alpha_1, \dots, \alpha_N) \quad (3.4.32)$$

with α_ν equal to 0 or 1. Therefore these α_ν are the digits of the number j in binary notation. At this point we choose to set $\bar{w}_{jk} = +w$ when $\alpha_k = 1$, and $\bar{w}_{jk} = -w$ when $\alpha_k = 0$. In closed notation this definition reads

$$\bar{w}_{jk} \equiv \bar{w}_{j^{\text{bin}} k} = (2\alpha_k - 1)w \quad (3.4.33)$$

Notice that the left-hand side depends on j , while the right-hand side not. At first glance it may seem that something has been forgotten, but recall that α_k is the k -th digit associated to the *specific number* j written in binary notation. So the dependence on j is actually inherent in the choice of α_k ¹⁴. We also choose the threshold potentials of all hidden neurons uniformly, namely

$$\bar{\vartheta}_j \equiv \bar{\vartheta} = (N - 1)w \quad (3.4.34)$$

In order to recognize the meaning of these assignments, we compute the total synaptic potential at the j -th hidden neuron:

$$\bar{h}_j = \sum_{k=1}^N \bar{w}_{jk} \sigma_k - \bar{\vartheta}_j = w \left[\sum_{k=1}^N (2\alpha_k - 1) \sigma_k - (N - 1) \right] \quad (3.4.35)$$

All N terms in this sum over k are clearly of modulus 1: in fact $2\alpha_k - 1$ and σ_k can only assume the values $+1$ or -1 . Therefore, their sum is only equal to N if *all* terms are equal to $+1$, whence $\bar{h}_j > 0$. Otherwise, the value of the sum is *at most* $N - 2$: this corresponds to the case in which $N - 1$ terms in the sum are equal to $+1$ and the remaining one, supposed to be different, is necessarily -1 . Accordingly, with this second option the synaptic potential is always negative ($\bar{h}_j < 0$) because $N - 2m < N - 1$ for every $m = 1, \dots, N$ (the result changes by steps of 2). So this implies that the total synaptic potential for the j -th neuron is positive only if $2\alpha_k - 1 = \sigma_k$ for all k ; in compact form

$$s_j = \text{sgn}(\bar{h}_j) = \begin{cases} +1, & \text{if } 2\alpha_k - 1 = \sigma_k \quad \forall k \\ -1, & \text{otherwise} \end{cases} \quad (3.4.36)$$

¹⁴To be more precise we should have written $\alpha_k^{(j)}$.

3 Neural networks

But once we have fixed the values σ_k of the input neurons, the condition $2\alpha_k - 1 = \sigma_k$ for all k determines a set $\{\alpha_k\}_{k=1,\dots,N}$ which strictly identifies a specific number j in binary notation, that is a specific neuron. This means that only a single neuron in the hidden layer will and can become active (value +1) for a given input. This neuron is given by the condition $\alpha_k = (\sigma_k + 1)/2$ for all k in binary representation:

$$s_j = \begin{cases} +1, & \text{if } j = j_0 \\ -1, & \text{if } j \neq j_0 \end{cases} \quad (3.4.37)$$

with

$$j_0^{\text{bin}} = \left(\frac{\sigma_1 + 1}{2}, \dots, \frac{\sigma_N + 1}{2} \right) \quad (3.4.38)$$

Each argument of the Boolean function thus activates exactly one hidden neuron, which is specific to that argument. All we still have to do is find out how the specific state of the hidden layer of neurons can be utilized to generate the correct output value S .

For this purpose we must determine the appropriate strengths w_j of the synaptic connections between the hidden neurons and the output neuron, and its activation threshold ϑ (here we need only a single index at the synaptic couplings, because there is only one output neuron). Since each hidden neuron is mapped onto a unique input, i.e. a unique function argument $(\sigma_1, \dots, \sigma_N)$, we may set:

$$w_j = \begin{cases} +1, & \text{if } F(\sigma_1, \dots, \sigma_N) = \text{True} \\ -1, & \text{if } F(\sigma_1, \dots, \sigma_N) = \text{False} \end{cases} \quad (3.4.39)$$

where $(\sigma_1, \dots, \sigma_N)$ is referred to the specific input pattern which would activate exactly neuron j . Finally, by assigning the value $\vartheta = -\sum_j w_j$ to the activation threshold, the effective polarization potential at the output neuron is given according to (3.4.36) by

$$h = \sum_j w_j s_j - \vartheta = -\sum_{j \neq j_0} w_j + 2w_{j_0} + \sum_{j \neq j_0} w_j = 2w_{j_0} \quad (3.4.40)$$

where j_0 is the binary number denoting the function argument. In view of (3.4.39) the correct sign of the desired output value is obtained from the usual deterministic neural evolution equation $S = \text{sgn}(h)$. Because we have not made any restricting assumptions as to the nature of the Boolean function, this completes the proof that every Boolean function can be represented in this manner. \square

As is the case with many mathematical proofs, this sort of theorem only states that the representation is possible in principle. However, the scheme on which the proof is based may be totally useless in practice, because the number of required hidden neurons (2^N) is much too large. Moreover, one must realize that things can hardly be different, since we did not make any assumptions concerning the function F . The representation of arbitrary Boolean functions is a computationally “hard” problem (NP-complete). Here this finds its expression in the fact that an exponentially large number of hidden neurons are generally required.

The success of a three-layer perceptron with 2^N hidden neurons is always guaranteed, but in some case there is a simpler representation. Only in some special cases it is possible to succeed with far fewer neurons in the hidden layer than 2^N . Boolean functions which can be represented by a number of neurons that is lower than 2^N have been called NERFs (*Network Efficiently Representable Functions*). We have already studied one example of such a function in the previous sections, namely the exclusive-OR (XOR) function. The XOR

function has two logical arguments, but can be represented on a network containing only 2, instead of 4, hidden neurons. This is possible for the XOR due to its simple structure (it can be easily decomposed in AND, OR and NOT operations), but in general this is not the case for whatever function we are trying to reproduce. This most economical representation of the XOR, shown in Fig. 3.18, is compared in Fig. 3.21 to the standard representation involving four hidden neurons.

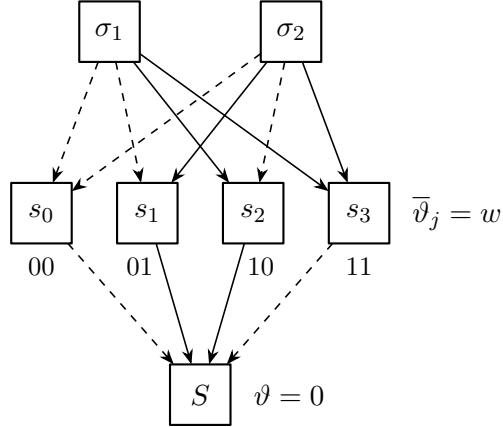


Figure 3.21: Standard representation of the XOR function by a three-layer perceptron with four hidden neurons. Dashed connections indicate negative couplings, while solid ones indicate positive couplings.

An entirely different question is whether a Boolean function of N arguments can be “learned” by a neural network in an amount of time that only grows polynomially with N . This question has been studied in a somewhat more general context, where the learning protocol allowed for two different levels: (1) Examples of argument vectors for which the function is True are given at random (“positive” examples); (2) for any chosen argument vector the function value is provided.

A related question of practical interest is whether, and under what conditions, the function can be learned from a few examples of the mapping: argument \rightarrow function value. This is commonly called the problem of *generalization*. Ideally, the network should be able to generalize correctly from input–output relations that were included in the learning set to those that were not.

3.4.4 Continuous functions

Another practically interesting application of multilayered neural networks is the prediction of functions which are known only at a certain number of discrete points. Neural networks are intrinsically nonlinear systems, and it is worthwhile investigating their forecasting capabilities. Extensive studies have been performed by Lapedes and Farber, who were able to show that *two hidden layers suffice for the representation of arbitrary, “reasonable” functions of any number of continuous arguments*. Actually, it can be shown that even a single layer of hidden neurons has sufficient flexibility to represent any continuous function. However, the proof, which is based on a general theorem of Kolmogorov concerning the representation of functions of several variables, is rather formal and does not guarantee the existence of a reasonable representation in practice, in contrast to the constructive proof that can be given for networks with two hidden layers.

The nonrigorous but constructive proof of Lapedes is based on the fact that *all continuous functions can be expressed as superpositions of simple, localized “bump” functions*. We shall

3 Neural networks

show by construction that a perceptron with a single hidden layer can generate such a function in one dimension, and that two hidden layers are sufficient in any higher dimension. As noted above, this is not the minimal number of required hidden layers, but this is not essential since the back-propagation algorithm works for any number of hidden layers.

We assume that the hidden layers of the network are constructed from analog-valued neurons with output values between 0 and 1, and we use the Fermi function $f(x) = 1/(1 + e^{-2\beta x})$ (3.2.57) to describe post-synaptic response. In order to be able to represent arbitrary continuous functions we allow the output neurons to assume any real value; so we use the linear function $f(x) = x$ for the response of the neurons in the output layer. The network is accordingly described by (3.4.25–3.4.27), except for the first part of (3.4.27), which is replaced by a linear relation:

$$\bar{s}_k = f(\bar{h}_k), \quad \bar{h}_k = \sum_{l=1}^{N_{\text{in}}} \bar{w}_{kl} \sigma_l - \bar{\vartheta}_k \quad (3.4.41)$$

$$s_j = f(\bar{h}_j), \quad \bar{h}_j = \sum_{k=1}^{N_{\text{hid},1}} \bar{w}_{jk} \bar{s}_k - \bar{\vartheta}_j \quad (3.4.42)$$

$$S_i = h_i, \quad h_i = \sum_{j=1}^{N_{\text{hid},2}} w_{ij} s_j - \vartheta_i \quad (3.4.43)$$

We begin with functions of a one-dimensional variable x . The function

$$g(x) = f(x) - f(x - c) \quad (3.4.44)$$

where $f(x)$ is the Fermi function and $c > 0$, describes a “bump” of width c and normalized height 1, which is localized at the point $x = c/2$ (Fig. 3.22). By means of a linear mapping of the variable x we can transform (3.4.44) into a bump of arbitrary width and height, and move it to any desired location on the x axis (Fig. 3.23):

$$g(x) = \alpha [f(ax - c_1) - f(ax - c_2)] \quad (3.4.45)$$

This equation is just of the form obtainable by a combination of the two network equations (3.4.43) and (3.4.42). So with more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.

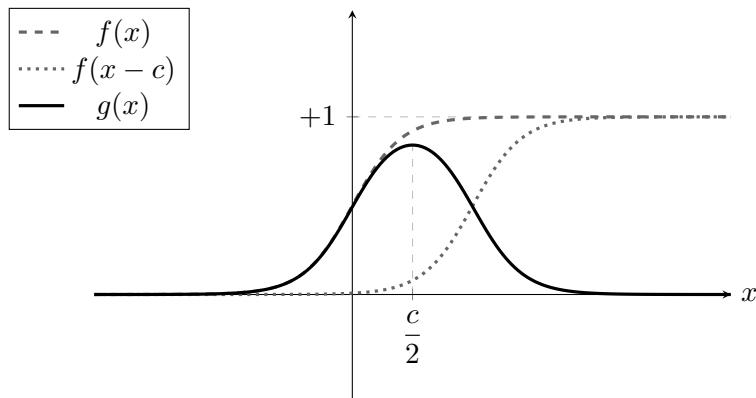


Figure 3.22: Equation (3.4.44).

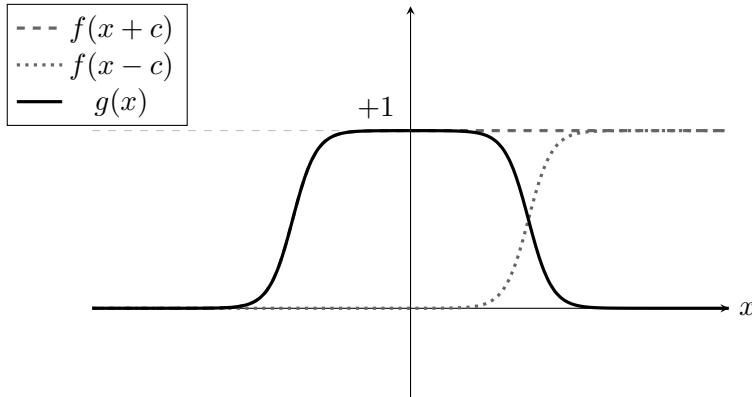


Figure 3.23: Equation (3.4.45) with $\alpha = 1$.

A second hidden layer allows the generation of multidimensional bump functions. At first glance it is tempting to create a two-dimensional bump function by multiplying two one-dimensional functions in the form $g(x, y) = g_1(x)g_2(y)$. However, this operation exceeds the capability of our neural network, because the synapses are summation, not multiplication, devices. In other words, signals can only be added, but never multiplied, at a synapse. This problem may be resolved in the following way. In two dimensions the one-dimensional bump function represents an infinitely extended, straight “ridge”. The sum of two such functions running in two different directions (x and y) in the two-dimensional plane, i.e.:

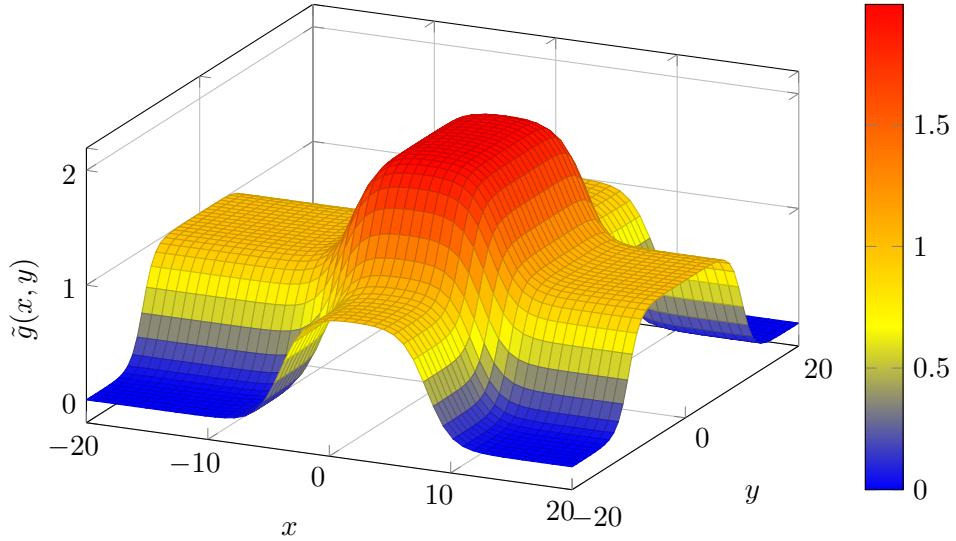
$$\tilde{g}(x, y) = g_1(x) + g_2(y) = \alpha [f(ax - b) - f(ax - c) + f(ay - d) - f(ay - e)] \quad (3.4.46)$$

reaches its highest value (about 2α) at the center of the intersection between the two ridges. But this is still not enough because the function obtained is not yet sufficiently localized (see Fig. 3.24a). Therefore, if we choose α as a very large number and introduce an activation threshold between α and 2α at the neurons in the second hidden layer, then the function $f(x)$, when applied to the right-hand side of (3.4.46), suppresses all parts of the intersecting bump lines except in the immediate vicinity of the center of the intersection. For $\alpha \gg 1$ we hence obtain in

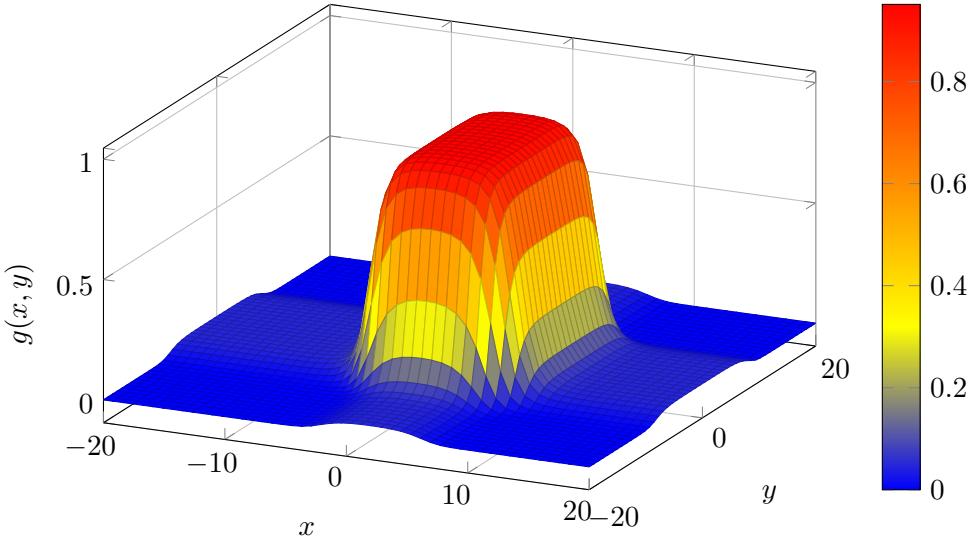
$$\begin{aligned} g(x, y) &= f\left\{\tilde{g}(x, y) - 1.5\alpha\right\} = \\ &= f\left\{\alpha[f(ax - b) - f(ax - c) + f(ay - d) - f(ay - e) - 1.5]\right\} \quad (3.4.47) \end{aligned}$$

a suitable bump function for the representation of functions in two dimensions. In simple terms, we have essentially stretched the graph near the pseudo-bump and we have “cut away” all the part of the surface below a certain level in order to shrink less pronounced contributions (see Fig. 3.24b). By adding further contributions to the argument of the outer Fermi function in (3.4.47) we can easily generalize the construction to any number of dimensions.

Once again the question may be posed whether this representation of continuous functions in terms of localized bump functions can be learned by the gradient method with error back-propagation. It is also unclear whether it is the best way of representing a given function on the neural network.



(a) Pseudo-bump generated by two intersecting ridges.



(b) The true bump in two dimensions.

Figure 3.24: These specific examples are taken from [\[1\]](#).

3.4.5 Generalization and fitting

Up to now we have mostly discussed how a feed-forward, layered neural network (perceptron) can learn to represent given input–output relations, such as the optimal reaction to an external stimulus, by a suitable choice of synapses and activation thresholds. If this were all there is, the neural network would only act as a convenient storage and recall device for known information, similar to the associative memory networks discussed at the beginning. The deeper intention is, of course, to use the network, after completion of the training phase, to process also inputs that were never learned. In other words, one would like to know whether the neural network can *generalize* the acquired knowledge in a meaningful way. Experience has shown that networks often succeed in this task, but not always, and sometimes only to a certain degree.

While the gradient-learning algorithm with error back-propagation is a practical method

of properly choosing the synaptic weights and thresholds of neurons, it provides no insight into the problem of how to choose the network architecture that is appropriate for the solution of a given problem. How many hidden layers are needed and how many neurons should be contained in each layer? If the number of hidden neurons is too small, no choice of the synapses may yield the accurate mapping between input and output, and the network will fail in the learning stage. If the number is too large, many different solutions will exist, most of which will not result in the ability to generalize correctly for new input data, and the network will usually fail in the operational stage. Intuitively it is clear that the ability of a network to generalize from learned examples must decrease with a rising number of hidden neurons. *The smallest network that can perform a certain task must also have the greatest ability to generalize.*

In the previous section we have demonstrated that a multilayered perceptron with one hidden layer is essentially able to learn and then to reproduce whatever continuous function of any number of continuous arguments. But what does it mean, in practice, that our feed-forward, layered network is capable of representing a continuous function? The basic idea behind it all is that we know a certain number of points with abscissa value x_i and ordinate y_i , and we ask the network to automatically generate new pairs (x'_j, y'_j) that are distributed on a two-dimensional plane according to the same function that best approximates the starting (x_i, y_i) pairs. This is exactly what happens in interpolation and fitting procedures, by means of which we try to reconstruct the $x \mapsto y = f(x)$ mapping of the initial points, so that we can then easily extract new points from that distribution without having to “act experimentally”.

A possible way to approximate the distribution of those points (x_i, y_i) , whence a method of obtaining new samples distributed in the same way, is the **interpolation**. A curve is said to be an interpolating curve if it passes *exactly* through all points of the initial dataset (Fig. 3.25). In particular, if we have N points, the simplest way to ensure the passage through all data is to use a polynomial $P_{N-1}(x)$ of order $N - 1$. At a first glance, this might seem like the best solution, but in the regions between the initial x_i the polynomial typically assumes completely spurious values because of its oscillatory nature, and so in the extrapolation phase we actually lose control if we are not exactly centered on those points. Moreover, if we move outside the range of the initial dataset, the polynomial diverges dramatically, and so we cannot expect it to provide good estimates in those regions. These two problems lead to the conclusion that in most of cases it is not a good idea to use interpolation to extrapolate new points.

For these reasons, a valid alternative is to resort to **fitting** techniques, which typically try to approximate the initial dataset with a polynomial having a significant lower order, $n < N$ (as in the linear regression, for instance). In curve fitting problems, the constraint that the interpolant has to go exactly through the data points is clearly relaxed, thus allowing us to keep the two problems mentioned above under control. Now it is only required to approach the data points as closely as possible (within some other constraints); this requires parameterizing the potential interpolants and having some way of measuring the error (in the simplest case this leads to least squares approximation).

The function

$$D(n) = \sum_{i=1}^N [y_i - P_{n-1}(x_i)]^2 \quad (3.4.48)$$

gives a sort of discrepancy between the initial dataset and the chosen best approximation $P_{n-1}(x)$. Its behavior with n is shown in Figure 3.26. For small values of n the discrepancy is naturally high, but then, beyond a certain scale, it starts flattening and settling at small values. Finally, for $n = N$ the deviation D goes exactly to 0 because in that case the

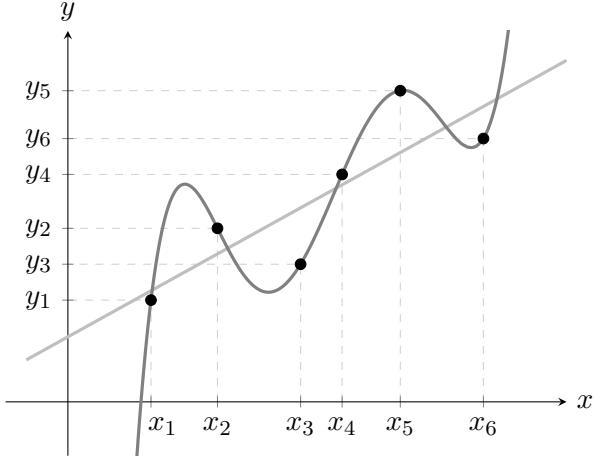


Figure 3.25: Polynomial interpolation of order 5 ($P_5(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$) and linear regression (fitting).

polynomial becomes an interpolation ($P_{n-1}(x) = P_{N-1}(x)$), meaning that it passes exactly through the initial points ($y_i = P_{N-1}(x_i)$). Typically the optimal fitting function is given by values of $n = \bar{n}$ near the region in which D starts oscillating at small values, which goes as $\bar{n} \sim \sqrt{N}$.

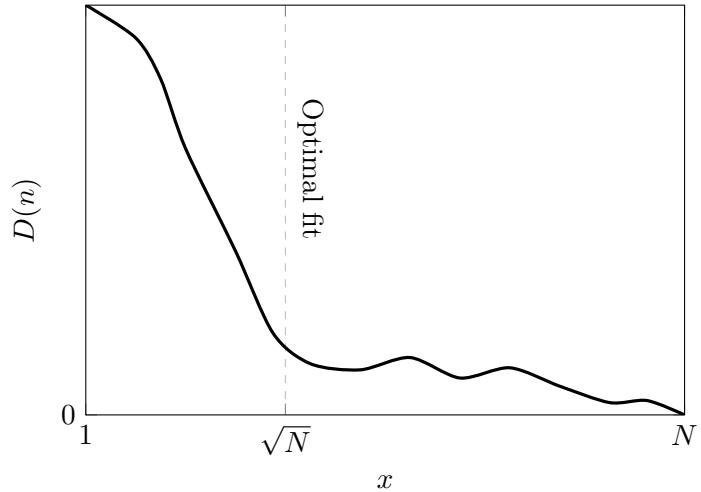


Figure 3.26

Fitting is exactly what we want to achieve for our neural network. In fact, all this mathematical structure is implemented in our perceptron in the following way. Let's focus momentarily on a class of functions like $f(x; \alpha, \beta) = \alpha e^{\beta x}$, for example, that we want to be learned from our architecture. During the training phase with the error back-propagation algorithm we must provide the network with a certain number of input values σ_k^μ , where the index μ refers to specific values of the continuous parameters on which the function $f(x; \alpha, \beta)$ depends, i.e. many different couples (α_μ, β_μ) in our case, while the index k characterizes the initial dataset. Furthermore, for each of these patterns we must also give the desired output coordinates ζ_i^μ . In other words, in the learning procedure we provide

the network the grids

$$x_1, x_2, \dots, x_{N_{\text{in}}} \quad (3.4.49)$$

$$x'_1, x'_2, \dots, x'_{N_{\text{out}}} \quad (3.4.50)$$

as input neurons and desired outputs respectively, i.e. $\sigma_k^\mu = x_k$ and $\zeta_i^\mu = x'_i$, $\mu = 1, \dots, p$ (these two grids are the new positions in the Cartesian plane we are playing with; hence they never change). Thus the input neurons take the values x_k of the initial dataset and the desired outputs take the values x'_i in correspondence of which we want to extrapolate new points. Then, once the net has been trained, we can finally use it to predict new points! This means that we choose a specific function $f(x; \alpha, \beta)$ from the general family (i.e. specific values for the parameters α and β), we provide the $y_k = f(x_k; \alpha, \beta)$ as input neurons and the network will be able to give an approximation of the ordinates $y'_i = f(x'_i; \alpha, \beta)$ which would correspond to the output grid. In this sense, the perceptron has learned the meaning of an exponential, and has done so through a fitting procedure (from (x_k, y_k) to the new (x'_i, y'_i)).

If we use a too small number of hidden neurons we are not able, already in the training phase, to obtain a correct mapping into the desired output. On the other hand, if we build a perceptron with too many layers we are essentially *overfitting* the initial dataset, since the mapping starts becoming an interpolation. If this happens, then we lose the network capability to generalize, i.e. to provide a correct answer even if fed with an input which is slightly different from the one it was trained with.

3.5 Boltzmann machines

The power of “hidden” neurons, i.e. neurons that do not directly communicate with the outside world, to perform complex tasks has become apparent in the case of layered feed-forward networks (perceptrons), discussed in Section 3.4. On the other hand, the theoretical treatment of such neural networks with strongly asymmetric synapses is much more difficult than that of symmetrically connected networks, developed in Sec. 3.2. It is therefore tempting to combine these two concepts, i.e. to study neural networks exhibiting hidden neurons and symmetric synapses ($w_{ik} = w_{ki}$).

The associative memory models studied in Sections 3.2.1–3.2.7 did not contain any hidden neurons, and we didn’t have any differentiation between input and output neurons; they had all the same role. The patterns to be stored were imprinted on all N neurons of the network, the patterns to be recalled were presented to all these neurons, and the recovered patterns were exhibited by the complete network, and not only by a part of it. Even with the most powerful learning algorithms such a network can only store at most $p = 2N$ patterns. Hinton and Sejnowski have therefore proposed adding hidden neurons to the stochastic Hopfield network which operate according to the principles explained in Sec. 3.2.6. Such “mixed” networks form general computing machines based on stochastic computational rules, and have been termed **Boltzmann machines** as opposed to the computers based on von Neumann’s principles of digital computing. The individual elements of a Boltzmann machine can take one of the two values $s_i = \pm 1$, assuming the positive value with probability $f(h_i)$, where as usual $h_i = \sum_k w_{ik} s_k$ and $f(h)$ is the Fermi function (3.2.57).

We learned in Sec. 3.2.4 that the dynamics of such a network can be described by the “energy” function

$$E[s] = -\frac{1}{2} \sum_{i,k} w_{ik} s_i s_k \quad (3.5.1)$$

3 Neural networks

The minima of this function correspond to stable configurations of the network (both global minima, associated to stored patterns, and false local minima). The concept of the Boltzmann machine stipulates that the neural network is first operated at a high temperature, which is gradually lowered until the network is trapped in an equilibrium configuration around a single minimum of the energy function. Averages, like $\langle s_i \rangle$ or $\langle s_i s_k \rangle$, are then easily computed by temporal sampling of configurations.

By virtue of the symmetry of the synapses it is possible to derive many general results concerning the properties of Boltzmann machines by the methods of statistical physics (ensembles). In particular the probability $P[s]$ of finding the network in a given state $\{s_i\}$

$$P[s] = \frac{1}{Z} e^{-\beta E[s]} \quad (3.5.2)$$

and the partition function Z of the network

$$Z = \sum_{[s]} e^{-\beta E[s]} \quad (3.5.3)$$

(sum over all possible configurations) will play a special role in the following considerations.

According to the concept of the Boltzmann machine we distinguish between *hidden* and *peripheral neurons* (Fig. 3.27). The latter may be further subdivided into input neurons, which receive information from outside, and output neurons, which communicate the result of the “computation”. E.g., in order to represent the exclusive-OR function, we would need three peripheral neurons, two (input) neurons to enter the two logical arguments and one (output) neuron to communicate the function value. Another characteristic task, which requires a division into input and output neurons, is the completion of patterns that are only partially known, e.g. if a stored picture of a human face is to be reconstructed from its left half. For such tasks one uses the term *heteroassociation* as opposed to the concept of autoassociation, where the complete, but maybe incorrect, pattern is presented to the network.

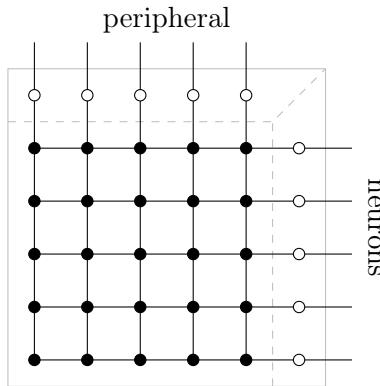


Figure 3.27: Schematic view of a network with hidden (black dots) and peripheral neurons (open circles). In principle, all neurons can be connected to all neurons.

Before continuing with the discussion, it is necessary to introduce some concepts of information theory¹⁵ which will then be exploited in the following.

¹⁵Information theory is the scientific study of the quantification, storage, and communication of information.

3.5.1 Information theory

In information theory the *entropy* is a quantity that, roughly speaking, measures how “interesting” or “surprising” a set of responses is. Suppose that we are given a set of neural responses. If each response is identical, or if only a few different responses appear, we might conclude that this data set is relatively uninteresting. A more interesting set might show a larger range of different responses, perhaps in a highly irregular and unpredictable sequence. How can we quantify this intuitive notion of an interesting set of responses?

The methods we discuss are based on the probabilities $P(r)$ of observing a certain response labeled r . The most widely used measure of entropy, due to Shannon, expresses the “surprise” associated with seeing a certain response r as a function of the probability of getting that response, $h[P(r)]$, and quantifies the entropy as the average of $h[P(r)]$ over all possible responses. The function $h[P(r)]$, which acts as a measure of surprise, is chosen to satisfy a number of conditions. First, $h[P(r)]$ should be a decreasing function of $P(r)$ because low probability responses are more surprising than high probability responses. Further, the surprise measure for a response that consists of two independent spike counts should be the sum of the measures for each spike count separately. This assures that the entropy and information measures we ultimately obtain will be additive for independent sources. Suppose we record responses r_1 and r_2 from two neurons that respond independently of each other. Because the responses are independent, the probability of getting this pair of responses is the product of their individual probabilities, $P(r_1)P(r_2)$, so the additivity condition requires that

$$h[P(r_1)] + h[P(r_2)] = h[P(r_1)P(r_2)] \quad (3.5.4)$$

The logarithm is the only function that satisfies such an identity for all P . Thus, it only remains to decide what base to use for the logarithm. By convention, base 2 logarithms are used so that information can be compared easily with results for binary systems. To indicate that the base 2 logarithm is being used, information is reported in units of “bits”, with

$$h[P(r)] = -\log_2 P(r) \quad (3.5.5)$$

The minus sign makes h a decreasing function of its argument, as required. Note that information is really a dimensionless number. The bit, like the radian for angles, is not a dimensional unit but a reminder that a particular system is being used.

Expression (3.5.5) quantifies the surprise or unpredictability associated with a particular response. **Shannon’s entropy** is just this measure averaged over all responses

$$H = -\sum_r P(r) \log_2 P(r) \quad (3.5.6)$$

In the sum that determines the entropy, the factor $h = -\log_2 P(r)$ is multiplied by the probability that the response r occurs. Responses with extremely low probabilities may contribute little to the total entropy, despite having large h values, because they occur so rarely. In the limit when $P(r) \rightarrow 0$, $h \rightarrow \infty$, but an event that does not occur does not contribute to the entropy because the problematic expression $-0 \log_2 0$ is evaluated as $-\varepsilon \log_2 \varepsilon$ in the limit $\varepsilon \rightarrow 0$, which is 0. Very high probability responses also contribute little because they have $h \approx 0$. The responses that contribute most to the entropy have high enough probabilities so that they appear with a fair frequency, but not high enough to make h too small.

Computing the entropy in some simple cases helps provide a feel for what it measures. First, imagine the least interesting situation: when a neuron responds every time by firing

at the same rate. In this case, all of the probabilities $P(r)$ are 0, except for one of them, which is 1. This means that every term in the sum of equation (3.5.6) is 0 because either $P(r) = 0$ or $\log_2 1 = 0$. Thus, a set of identical responses has zero entropy (this is trivial).

Next, imagine that the neuron responds in only two possible ways, either with rate r_+ or r_- . In this case, there are only two nonzero terms in equation (3.5.6), and, using the fact that $P(r_-) = 1 - P(r_+)$, the entropy is

$$H = -[1 - P(r_+)] \log_2 [1 - P(r_+)] - P(r_+) \log_2 P(r_+) \quad (3.5.7)$$

This entropy, plotted in Figure 3.28, takes its maximum value of 1 bit when $P(r_-) = P(r_+) = 1/2$ (maximal uncertainty of the outcome). Thus, a code consisting of two equally likely responses has one bit of entropy.

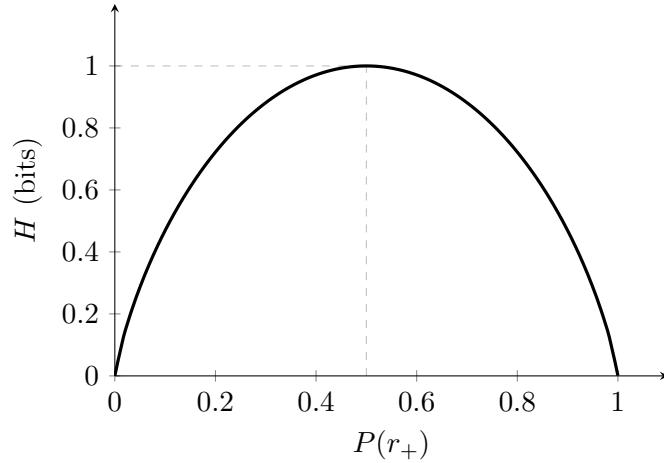


Figure 3.28: The entropy of a binary code.

3.5.2 Mutual information

To convey information about a set of stimuli, neural responses must be different for different stimuli. Entropy is a measure of response variability, but it does not tell us anything about the source of that variability. A neuron can provide information about a stimulus only if its response variability is correlated with changes in that stimulus, rather than being purely random or correlated with other unrelated factors. One way to determine whether response variability is correlated with stimulus variability is to compare the responses obtained using a different stimulus on every trial with those measured in trials involving repeated presentations of the same stimulus. Responses that are informative about the identity of the stimulus should exhibit larger variability for trials involving different stimuli than for trials that use the same stimulus repetitively. In other words, we prefer neurons that provide always the same response to the same stimulus, thing that in nature could not happen for a variety of reasons (from the non-perfection of the neuron itself, a different chemical transmission between subsequent stimuli, etc.), because we want the system to behave in a predictable way. Mutual information is an entropy-based measure related to this idea.

The **mutual information** is the difference between the total response entropy and the average response entropy on trials that involve repetitive presentation of the same stimulus. Subtracting the entropy when the stimulus does not change removes from the total entropy the contribution from response variability that is not associated with the identity of the

stimulus. When the responses are characterized by a spike-count rate, the total response entropy is given by equation (3.5.6). The entropy of the responses evoked by repeated presentations of a given stimulus s is computed using the conditional probability $P(r|s)$, i.e the probability of a response r given that stimulus s was presented, instead of the response probability $P(r)$ in equation (3.5.6). The entropy of the responses to a given stimulus is thus

$$H_s = - \sum_r P(r|s) \log_2 P(r|s) \quad (3.5.8)$$

If we average this quantity over all the stimuli, we obtain a quantity called the *noise entropy*:

$$H_{\text{noise}} = \sum_s P(s) H_s = - \sum_{s,r} P(s) P(r|s) \log_2 P(r|s) \quad (3.5.9)$$

This is the entropy associated with that part of the response variability that is not due to changes in the stimulus, but arises from other sources. Then, as said before, the mutual information is obtained by subtracting the noise entropy from the full response entropy, which from previous equations gives

$$I_m = H - H_{\text{noise}} = - \sum_r P(r) \log_2 P(r) + \sum_{s,r} P(s) P(r|s) \log_2 P(r|s) \quad (3.5.10)$$

The probability of a response r is naturally related to the conditional probability $P(r|s)$ and the probability $P(s)$ that stimulus s is presented by the identity

$$P(r) = \sum_s P(s) P(r|s) \quad (3.5.11)$$

Using this, and writing the difference of the two logarithms in equation (3.5.10) as the logarithm of the ratio of their arguments, we can rewrite the mutual information as

$$I_m = \sum_{s,r} P(s) P(r|s) \log_2 \left[\frac{P(r|s)}{P(r)} \right] \quad (3.5.12)$$

Moreover, recall that

$$P(r, s) = P(s) P(r|s) = P(r) P(s|r) \quad (3.5.13)$$

is the *joint probability* of stimulus s appearing and response r being evoked (sometimes written as $P(r \cap s)$), and must be symmetric due to Bayes theorem¹⁶. Hence this equation can be used to derive yet another form for the mutual information:

$$I_m = \sum_{s,r} P(r, s) \log_2 \left[\frac{P(r, s)}{P(r) P(s)} \right] \quad (3.5.15)$$

This equation reveals that the mutual information is symmetric with respect to interchange of s and r , which means that the mutual information that a set of responses conveys about a set of stimuli is identical to the mutual information that the set of stimuli conveys about the responses.

¹⁶Formally, with two events A and B it states that

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \quad (3.5.14)$$

with $P(B) \neq 0$.

3 Neural networks

To provide some concrete examples, we compute the mutual information for a few simple cases. First, suppose that the responses of the neuron are completely unaffected by the identity of the stimulus. In this case, $P(r, s) = P(r)P(s)$ (or $P(r|s) = P(r)$), and from equation (3.5.15) (or (3.5.12)) it follows immediately that $I_m = 0$ because $\log_2 1 = 0$. At the other extreme, suppose that each stimulus s produces a unique and distinct response r_s . Then, $P(r = r_s) = P(s)$ and $P(r|s)$ is 1 if $r = r_s$ and 0 otherwise. This causes the sum over r in equation (3.5.12) to collapse to just one term, and the mutual information becomes

$$I_m = \sum_s P(s) \log_2 \left[\frac{1}{P(r_s)} \right] = - \sum_s P(s) \log_2 P(s) \quad (3.5.16)$$

The last expression, which follows from the fact that $P(r_s) = P(s)$, is the entropy of the stimulus (remember the definition of Shannon's entropy (3.5.6)). Thus, with no variability and a one-to-one map from stimulus to response, the mutual information is equal to the *full stimulus entropy*.

Finally, imagine that there are only two possible stimulus values, which we label + and −, and that the neuron responds with just two rates, r_+ and r_- . We associate the response r_+ with the + stimulus, and the response r_- with the − stimulus. But the encoding is not perfect: the probability of an incorrect response is P_X , meaning that for the correct responses $P(r_+|+) = P(r_-|-) = 1 - P_X$, and for the incorrect responses $P(r_+|-) = P(r_-|+) = P_X$. We assume that the two stimuli are presented with equal probability ($P(+) = P(-) = 1/2$) so that $P(r_+) = P(r_-) = 1/2$, which, from equation (3.5.7), makes the full response entropy 1 bit. The noise entropy is

$$\begin{aligned} H_{\text{noise}} &= - \sum_{\substack{s=+, - \\ r=r_+, r_-}} P(s) P(r|s) \log_2 P(r|s) \\ &= - \left[P(+) P(r_+|+) \log_2 P(r_+|+) + P(+) P(r_-|+) \log_2 P(r_-|+) + \right. \\ &\quad \left. + P(-) P(r_+|-) \log_2 P(r_+|-) + P(-) P(r_-|-) \log_2 P(r_-|-) \right] \\ &= -(1 - P_X) \log_2 (1 - P_X) - P_X \log_2 P_X \end{aligned} \quad (3.5.17)$$

Thus, the mutual information is

$$I_m = 1 + (1 - P_X) \log_2 (1 - P_X) + P_X \log_2 P_X \quad (3.5.18)$$

This is plotted in Figure 3.29. When the encoding is error-free ($P_X = 0$), the mutual information is 1 bit, which is equal to both the full response entropy and the stimulus entropy; i.e. the noise is null. When the encoding is completely random ($P_X = 1/2$), the mutual information goes to 0.

The mutual information is related to a measure used in statistics called the **Kullback–Leibler (KL) divergence**. The KL divergence between one probability distribution $P(r)$ and another distribution $Q(r)$ is

$$D_{\text{KL}}(P, Q) = \sum_r P(r) \log_2 \left[\frac{P(r)}{Q(r)} \right]$$

$$(3.5.19)$$

which is a measure of how one probability distribution is different from a second, reference probability distribution. The KL divergence has a property normally associated with a distance measure, $D_{\text{KL}}(P, Q) \geq 0$ with equality if and only if $P = Q$. However, unlike

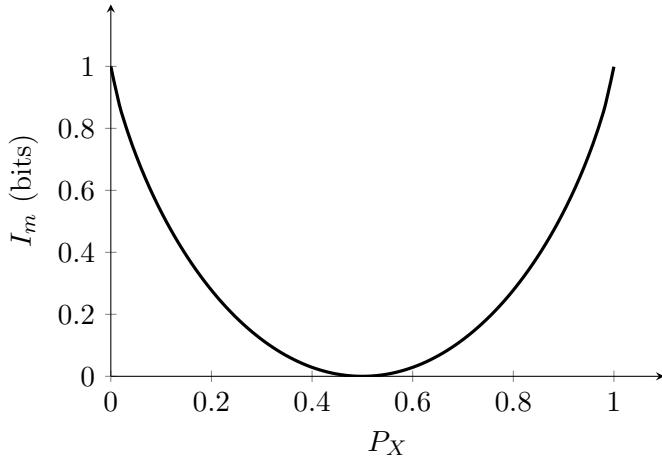


Figure 3.29: The mutual information for a binary encoding of a binary stimulus.

a distance, it is not symmetric with respect to interchange of P and Q . Comparing the definition (3.5.19) with equation (3.5.15), we see that the mutual information is the Kullback–Leibler divergence between the distributions $P(r, s)$ and $P(r)P(s)$.

$$I_m = D_{\text{KL}}(P(r, s) \parallel P(r)P(s)) \quad (3.5.20)$$

If the stimulus and the response were independent of one another, $P(r, s)$ would be equal to $P(r)P(s)$. Accordingly, the KL distance would be exactly 0. The fact that $D_{\text{KL}}(P, Q) \geq 0$ proves that the mutual information cannot be negative ($I_m \geq 0$). In addition, it can never be larger than either the full response entropy or the entropy of the stimulus set.

3.5.3 The ‘‘Boltzmann’’ learning rule

At this point we are left in the determination of the appropriate symmetric synaptic connections w_{ik} in order for the Boltzmann machine to behave according to our wishes. In particular, we will discuss a learning algorithm which is based on the concept of entropy. As a preparation we first have to introduce an appropriate notation to distinguish between the hidden and the peripheral neurons. We denote the collective states of the output neurons by an index α , those of the input neurons by an index β , and those of the hidden neurons by an index γ . The state of the complete network is then denoted by the index combination $\alpha\beta\gamma$, etc.

For a certain training set the probability distribution of encountering a specific input configuration β may be described by Q_β . This distribution is given from outside, independent of the network dynamics. If the conditional probability of finding a certain state α of the output neurons for a given input configuration β is denoted as $P_{\alpha|\beta}$, the true distribution of output states is given by

$$P_\alpha = \sum_\beta P_{\alpha|\beta} Q_\beta \quad (3.5.21)$$

which is essentially the same as (3.5.11). Similar notations may apply if the states of the hidden neurons are also included. The index to the right of a vertical bar always indicates the fixed configuration for a conditional probability. All probability distributions are assumed to be normalized to unity, e.g.:

$$\sum_\alpha P_{\alpha|\beta} = \sum_{\alpha,\gamma} P_{\alpha\gamma|\beta} = 1 \quad (3.5.22)$$

3 Neural networks

The conditional probability $P_{\alpha\gamma|\beta}$:

$$P_{\alpha\gamma|\beta} = \frac{1}{Z_{|\beta}} e^{-\frac{E_{\alpha\gamma\beta}}{T}} \quad (3.5.23)$$

is determined by the synaptic strengths w_{ik} of the network; in fact, according to the expression of the energy (3.5.2) we have

$$P_{\alpha|\beta} = \sum_{\gamma} P_{\alpha\gamma|\beta} = \frac{1}{Z_{|\beta}} \sum_{\gamma} e^{-\frac{E_{\alpha\gamma\beta}}{T}} \quad (3.5.24)$$

where the energy $E_{\alpha\gamma\beta}$ is given by (3.5.1), and $Z_{|\beta}$ is the partition function for fixed input configuration β :

$$Z_{|\beta} = \sum_{\alpha,\gamma} e^{-\frac{E_{\alpha\gamma\beta}}{T}} \quad (3.5.25)$$

Now, for symmetric neural networks without hidden neurons Hebb's rule provides a satisfactory solution of the question how the synaptic strengths w_{ik} should be determined so that the network is able to perform a specific task. In that case we started by requiring the vanishing of the so-called Hamming distance between the stored and presented patterns. However in the presence of hidden neurons Hebb's rule is no longer sufficient, since it does not tell us how to fix the synapses between peripheral and hidden neurons, and among the hidden neurons themselves. On the other hand, in the case of multilayered perceptrons we had found a successful algorithm on the basis of the gradient method with error back-propagation. In that second case we had defined an appropriate function D describing the deviation between the desired output ζ_i^{μ} and the actual output S_i^{μ} , and our task of finding the best couplings was fulfilled with the minimization of that function. Notice that in Boltzmann machines we are talking about stochastic networks; this means that for whatever choice of the couplings we cannot oblige our system to produce specific outcomes, i.e. specific values for each neuron, in response to a specific stimulus because these responses are randomly generated, according to a certain probability distribution $P_{\alpha|\beta}$ (the net is not deterministic). Nevertheless, $P_{\alpha|\beta}$ still remains a degree of freedom to be arbitrarily fixed for our purposes ($P_{\alpha|\beta}$ is what we obtain from a run of the net). That is, we can require that the synapses be such that the distribution of possible outcomes (i.e. responses) converges to a desired law of probability. Therefore, now the learning task becomes to choose the synapses in such a way that the conditional probability $P_{\alpha|\beta}$ takes on the desired value, which we denote by $Q_{\alpha|\beta}$. Note that the mapping is till stochastic, but we have chosen with our hand the behavior of the randomness source. But even by taking this shrewdness into account, the concepts of the gradient method cannot be directly applied to the Boltzmann machines, because the synaptic connections must be symmetric and are not forward oriented. In other words, the choice of the gradient method is not conveniently treated by the methods of statistical physics.

A completely new concept is required here. In particular, here we are saved by the fact that the principles of statistical physics can be applied to the Boltzmann machine. These are specifically designed to find that distribution which *minimizes the entropy* of a thermodynamic system. We therefore define the deviation function as follows

$$D = \sum_{\beta} Q_{\beta} \sum_{\alpha} Q_{\alpha|\beta} \ln \left(\frac{Q_{\alpha|\beta}}{P_{\alpha|\beta}} \right) \quad (3.5.26)$$

In the context of information theory D is essentially the Kullback–Leibler divergence (3.5.19) resulting from a measurement of the distribution $P_{\alpha|\beta}$, with the inclusion of an average over

the possible input stimuli β . This distance vanishes if the measured distribution coincides with the expected distribution $Q_{\alpha|\beta}$. As said before, it is easy to show that the expression of D is positive (semi-) definite and reaches its minimum value for $P_{\alpha|\beta} = Q_{\alpha|\beta}$. The proof is based on the relation¹⁷

$$\ln x \geq 1 - \frac{1}{x} \quad (3.5.30)$$

and makes use of the normalization of the probability distribution:

$$\begin{aligned} D &\geq \sum_{\beta} Q_{\beta} \sum_{\alpha} Q_{\alpha|\beta} \left(1 - \frac{P_{\alpha|\beta}}{Q_{\alpha|\beta}}\right) = \\ &= \sum_{\alpha} \sum_{\beta} Q_{\beta} (Q_{\alpha|\beta} - P_{\alpha|\beta}) \\ &= \sum_{\alpha} (Q_{\alpha} - P_{\alpha}) \\ &= 1 - 1 \\ &= 0 \end{aligned} \quad (3.5.31)$$

We now apply the gradient method to the deviation function (3.5.26) and adjust the synaptic strengths according to the rule

$$\delta w_{ik} = -\varepsilon \frac{\partial D}{\partial w_{ik}} = \varepsilon \sum_{\alpha, \beta} Q_{\beta} \frac{Q_{\alpha|\beta}}{P_{\alpha|\beta}} \frac{\partial P_{\alpha|\beta}}{\partial w_{ik}} \quad (3.5.32)$$

Here we have taken into account that the desired conditional response probabilities $Q_{\alpha|\beta}$ do not depend on the values of the synaptic couplings w_{ik} . The essential point in the derivation is that the partial derivative of $P_{\alpha|\beta}$ with respect to the synaptic connection w_{ik} can be determined solely from the behavior of the two connected neurons i and k . For this purpose we express $P_{\alpha|\beta}$ explicitly in terms of the w_{ik} with the help of the previous expressions for the probability and the partition function:

$$P_{\alpha\gamma|\beta} = \frac{1}{Z_{|\beta}} e^{-\frac{E_{\alpha\gamma\beta}}{T}} \quad (3.5.33)$$

$$P_{\alpha|\beta} = \frac{1}{Z_{|\beta}} \sum_{\gamma} e^{-\frac{E_{\alpha\gamma\beta}}{T}} \quad (3.5.34)$$

$$Z_{|\beta} = \sum_{\alpha, \gamma} e^{-\frac{E_{\alpha\gamma\beta}}{T}} \quad (3.5.35)$$

In particular, the synaptic strengths enter the energy function

$$E_{\alpha\gamma\beta} = -\frac{1}{2} \sum_{i,k} w_{ik} [s_i s_k]_{\alpha\gamma\beta} \quad (3.5.36)$$

¹⁷This a consequence of the logarithm being a concave function: since the derivative of the logarithm ($1/x$) is monotonously decreasing, then the first-order Taylor approximation constitutes an upper bound

$$f(x) \leq f(y) + f'(y)(x - y) \quad (3.5.27)$$

Namely

$$\ln x \leq \ln y + \frac{1}{y}(x - y) \quad (3.5.28)$$

and setting $x = 1$ we obtain

$$\ln y \geq 1 - \frac{1}{y} \quad (3.5.29)$$

as stated at the beginning.

3 Neural networks

where $[s_i s_k]_{\alpha\gamma\beta}$ denotes the product $s_i s_k$ of states of the two neurons under consideration if the total network configuration is given by $\alpha\gamma\beta$, corresponding to the configuration α for the output neurons, γ for the hidden neurons and β for the input neurons. Note that this is equivalent to (3.5.1). Thus the partial derivative of $P_{\alpha|\beta}$ with respect to w_{ik} yields

$$\begin{aligned}\frac{\partial P_{\alpha|\beta}}{\partial w_{ik}} &= \frac{1}{2T Z_{|\beta}} \sum_{\gamma} e^{-\frac{E_{\alpha\gamma\beta}}{T}} [s_i s_k]_{\alpha\gamma\beta} - \frac{1}{2T Z_{|\beta}^2} \sum_{\alpha,\gamma} e^{-\frac{E_{\alpha\gamma\beta}}{T}} [s_i s_k]_{\alpha\gamma\beta} \sum_{\gamma'} e^{-\frac{E_{\alpha\gamma'\beta}}{T}} \\ &= \frac{1}{2T} \left(\sum_{\gamma} P_{\alpha\gamma|\beta} [s_i s_k]_{\alpha\gamma\beta} - \sum_{\alpha,\gamma} P_{\alpha\gamma|\beta} [s_i s_k]_{\alpha\gamma\beta} P_{\alpha|\beta} \right) \\ &= \frac{1}{2T} \left(\langle s_i s_k \rangle_{|\alpha\beta} - \langle s_i s_k \rangle_{|\beta} \right) P_{\alpha|\beta}\end{aligned}\quad (3.5.37)$$

where the indices $|\beta$ and $|\alpha\beta$ indicates that the average is performed in the presence of a fixed input state β or fixed input-output state $\alpha\beta$, respectively.

$$\langle s_i s_k \rangle_{|\beta} = \sum_{\alpha,\gamma} P_{\alpha\gamma|\beta} [s_i s_k]_{\alpha\gamma\beta} \quad (3.5.38)$$

$$\langle s_i s_k \rangle_{|\alpha\beta} = \frac{1}{P_{\alpha|\beta}} \sum_{\gamma} P_{\alpha\gamma|\beta} [s_i s_k]_{\alpha\gamma\beta} \quad (3.5.39)$$

Notice the connection between these two correlations:

$$\langle s_i s_k \rangle_{|\beta} = \sum_{\alpha} \frac{P_{\alpha|\beta}}{P_{\alpha|\beta}} \sum_{\gamma} P_{\alpha\gamma|\beta} [s_i s_k]_{\alpha\gamma\beta} = \sum_{\alpha} P_{\alpha|\beta} \langle s_i s_k \rangle_{|\alpha\beta} \quad (3.5.40)$$

The iterative adjustment of the synaptic couplings can now be derived from (3.5.32), considering the relation (3.5.22) for the normalization, giving the following result:

$$\begin{aligned}\delta w_{ik} &= \varepsilon \sum_{\alpha,\beta} Q_{\beta} \frac{Q_{\alpha|\beta}}{P_{\alpha|\beta}} \frac{\partial P_{\alpha|\beta}}{\partial w_{ik}} \\ &= \frac{\varepsilon}{2T} \sum_{\alpha,\beta} Q_{\beta} \frac{Q_{\alpha|\beta}}{P_{\alpha|\beta}} \left(\langle s_i s_k \rangle_{|\alpha\beta} - \langle s_i s_k \rangle_{|\beta} \right) P_{\alpha|\beta} \\ &= \frac{\varepsilon}{2T} \sum_{\beta} Q_{\beta} \left(\sum_{\alpha} Q_{\alpha|\beta} \langle s_i s_k \rangle_{|\alpha\beta} - \langle s_i s_k \rangle_{|\beta} \right)\end{aligned}\quad (3.5.41)$$

In principle the one-half factor can be absorbed in the displacement ε and we can simply write

$$\delta w_{ik} = \frac{\varepsilon}{T} \sum_{\beta} Q_{\beta} \left(\sum_{\alpha} Q_{\alpha|\beta} \langle s_i s_k \rangle_{|\alpha\beta} - \langle s_i s_k \rangle_{|\beta} \right) \quad (3.5.42)$$

This is the optimal way to build a Boltzmann machine, known as the *Boltzmann learning rule*. The first term in parentheses describes the correlation between the states of the two neurons i and k , under the condition that the state of the input neurons is kept fixed during the operation of the network and an average is taken over the output states with the desired probabilities $Q_{\alpha|\beta}$. The second term expresses that the unconditional correlation between the states of the two neurons is subtracted from this result. As already mentioned above, it is essential that δw_{ik} can be determined from observations of the states s_i and s_k of the two connected neurons alone. The first term is obtained as the average value of the

product $s_i s_k$ when the network evolves with fixed input and output states¹⁸. The synapses can therefore be adjusted on the basis of local observations, which greatly simplifies the network architecture, especially for large networks.

The associative memory networks studied in Section 3.2 correspond to the special case that all neurons are both input and output neurons, and that there are no hidden neurons. In this case we have $Q_{\alpha|\beta} = \delta_{\alpha\beta}$ for all memorized patterns σ_i^μ , since the network is supposed to return the stored pattern when it is presented as an input:

$$\delta w_{ik} = \frac{\varepsilon}{T} \sum_{\beta} Q_{\beta} \left(\langle s_i s_k \rangle_{|\beta\beta} - \langle s_i s_k \rangle_{|\beta} \right) \quad (3.5.43)$$

The first term in parentheses then yields precisely Hebb's rule (3.2.18) ($w_{ik} \rightarrow w_{ik} + \delta w_{ik}$), and the second term describes the intentional forgetting of patterns that have been invoked by random input. In this case the Boltzmann learning rule (3.5.42) reduces to the iterative learning algorithm (3.2.77):

$$\delta w_{ij} = \frac{\lambda}{N} \left(\sigma_i^{\beta} \sigma_k^{\beta} - s_i^{\infty} s_j^{\infty} \right) \quad (3.5.44)$$

In other words, (3.5.42) can be considered a generalization of the concept of repeated forgetting and relearning to symmetric neural networks with hidden neurons. For this reason, the two terms in (3.5.42) are sometimes called “wake” (where we fix both input and output) and “sleep” (fixed only the input) phases of the training procedure, or also “Hebbian” and “anti-Hebbian” terms, respectively.

The training process of a Boltzmann machine consists of a series of alternating steps. For each response to be learned, in the first step both input and output units are clamped, and the response of the network, after settling to thermal equilibrium, is measured. Subsequently the same procedure is repeated while letting the output units evolve freely (i.e. once we reached the statistical equilibrium we run the machine many time in order to compute the averages). According to the rule of “contrastive Hebbian synapses” (3.5.42) the couplings w_{ik} are updated using the difference of both measurements. This has to be done for all input patterns β , and usually many such training cycles are required. This may be computationally expensive since correlations of stochastically fluctuating quantities $\langle s_i s_k \rangle$ have to be measured.

3.5.3.1 Applications

Sejnowski et al. [12] have studied the performance of these Boltzmann machines, i.e. of symmetrically connected networks with hidden neurons, for a number of problems which also require the presence of hidden neurons if they are posed to networks of the perceptron type.

The first example is the already familiar exclusive-OR (XOR), which requires two input neurons and one output neuron. In Sec. 3.4.1 we have solved the XOR problem by including also two hidden neurons, which work as AND and OR gates separately. In multilayered perceptrons the connections are always between one layer and the next one. In the present context of Boltzmann machine, however, *all* four neurons are symmetrically connected by synapses, except that there is no direct synaptic connection between the two input neurons. The fact that now we can connect directly an input unit with an output unit allows us to reduce the number of hidden neurons to one (see Fig. 3.30). After 30 learning

¹⁸In electrical engineering this type of operation of an electrical circuit is called *clamping* of the input and output voltage.

sweeps over the training set with simultaneous gradual lowering of the temperature the rate of success reached 96 %. After 255 further rounds and continued cooling the network performed perfectly, within the limitations set by the presence of the remaining small thermal fluctuations.

Since the Boltzmann network operates stochastically, it does not converge to the same solution at every attempt, if there is more than one valid solution of the problem. In total, the network found eight different strategies to represent the XOR function which made use of the hidden neuron in different ways. One example for the values of the synaptic strengths and the threshold potentials is shown in Fig. 3.30.

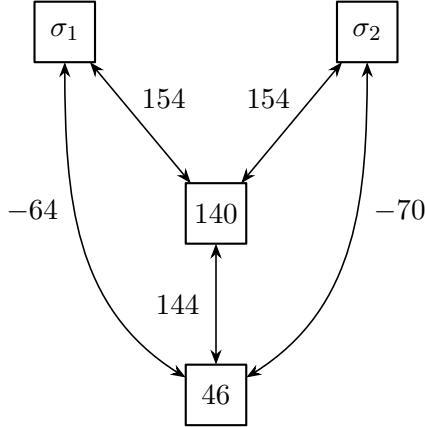


Figure 3.30: Graphical representation of the synaptic connections for a specific solution of the exclusive-OR (XOR) function (after [12]).

On the other hand, one of the weak points of the Boltzmann machine is that it is not very successive in recognizing patterns, because it evolves slowly and does not converge rapidly.

3.6 Restrictive Boltzmann machines

Nowadays the most used type of Boltzmann machines are the so-called **restricted Boltzmann machines** (RBMs). As their name implies, RBMs are a variant of Boltzmann machines, with the restriction that their neurons must form a bipartite graph: all peripheral neurons (sometimes also called “visible” neurons) are connected to all hidden and vice versa, but units belonging to the same layer are not possible, i.e. visible-visible and hidden-hidden connections are forbidden (Fig. 3.31). By contrast, “unrestricted” Boltzmann machines may have connections between hidden units. This restriction clearly reduces the number of free parameters and hence they allows for more efficient training algorithms than are available for the general class of Boltzmann machines.

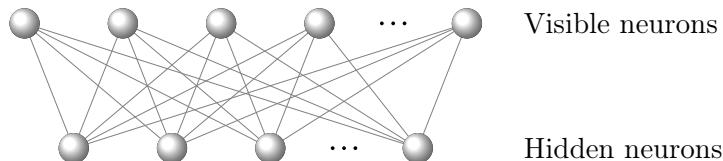


Figure 3.31: Schematic representation of a restricted Boltzmann machine.

In 2005 it was found that such restricted Boltzmann machines are most efficiently trained with a different technique with respect to the standard one, called *contrastive divergence*

algorithm. The idea behind using a different algorithm is that in Boltzmann machines we first need to reach the statistical equilibrium before computing the ensemble averages $\langle s_i s_k \rangle$; this means that the standard learning phase can take a long time to complete, and so it is not very efficient. Instead, for RBMs the way of training the net is the following. Roughly speaking, in a contrastive divergence algorithm scheme we start with a fixed input–output (visible neurons) grid of values $\mathbf{x} = (x_1, \dots, x_{N_{\text{vis}}})$ and we check (i.e. learn) the values $\mathbf{h} = (h_1, \dots, h_{N_{\text{hid}}})$ for the hidden neurons; this is basically the first term of equation (3.5.42). Then, from the values \mathbf{h} of the hidden neurons kept frozen, we get new values \mathbf{x}' for the visible neurons. Finally, we repeat again this procedure from the visible neurons \mathbf{x}' to new hidden neurons \mathbf{h}' . So the contrastive divergence algorithm works in alternating directions: forward ($\mathbf{x} \Rightarrow \mathbf{h}$), backward ($\mathbf{h} \Rightarrow \mathbf{x}'$) and again forward ($\mathbf{x}' \Rightarrow \mathbf{h}'$). The mathematical formula describing this algorithm is essentially similar to the standard (3.5.42), but the second term is substituted with the determination of \mathbf{h}' . In conclusion, this surely less time-consuming because we need not to reach the statistical equilibrium before updating the parameters.

Restricted Boltzmann machines can also be used in deep learning networks, term that has become ordinary in the early. In this sense we talk about “deep learning” for algorithms that uses multiple layers to progressively extract higher-level features from the raw input. In particular, in 2006 **deep belief networks** (DBNs) were introduced, which are a class of deep neural networks composed of multiple layers of hidden units, with connections between the layers but not between units within each layer. DBNs can be viewed as a composition of simple, unsupervised networks such as restricted Boltzmann machines, where each sub-network’s hidden layer serves as the visible layer for the next. In other words, a deep belief network is formed by stacking many RBMs one on top of the other (Fig. 3.32). This composition leads to a fast, layer-by-layer unsupervised training procedure, where contrastive divergence is applied to each sub-network in turn, starting from the “lowest” pair of layers (the lowest visible layer is a training set). Overall they are very efficient for two reasons: (1) the component RBMs can be trained separately by itself; (2) if we want to analyze many complicated data, probably characterized by specific features, with the first restricted Boltzmann machine we classify them in a certain way, while with the second in another. So we can have a *progressive* understanding, from rough to a more detailed one. Historically speaking, this has been the first idea of progressive learning.

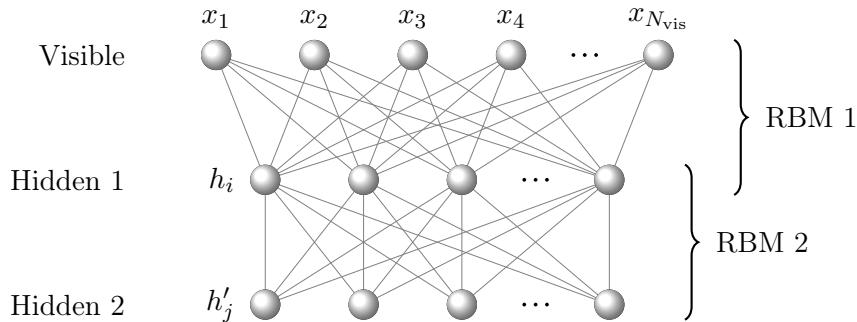


Figure 3.32: Schematic representation of a deep belief network.

CHAPTER 4

Machine learning techniques

Contents

4.1	Introduction to machine learning	128
4.1.1	Supervised/unsupervised learning	128
4.1.1.1	Supervised learning	128
4.1.1.2	Unsupervised learning	129
4.1.1.3	Semisupervised learning	130
4.1.1.4	Reinforcement learning	130
4.1.2	Batch and online learning	131
4.1.2.1	Batch learning	131
4.1.2.2	Online learning	132
4.1.3	Instance-based and model-based learning	132
4.1.3.1	Instance-based learning	133
4.1.3.2	Model-based learning	133
4.2	Testing and validating	133
4.3	Quality of data and preprocessing	135
4.4	California housing prices	136
4.4.1	Frame the problem	136
4.4.2	Select a performance measure	136
4.4.3	Get the data	137
4.4.4	Create a test set	140
4.4.5	Data visualization	142
4.4.6	Looking for correlations	144
4.4.7	Experimenting with attribute combinations	146
4.4.8	Data cleaning	147
4.4.9	Handling text and categorical attributes	148
4.4.10	Feature scaling	149
4.4.11	Transformation pipelines	150
4.4.12	Select and train a model	150
4.4.13	Better evaluation using cross-validation	152
4.4.14	Fine-tune your model	154
4.4.15	Analyze the best models and their errors	156
4.4.16	Evaluate your system on the test set	156
4.5	Training models	157
4.5.1	Linear regression	157
4.5.2	Gradient descent	161
4.5.2.1	Stochastic gradient descent	163
4.5.2.2	Mini-batch gradient descent	164
4.5.3	Polynomial regression	165
4.5.4	Learning curves	166

4.5.4.1	On the nature of errors	168
4.5.5	Regularized linear models	169
4.5.5.1	Ridge regression	169
4.5.5.2	Lasso regression	170
4.5.5.3	Early stopping	170
4.5.6	Logistic regression	171
4.5.6.1	Estimating probabilities	172
4.5.6.2	Training and cost function	172
4.5.6.3	Decision boundaries	173
4.5.6.4	Softmax regression	175
4.6	Support vector machines	176
4.6.1	Linear SVM classification	176
4.6.1.1	Soft margin classification	177
4.6.2	Nonlinear SVM classification	178
4.6.2.1	Polynomial kernel	179
4.6.3	Mathematical formulation	180
4.6.3.1	Decision function and predictions	180
4.6.3.2	Training objective	181
4.6.3.3	The dual problem	182
4.6.4	Kernelized SVMs	183
4.7	Dimensionality reduction	184
4.7.1	The curse of dimensionality	184
4.7.2	Main approaches for dimensionality reduction	185
4.7.2.1	Projection	185
4.7.2.2	Manifold learning	186
4.7.3	PCA	187
4.7.3.1	Determination of the principal components	188
4.7.3.2	Choosing the right number of dimensions	190
4.7.3.3	PCA for compression	190
4.7.4	Kernel PCA	191
4.7.5	LLE	192
4.8	Unsupervised learning techniques	193
4.8.1	Clustering	193
4.8.1.1	K-means	194
4.8.2	Gaussian mixtures	201
4.8.2.1	Anomaly detection	204
4.8.2.2	Selecting the number of clusters	204
4.9	Artificial neural networks with Keras	206
4.9.1	Summary of artificial neural networks	207
4.9.2	Building an image classifier using Keras	214
4.9.2.1	Using Keras to load the dataset	214
4.9.2.2	Creating the model using the sequential API	215
4.9.2.3	Compiling the model	217
4.9.2.4	Training and evaluating the model	218
4.10	Training deep neural networks (credit: Francesco Di Clemente)	219
4.10.1	Learning rate scheduling	220
4.10.2	The vanishing/exploding gradients problems	220
4.10.3	Faster optimizers	223
4.10.3.1	Momentum optimization	224
4.10.3.2	Nesterov accelerated gradient	224
4.10.3.3	AdaGrad	225
4.10.3.4	RMSProp	226
4.10.3.5	Adam	226
4.10.4	Avoiding overfitting through regularization	227
4.10.4.1	Dropout	227
4.10.4.2	Monte Carlo dropout	228

4 Machine learning techniques

4.10.5	Example: Higgs dataset	229
4.11	Convolutional neural networks	229
4.11.1	Convolutional layers	230
4.11.1.1	Filters	231
4.11.1.2	Stacking multiple feature maps	232
4.11.1.3	Memory requirements	234
4.11.2	Pooling layers	235
4.11.3	Data augmentation	236

4.1 Introduction to machine learning

Machine learning is the science of programming computers so they can *learn from data*. Here is a more general engineering-oriented definition:

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 (e.g., flagged by users) and examples of regular (nonspam, 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.

There are so many different types of machine learning systems that it is useful to organize them in broad categories based on:

- Whether or not they are trained with human supervision. Machine learning systems can be classified according to the amount and type of supervision they get during training (supervised, unsupervised, semisupervised and reinforcement learning).
- Whether or not they can learn incrementally on the fly (online versus batch learning).
- Whether they work by simply comparing new data points to known data points, or instead detect patterns in the training data and build a predictive model, much like scientists do (instance-based versus model-based learning).

These criteria are not exclusive; you can combine them in any way you like.

4.1.1 Supervised/unsupervised learning

4.1.1.1 Supervised learning

In **supervised learning**, the training data you feed to the algorithm includes the desired solutions, called *labels*, i.e. we provide examples that are already classified.

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 (Fig. 4.1).

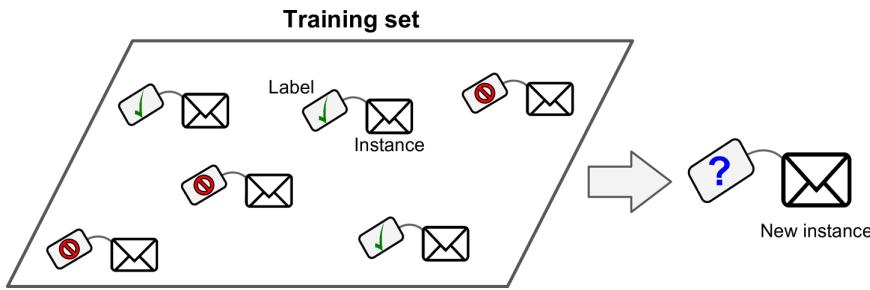


Figure 4.1

Another typical task is to predict a *target* numeric value, given a set of *features*¹ called *predictors*. This sort of task is called *regression* because we start with an initial sample and the algorithm tries to extrapolate (Fig. 4.2).

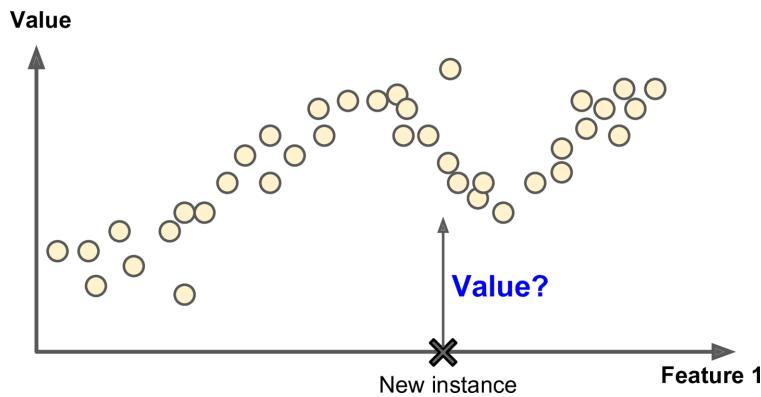


Figure 4.2

4.1.1.2 Unsupervised learning

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

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 (Fig. 4.3). At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40 % of your visitors are males who love comic books, while 20 % are young sci-fi lovers, and so on. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups.

Another important unsupervised task is *anomaly detection*. The system is trained with normal instances, and when it sees a new instance it can tell whether it looks like a normal one or whether it is likely an anomaly. This happens, for example, in 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.

¹In machine learning an attribute is a data type, while a feature has several meanings depending on the context, but generally means an attribute plus its value. Many people use the words attribute and feature interchangeably, though.

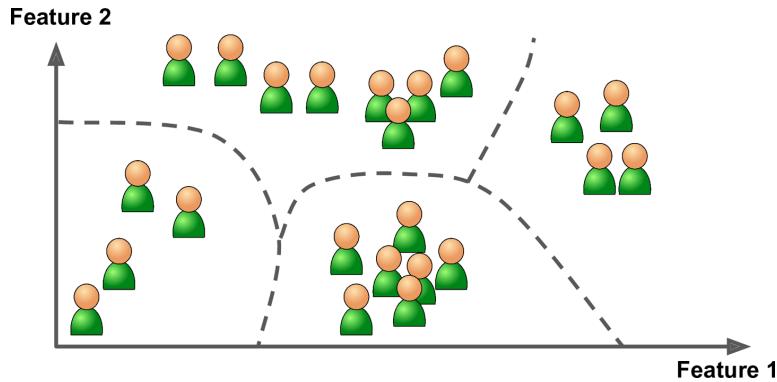


Figure 4.3



Figure 4.4

4.1.1.3 Semisupervised 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 **semisupervised 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 in photos 1, 5 and 11, while another person B shows up in 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 person in a single photo, and it is able to name everyone in every photo, which is useful for searching photos.

The unsupervised character of the learning is the clustering procedure. Consider for instance the situation depicted in Fig. 4.5. Triangles and squares represent the training set the algorithm has learned from, while the little gray circles are the new data that the algorithm has managed to group into clusters (some closer to triangles, others to squares). Now, suppose that the data indicated with a cross is presented to the system; by proximity it would seem closer to the squares than to the triangles of the training set. However, since the algorithm contains some unsupervised learning, it is able to recognize that the cross appears to be more likely to belong to the data cluster around the triangles. Therefore the system will be more inclined to categorize the cross on the same plane as the triangles.

4.1.1.4 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

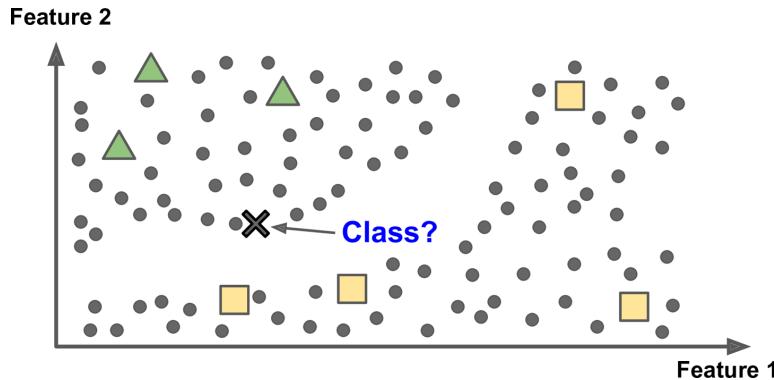


Figure 4.5

in return (or penalties in the form of negative rewards, as in Figure 4.6). 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.

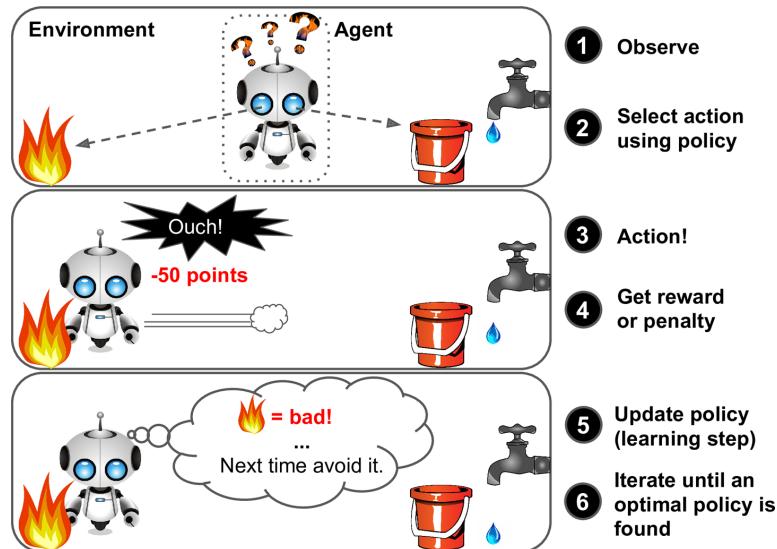


Figure 4.6

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 applied to the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself.

4.1.2 Batch and online learning

Another criterion used to classify machine learning systems is whether or not the system can learn incrementally from a stream of incoming data.

4.1.2.1 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, and then it is launched into

4 Machine learning techniques

production and 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 a 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.

4.1.2.2 Online learning

In **online learning**, you train the system 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 they arrive (Fig. 4.7).

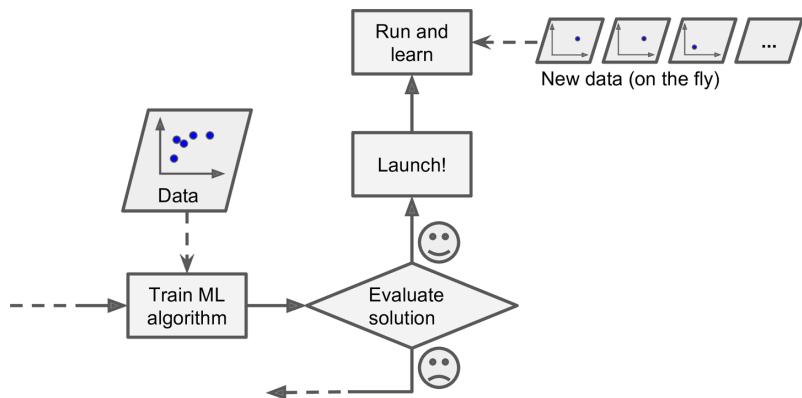


Figure 4.7

Online learning is great for systems that receive data as a continuous flow and need to adapt to change rapidly or autonomously. It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not need them anymore, so you can discard them. 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.

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 flag only the latest kinds of spam it was shown). Conversely, if you set a low learning rate, the system will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points.

4.1.3 Instance-based and model-based learning

One more way to categorize machine learning systems is by 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.

4.1.3.1 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 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, your spam filter could be programmed 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 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 generalizes to new cases using a similarity measure.

4.1.3.2 Model-based learning

Another way to generalize from a set of examples is to build a model of these examples, then use that model to make predictions. In this way, when the system encounters new data, it does not try to detect similarity or proximity to the other examples it has learned; rather it categorizes the new element based on a well-defined model. For example, we can choose that all the data that falls within the dotted line in Fig. 4.8 are always classified as triangles, while the others as squares. This is called **model-based learning**.

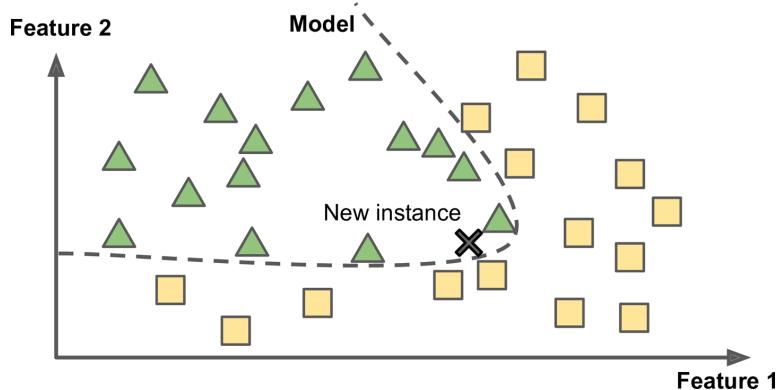


Figure 4.8

4.2 Testing and validating

In a famous paper published in 2001, some Microsoft researchers showed that very different machine learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation once they were given enough data. As the authors put it: “these results suggest that we may want to reconsider the tradeoff between spending time and money on algorithm development versus spending it on corpus development.” In other words, *data matters more than algorithms for complex problems*; typically, when we have huge amounts of data, the size of the dataset is dominant over the specific choice of the algorithm, which becomes essentially insignificant. It should

4 Machine learning techniques

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.

In previous chapters we have often referred to the fact that a good machine, once properly instructed, is also able to generalize when it encounters "unexpected" instances. 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 split your data into two sets: the **training set** and the **test set**. As these names suggest, you train your model using the training set (in order to fix the parameters), and you test it using the test set. The error rate on new cases is called the *generalization error*, and by evaluating your model on the test set, you get an estimation of this error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., 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. This means that the machine will work almost perfectly on the training set, and only on this, but it will almost always fail in generalizing. Overfitting typically occurs when we try to learn the training set with a model having many degrees of freedom; e.g., when we have a sample of N points and we decide to describe the distribution by fitting the data with a polynomial of order $N - 1$ (cf. interpolation in Sec. 3.4.5). Instead, 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. Reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples. Of course, the crucial point is to find the right balance between fitting the data perfectly and keeping the model simple enough to ensure that it will generalize well.

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization* (N.B.: constraining, not reducing the degrees of freedom!). The amount of regularization to apply during learning can be controlled by a **hyperparameter**. A hyperparameter is a parameter defining a learning algorithm (not a model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. Examples of hyperparameters are, for example:

- the topology and size of a neural networks (recall that in the previous chapter we did not say what was the optimal number of layers and hidden neurons for the perceptron);
- the parameter ε in the gradient method;
- the learning rate;
- the mini-batch size.

If you set the regularization hyperparameter to a very large value, you will get an almost flat model; 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.

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?

A common solution to this problem is to have a second holdout set: you simply hold out part of the training set to evaluate several candidate models and select the best one.

The new held-out set is called the **validation set**. More specifically, you train multiple models with various hyperparameters on the **reduced training set** (i.e., the full training set minus the validation set), and you select the model that performs best on the validation set. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

This solution usually works quite well. However, if the validation set is too small, then model evaluations will be imprecise: you may end up selecting a suboptimal model by mistake. Conversely, if the validation set is too large, then the remaining training set will be much smaller than the full training set. Why is this bad? Well, since the final model will be trained on the full training set, it is not ideal to compare candidate models trained on a much smaller training set. One way to solve this problem is to perform repeated **cross-validation**, using many small validation sets. A training dataset can be repeatedly split up into a reduced training dataset and a validation dataset. These repeated partitions can be done in various ways, such as dividing into two equal datasets and using them as training/validation, and then validation/training, or repeatedly selecting a random subset as a validation dataset. Each model is evaluated once per validation set after it is trained on the rest of the data. By averaging out all the evaluations of a model, you get a much more accurate measure of its performance. There is a drawback, however: the training time is multiplied by the number of validation sets.

4.3 Quality of data and preprocessing

In some cases, it's easy to get a large amount of data for training, but this data probably won't be perfectly representative of the data that will be used in production. In this case, the most important rule to remember is that the validation set and the test set must be as *representative* as possible of the data you expect to use in production, i.e. representative of the new cases you want to generalize to.

Another thing which can go wrong in machine learning projects is to have poor-quality data. Obviously, if your training data is full of errors, outliers, and noise (e.g., 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.

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 process, called *feature engineering*, involves the following steps.

- *Feature selection*: 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;
- Creating new features by gathering new data.

In summary, before feeding the machine with data to train on, it is necessary to practice the necessary amount of *preprocessing*, in order to reduce the data mismatch.

4.4 California housing prices

When you are learning about machine learning, it is best to experiment with real-world data, not artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. In this section we'll use the California Housing Prices dataset from the StatLib repository. This dataset is based on data from the 1990 California census. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning, so we will pretend it is recent data. For teaching purposes we've added a categorical attribute and removed a few features.

Our first task is to use California census data to build a model of housing prices in the state. This data includes metrics such as the population, median income and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3000 people). We will call them “districts” for short.

4.4.1 Frame the problem

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? Let's see: it is clearly a typical supervised learning task, since you are given *labeled* training examples (each instance comes with the expected output, i.e., the district's median housing price). It is also a typical regression task, since you are asked to predict a value (the price). More specifically, this is a *multiple regression* problem, since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.). It is also a *univariate regression* problem, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a *multivariate regression* problem. Finally, there is no continuous flow of data coming into the system, there is 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.

4.4.2 Select a performance measure

The first step is to select a performance measure. A typical performance measure for regression problems is the **root-mean-square error** (RMSE). It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors. Equation (4.4.1) shows the mathematical formula to compute the RMSE:

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m [h(\mathbf{x}^{(i)}) - y^{(i)}]^2} \quad (4.4.1)$$

This equation introduces several very common machine learning notations that we will use throughout this text.

- m is the number of instances in the dataset you are measuring the RMSE on (e.g. the total number of districts).
- $\mathbf{x}^{(i)}$ 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 1416 inhabitants with a median income of \$ 38 372, and the median house value is \$ 156 400 (ignoring the other features for now), then

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1416 \\ 38\,372 \end{pmatrix} \quad (4.4.2)$$

and

$$y^{(1)} = 156\,400 \quad (4.4.3)$$

- \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 $(\mathbf{x}^{(i)})^T$.

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(m)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1416 & 38\,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \quad (4.4.4)$$

- 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. For example, if your system predicts that the median housing price in the first district is \$ 156 400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 156\,400$.

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. In that case, you may consider using the **mean absolute error** (MAE, also called the average absolute deviation):

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}| \quad (4.4.5)$$

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:

$$\text{Error}(\mathbf{X}, h) = \left[\frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|^k \right]^{1/k} \quad (4.4.6)$$

The higher the norm index k , the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive, in a negative sense, 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.

4.4.3 Get the data

First you will need to have Python installed. You will also need a number of Python modules: Jupyter, NumPy, pandas, Matplotlib and Scikit-Learn. A valid alternative to installing Python and related packages is Google Colab, which allows you to write and run

4 Machine learning techniques

Python code online in your browser without any configuration (however, this will inevitably increase the compilation time). For the code refer to the attached notebook [→](#).

In the first few lines we have essentially imported all the packages and downloaded the data from the online database. Now let's load the data using pandas. Once again, you should write a small function to load the data (`load_housing_data`). This function returns a pandas DataFrame object containing all the data. Let's take a look at the top five rows using the DataFrame's `head()` method (Fig. 4.9). Each row represents one district. There are 10 attributes: `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` and `ocean_proximity`.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Figure 4.9: In[5]: `housing = load_housing_data() ← housing.head()`.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of non-null values (Fig. 4.10).

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

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   longitude        20640 non-null   float64
 1   latitude         20640 non-null   float64
 2   housing_median_age 20640 non-null   float64
 3   total_rooms      20640 non-null   float64
 4   total_bedrooms   20433 non-null   float64
 5   population       20640 non-null   float64
 6   households       20640 non-null   float64
 7   median_income    20640 non-null   float64
 8   median_house_value 20640 non-null   float64
 9   ocean_proximity  20640 non-null   object  
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 4.10

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 five rows, you probably noticed that the values in the `ocean_proximity` 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 (Fig. 4.11).

Let's look at the other fields. The `describe()` method shows a summary of the numerical attributes (Fig. 4.12). The `count`, `mean`, `min` and `max` rows are self-explanatory. Note that the null values are ignored (so, for example, the 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

```
In [7]: housing["ocean_proximity"].value_counts()

Out[7]: <1H OCEAN      9136
        INLAND       6551
        NEAR OCEAN    2658
        NEAR BAY      2290
        ISLAND         5
Name: ocean_proximity, dtype: int64
```

Figure 4.11

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 25-th percentile (or first quartile), the median, and the 75-th percentile (or third quartile).

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

Figure 4.12: In[8]: `housing.describe()`.

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) that have a given value range (on the horizontal axis). You can either plot 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 (Fig. 4.13).

There are a few things you might notice in these histograms:

- First, the median income attribute does not look like it is expressed in US dollars (USD). 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. The numbers represent roughly tens of thousands of dollars (e.g., 3 actually means about \$30 000).
- 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.
- These attributes have very different scales. We will discuss this later, when we explore feature scaling.
- Finally, 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 learning algorithms to detect patterns. We will try transforming these attributes later on to have more bell-shaped distributions.

4 Machine learning techniques

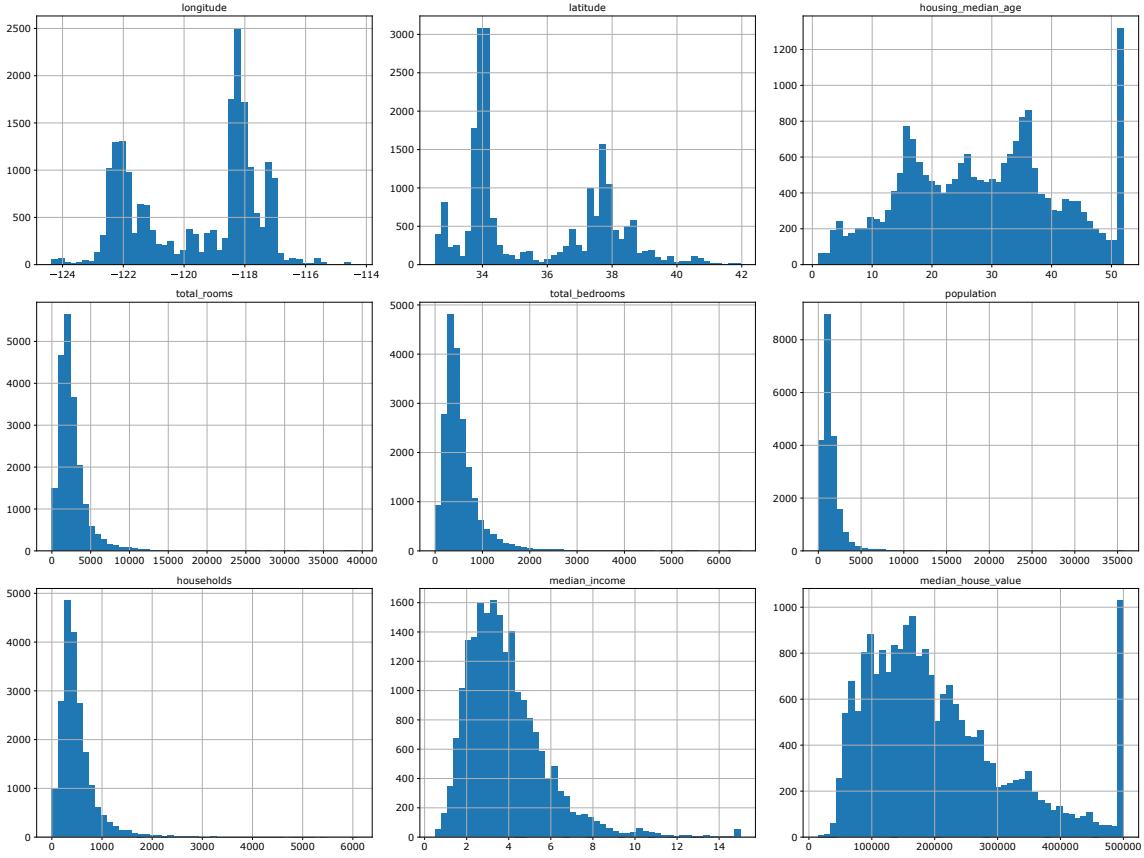


Figure 4.13: In[9]: `housing.hist(bins=50, figsize=(20,15))`.

4.4.4 Create a test set

At this point we have to train the machine, that is, we have to create a test set, put it aside and never look at it. Creating a test set is theoretically simple: pick some instances randomly, typically 20 % of the dataset, or less if your dataset is very large, and set them aside (`train_set, test_set = split_train_test(housing, 0.2)`). 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 (e.g., with `np.random.seed(42)`) before calling `np.random.permutation()` so that it always generates the same shuffled indices.

But both these solutions will break the next time you fetch an updated dataset. To have a stable train/test split even after updating the 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 and put that instance in the test set if the hash is lower than or equal to 20 % of the maximum hash value. 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. On line In[14] is a possible implementation. Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID. 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 that 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 (see the following lines).

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()`, with a couple of additional features. First, there is a `random_state` parameter that allows you to set the random generator seed. Second, you can pass it multiple datasets with an 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).

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 1000 people to ask them a few questions, they don't just pick 1000 people randomly in a phone book. They try to ensure that these 1000 people are representative of the whole population. For example, the US population is 51.3% females and 48.7% males, so a well-conducted survey in the US 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* (e.g. females and males), and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population (i.e. 51.3% and 48.7% respectively). If the people running the survey used purely random sampling from the whole population, there would be about a 12% chance of sampling a skewed test set that was 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 an income category attribute. Let's look at the median income histogram more closely (back in Fig. 4.13): most median income values are clustered around 1.5 to 6 (i.e., \$ 15 000–60 000), 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 a stratum's importance may be biased. This means that you should not have too many strata, and each stratum should be large enough. The code at line `In[23]` uses the `pd.cut()` function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than \$ 15 000), category 2 from 1.5 to 3, and so on. These income categories are represented in Figure 4.14.

Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn's `StratifiedShuffleSplit` class (`In[26]`). Let's see if this worked as expected. You can start by looking at the income category proportions in the test set (Fig. 4.15).

With similar code you can measure the income category proportions in the full dataset, i.e. for data depicted in Figure 4.14. Finally, Figure 4.16 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 skewed. Note also that the error in the random sampling can become quite large (e.g. about 5%), while for the stratified sampling it remains very low.

4 Machine learning techniques

```
In [24]: housing["income_cat"].value_counts()
```

```
Out[24]: 3    7236  
2    6581  
4    3639  
5    2362  
1     822  
Name: income_cat, dtype: int64
```

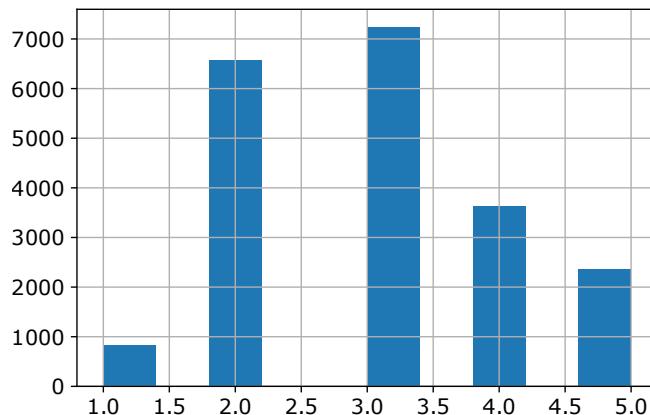


Figure 4.14: In[25]: housing["income_cat"].hist().

```
In [27]: strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

```
Out[27]: 3    0.350533  
2    0.318798  
4    0.176357  
5    0.114583  
1    0.039729  
Name: income_cat, dtype: float64
```

Figure 4.15

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

Figure 4.16: Sampling bias comparison of stratified versus purely random sampling.

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.

4.4.5 Data visualization

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 into a little more 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 that you can play with it without harming the training set: `housing = strat_train_set.copy()`.

Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data (Fig. 4.17).

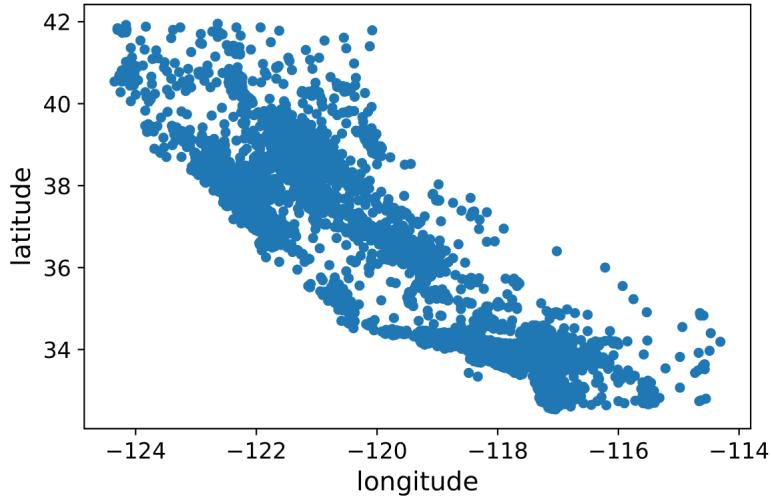


Figure 4.17: A geographical scatterplot of the data.

This looks like California all right, but other than that it is hard to see any particular pattern. Setting the `alpha` option to 0.1 (i.e. the transparency of the points) makes it much easier to visualize the places where there is a high density of data points. 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 (Fig. 4.18). Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.

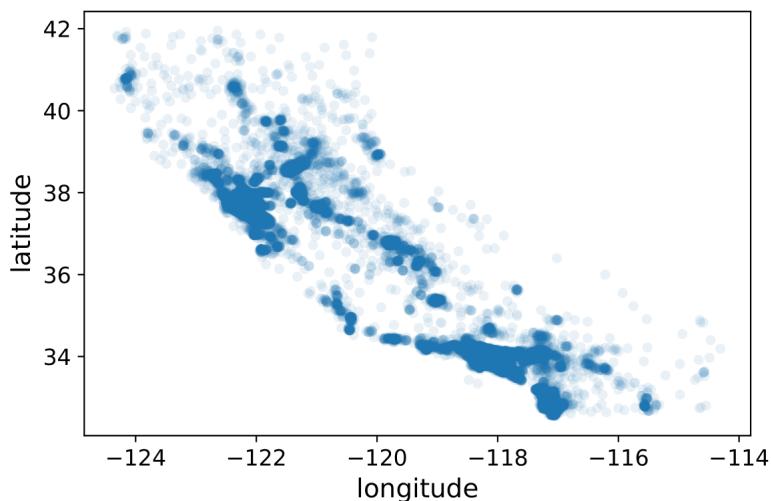


Figure 4.18: A better visualization that highlights high-density areas.

Now let's look at the housing prices (Fig. 4.19). The radius of each circle represents the

district's population (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). In addition, we superimposed a political map of California with its boundaries in the background. This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers.

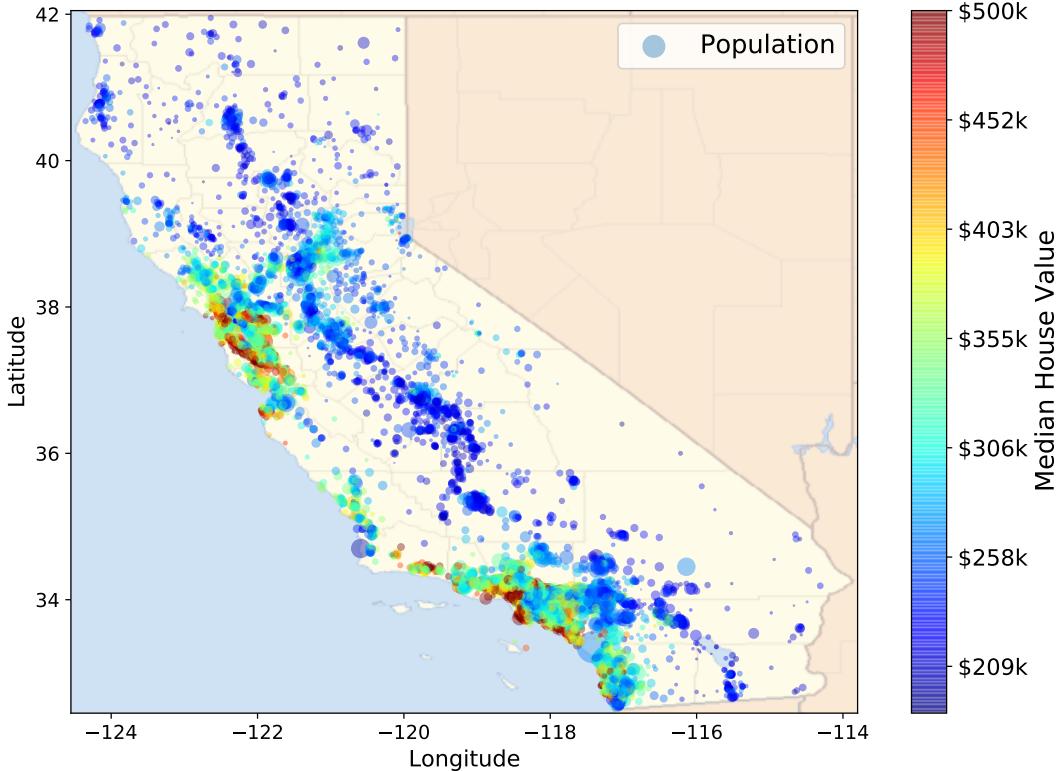


Figure 4.19: California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population.

4.4.6 Looking for correlations

Since the dataset is not too large, you can easily compute the standard **Pearson correlation coefficient** (also referred to as Pearson's r) between every pair of attributes using the `corr()` method. The correlation coefficient is a measure of linear correlation between two sets of data. As such, it is defined as the covariance of two variables, divided by the product of their standard deviations:

$$\rho_{X,Y} = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y} \quad (4.4.7)$$

with

$$\text{Cov}(X, Y) = E[(X - \mu_X)(Y - \mu_Y)] \quad (4.4.8)$$

Thus it is essentially a normalized measurement of the covariance, such that the result always has a value between -1 and 1 : $-1 \leq \rho \leq 1$.

Now let's look at how much each attribute correlates with the median house value (Fig. 4.20). 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 (i.e., prices have a slight tendency to go down when you go north). Finally, coefficients close to 0 mean that there is no linear correlation.

```
In [38]: corr_matrix = housing.corr()

In [39]: corr_matrix["median_house_value"].sort_values(ascending=False)

Out[39]: median_house_value    1.000000
          median_income       0.687160
          total_rooms          0.135097
          housing_median_age   0.114110
          households           0.064506
          total_bedrooms        0.047689
          population            -0.026920
          longitude             -0.047432
          latitude              -0.142724
Name: median_house_value, dtype: float64
```

Figure 4.20

Figure 4.21 shows various example plots along with the correlation coefficient between their horizontal and vertical axes. The correlation coefficient only measures *linear* correlations (“if x goes up, then y generally goes up/down”), whereas it may completely miss out on nonlinear relationships (e.g., “if x is close to 0, then y generally goes up”). Therefore all the plots of the bottom row have a correlation coefficient equal to 0, despite the fact that their axes are clearly not independent and ordered patterns are clearly recognizable: these are examples of nonlinear relationships. Another feature to remark is the fact that the correlation reflects the strength and direction of a linear relationship (top row), but it has nothing to do with the slope of that relationship (middle). For example, your height in meters has a correlation coefficient of 1 with your height in millimeters. Notice also that the figure in the center has a slope of 0 but in that case the correlation coefficient is undefined because the variance of y is zero.

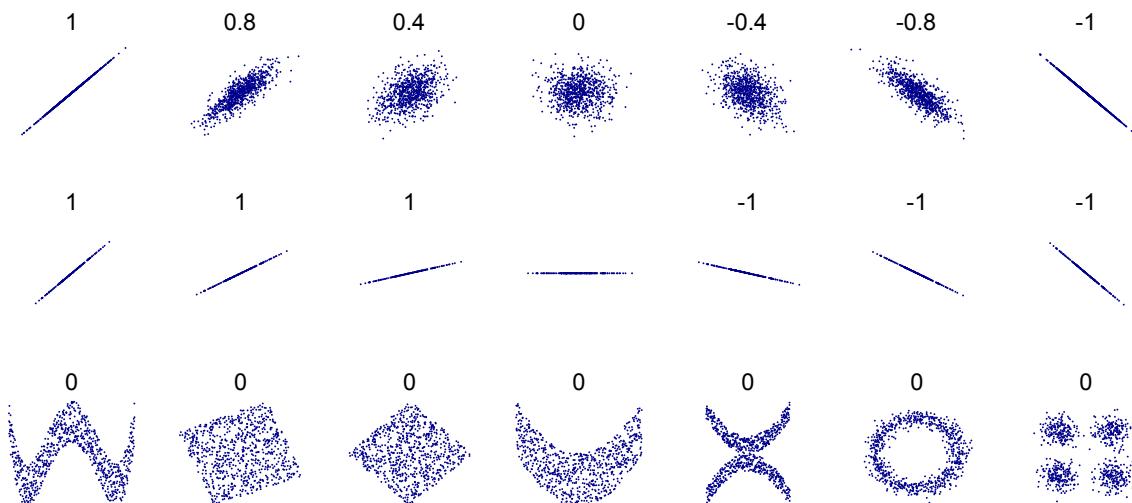


Figure 4.21: Several sets of (x, y) points, with the correlation coefficient of x and y for each set.

Another way to check for correlation between attributes is to use the pandas function

4 Machine learning techniques

`scatter_matrix()`, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you would get $11^2 = 121$ 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 (Fig. 4.22). The main diagonal (top left to bottom right) would be full of straight lines if pandas plotted each variable against itself, which would not be very useful. So instead pandas displays a histogram of each attribute (other options are available; see the pandas documentation for more details).

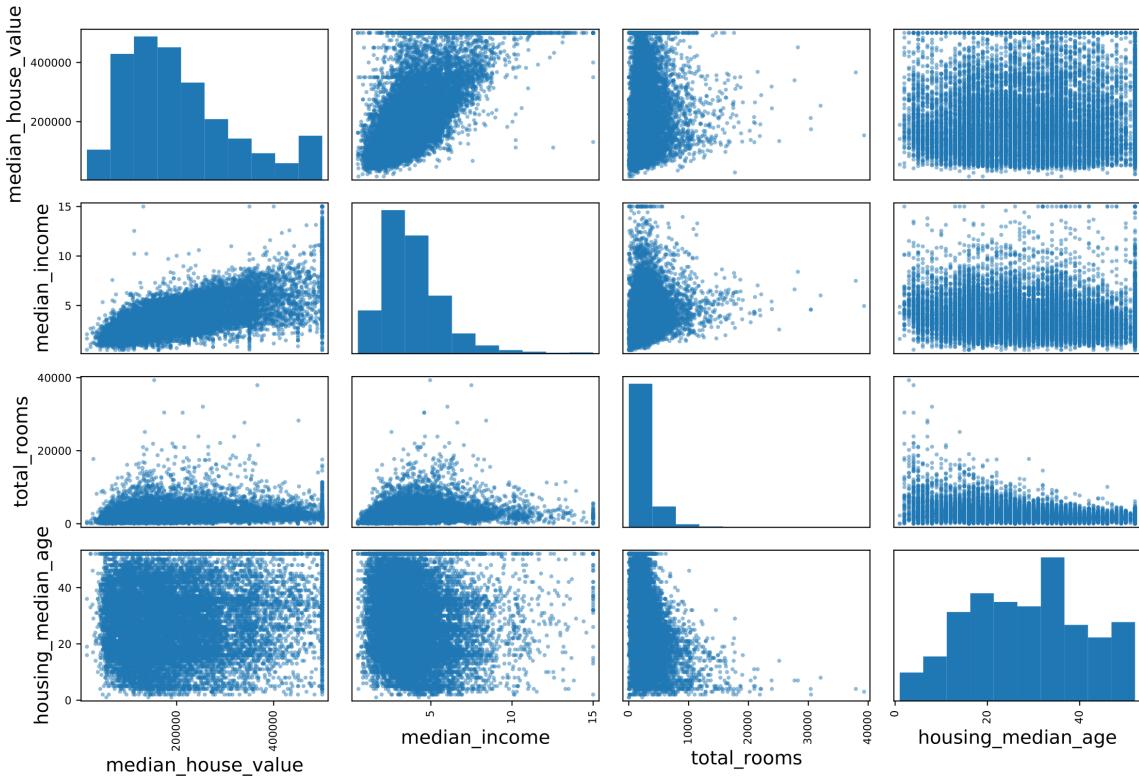


Figure 4.22

The most promising attribute to predict the median house value is the median income (recall Fig. 4.20), so let's zoom in on their correlation scatterplot (Fig. 4.23). 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.

4.4.7 Experimenting with attribute combinations

One last thing you may want to do before 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

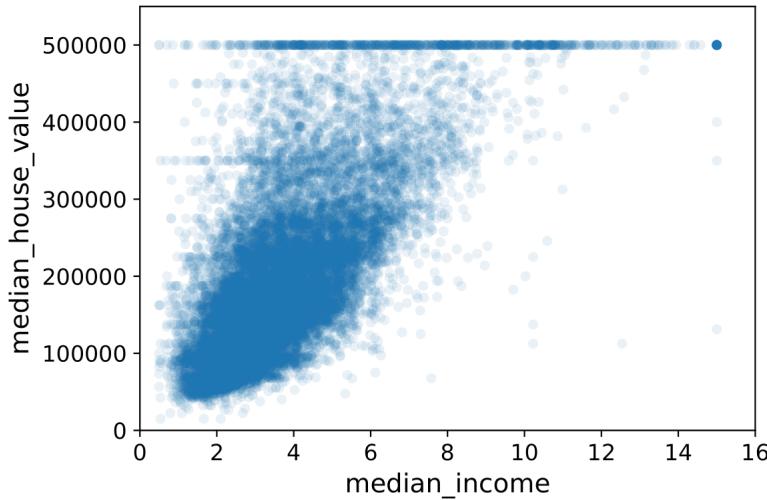


Figure 4.23

at. Let's create these new attributes (Fig. 4.24).

```
In [42]: housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]

In [43]: corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)

Out[43]: median_house_value      1.000000
median_income          0.687160
rooms_per_household    0.146285
total_rooms             0.135097
housing_median_age     0.114110
households              0.064506
total_bedrooms          0.047689
population_per_household -0.021985
population              -0.026920
longitude                -0.047432
latitude                 -0.142724
bedrooms_per_room       -0.259984
Name: median_house_value, dtype: float64
```

Figure 4.24

And now let's look at the correlation matrix again. 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 (rich people typically prefer many rooms, but not for sleeping). 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.

4.4.8 Data cleaning

Most machine learning algorithms cannot work with missing features, so let's create a few functions to take care of them. We saw earlier (and proposed again in Fig. 4.25) that the `total_bedrooms` attribute has some missing values, i.e. `NaN` values (Not a Number); so let's fix this. You have three options:

1. Get rid of the corresponding districts; not good because we risk to remove some very

4 Machine learning techniques

significant districts for our analysis.

2. Get rid of the whole attribute; not good because we have seen that there is a (negative) correlation between the bedroom/room ratio and the value of the houses.
3. Set the values to some value (zero, the mean, the median, etc.).

If you choose option 3, you should compute the median value on the training set and use it to fill the missing values in the training set. 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: `SimpleImputer`.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
4629	-118.30	34.07		18.0	3759.0	NaN	3296.0	1462.0	2.2708 <1H OCEAN
6068	-117.86	34.01		16.0	4632.0	NaN	3038.0	727.0	5.1762 <1H OCEAN
17923	-121.97	37.35		30.0	1955.0	NaN	999.0	386.0	4.6328 <1H OCEAN
13656	-117.30	34.05		6.0	2155.0	NaN	1039.0	391.0	1.6675 INLAND
19252	-122.79	38.48		7.0	6837.0	NaN	3468.0	1405.0	3.1662 <1H OCEAN

Figure 4.25

4.4.9 Handling text and categorical attributes

So far we have only dealt with numerical attributes, but now let's look at text attributes. In this dataset, there is just one: the `ocean_proximity` attribute. Let's look at its value for the first 10 instances in Fig. 4.26. It's not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute. Most machine learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's `OrdinalEncoder` class. For example: `<1H OCEAN` → 0, `INLAND` → 1, `ISLAND` → 2, `NEAR BAY` → 3, `NEAR OCEAN` → 4 (Fig. 4.27).

ocean_proximity	
17606	<1H OCEAN
18632	<1H OCEAN
14650	NEAR OCEAN
3230	INLAND
3555	<1H OCEAN
19480	INLAND
8879	<1H OCEAN
13685	INLAND
4937	<1H OCEAN
4861	<1H OCEAN

Figure 4.26

```
Out[64]: array([[0.],
 [0.],
 [4.],
 [1.],
 [0.],
 [1.],
 [0.],
 [1.],
 [0.],
 [0.]]))
```

Figure 4.27

You can get the list of categories using the `categories_` instance variable (Fig. 4.28). It is a list containing a 1D array of categories for each categorical attribute (in this case, a list containing a single array since there is just one categorical attribute).

```
In [65]: ordinal_encoder.categories_
Out[65]: [array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
   dtype=object)]
```

Figure 4.28

One issue with this representation is that machine learning algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad”, “average”, “good” and “excellent”), but it is obviously not the case for the `ocean_proximity` column. For example, categories 0 (`<1H OCEAN`) and 4 (`NEAR OCEAN`) are clearly more similar than categories 0 and 1 (`INLAND`). To fix this issue, a common solution is to create one binary attribute per category: if a district is “`<1H OCEAN`” we associate the attribute 1 to the `<1H OCEAN` category and 0 to all the others, if a district is “`INLAND`” we associate the attribute 1 to the `INLAND` category and 0 to all the others, and so on. This is called *one-hot encoding*, because for each instance only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors.

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 0s except for a single 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 nonzero elements. You can use it mostly like a normal 2D array, but if you really want to convert it to a (dense) NumPy array, just call the `toarray()` method (Fig. 4.29).

```
In [67]: housing_cat_1hot.toarray()
Out[67]: array([[1., 0., 0., 0., 0.],
   [1., 0., 0., 0., 0.],
   [0., 0., 0., 0., 1.],
   ...,
   [0., 1., 0., 0., 0.],
   [1., 0., 0., 0., 0.],
   [0., 0., 0., 1., 0.]])
```

Figure 4.29: Each row indicates a different district, while the columns represent the categorical entries.

4.4.10 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 ranges 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*.

4 Machine learning techniques

- Min-max scaling (many people call this *normalization*) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1. Practically, if we define the minimum and the maximum value for a specific feature k as

$$m_k := \min \{x_k^{(i)}\} \quad (4.4.9)$$

$$M_k := \max \{x_k^{(i)}\} \quad (4.4.10)$$

then we do min-max scaling by subtracting the minimum value and dividing by the maximum minus the minimum:

$$\tilde{x}_k^{(i)} = \frac{x_k^{(i)} - m_k}{M_k - m_k} \quad (4.4.11)$$

Scikit-Learn provides a transformer called `MinMaxScaler` for this. It also has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0–1.

- Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance:

$$\tilde{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sigma_k} \quad (4.4.12)$$

Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., 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). Min-max scaling would then 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).

4.4.11 Transformation pipelines

As you can see, there are many data transformation steps that need to be executed in the right order before processing the data. Fortunately, Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations. The `Pipeline` constructor takes a list of name/estimator pairs defining a sequence of steps. This is done because we want our procedure to be somehow automatized; therefore we join all these components into a big pipeline that will preprocess both the numerical and the categorical features (lines `In[73]`–`In[82]`).

4.4.12 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. Notice that now, after the preprocessing,

the training set is called `housing_prepared` and the corresponding labels, i.e. the prices, are stored in `housing_labels`. The size of the former is also changed because we have converted the categorical attributes into numerical ones (Fig. 4.30). Anyway, you are now ready to select and train a machine learning model.

```
In [77]: housing_prepared.shape
Out[77]: (16512, 16)
```

Figure 4.30

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 (the algorithm is already implemented inside Scikit-Learn). Remember that in a linear regression we have a sample of points (x_i, y_i) and we typically try to reproduce their distribution according to a straight line

$$\hat{y} = ax + b \quad (4.4.13)$$

Clearly, here we have different values for more than one feature, but we can still implement a linear model by combining them linearly in the following way:

$$\hat{y} = a_1x_1 + a_2x_2 + \dots + a_fx_f + b \quad (4.4.14)$$

Let's try it out on a few instances from the training set (Fig. 4.31). It works, although the predictions are not exactly accurate (e.g., the first prediction is off by close to 40 %!).

```
In [84]: # Let's try the full preprocessing pipeline on a few training instances
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)

print("Predictions:", lin_reg.predict(some_data_prepared))

Predictions: [210644.60459286 317768.80697211 210956.43331178 59218.98886849
189747.55849879]

In [85]: print("Labels:", list(some_labels))

Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Figure 4.31

Let's measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function (Fig. 4.32). This is better than nothing, but clearly not a great score: In the majority of districts the `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 section, 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, which rules out the last option. You could try to add more features (e.g., 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 nonlinear relationships in the data. The code should look familiar by now (Fig. 4.33). Now that the model is trained, let's evaluate it on the training set. As we can see, the error

4 Machine learning techniques

```
In [87]: 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
```

```
Out[87]: 68628.19819848923
```

Figure 4.32

is exactly 0. Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data, i.e. we passed from fitting to interpolation. How can we 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 of it for model validation.

```
In [89]: from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)
```

```
Out[89]: DecisionTreeRegressor(random_state=42)
```

```
In [90]: housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

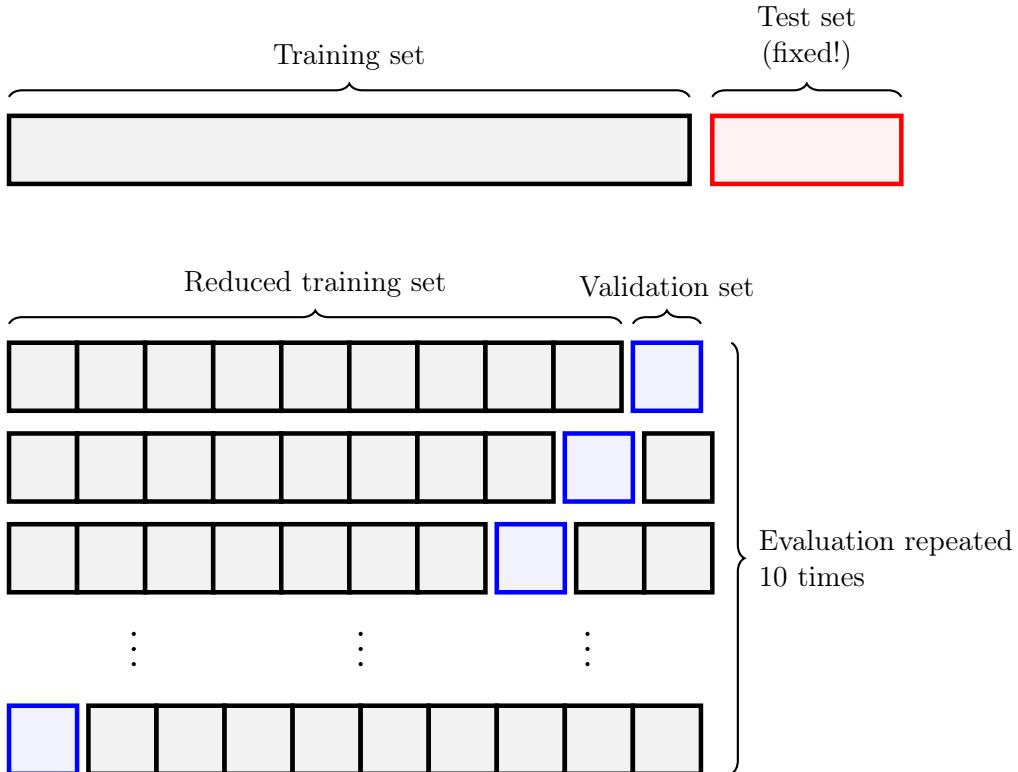
```
Out[90]: 0.0
```

Figure 4.33

4.4.13 Better evaluation using cross-validation

One way to evaluate the decision tree model would be 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. This can be done with Scikit-Learn's *K-fold cross-validation* feature. The following code randomly splits the full training set into 10 distinct subsets called *folds*; then it trains the model on 9 of these folds (which constitute the reduced training set) and it picks the remaining one for evaluation (validation set). This cross-validation procedure is repeated 10 times, every time selecting different folds for the reduced training set and the validation set, always in ratio 9:1. Thus, the result is an array containing the 10 evaluation scores (Fig. 4.34). Notice that Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better); therefore the scoring function is actually the opposite of the MSE (i.e., a negative value), which is why the preceding code computes `-scores` before calculating the square root (Fig. 4.35).

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 also a measure of how precise this estimate is (i.e., its standard deviation). The decision tree has a score of approximately 71 407, generally ± 2439 . You would not have this information if you just used one validation

Figure 4.34: Schematic representation of the K -fold cross-validation.

```
In [91]: from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                        scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

```
In [92]: def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)

Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
71115.88230639 75585.14172901 70262.86139133 70273.6325285
75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

Figure 4.35

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 (Fig. 4.36). As we can see from the output, we were 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 the next sections, 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

4 Machine learning techniques

```
In [93]: lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                                     scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)

Scores: [66782.73843989 66960.118071 70347.95244419 74739.57052552
68031.13388938 71193.84183426 64969.63056405 68281.61137997
71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798342
```

Figure 4.36

models is called *ensemble learning*, and it is often a great way to push machine learning algorithms even further. We will skip most of the code since it is essentially the same as for the other models:

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 (i.e., regularize it), or get a lot more training data. Before you dive much deeper into random forests, however, you should try out many other models from various categories of machine learning algorithms (e.g., several support vector machines with different kernels, and possibly a neural network; line In[98] of the code), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising models.

```
In [94]: from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

```
Out[94]: RandomForestRegressor(random_state=42)
```

```
In [95]: housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

```
Out[95]: 18603.515021376355
```

```
In [96]: from sklearn.model_selection import cross_val_score

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
Scores: [49519.80364233 47461.9115823 50029.02762854 52325.28068953
49308.39426421 53446.37892622 48634.8036574 47585.73832311
53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

Figure 4.37

4.4.14 Fine-tune your model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them, i.e. to optimize the hyperparameters of the models. Let's look at a few ways you can

do that. One option 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 use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code (Fig. 4.38) searches for the best combination of hyperparameter values for the `RandomForestRegressor`.

```
In [99]: from sklearn.model_selection import GridSearchCV

param_grid = [
    # try 12 (3x4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2x3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)

Out[99]: GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
                      param_grid=[{'max_features': [2, 4, 6, 8],
                                   'n_estimators': [3, 10, 30]},
                                  {'bootstrap': [False], 'max_features': [2, 3, 4],
                                   'n_estimators': [3, 10]}],
                      return_train_score=True, scoring='neg_mean_squared_error')
```

Figure 4.38

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 later on), 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). 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 five-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 in Fig. 4.39.

```
In [100]: grid_search.best_params_

Out[100]: {'max_features': 8, 'n_estimators': 30}
```

Figure 4.39

You can also get the best estimator directly (line `In[101]`). And of course the evaluation scores are also available (Fig. 4.40).

In this example, we obtain the best solution by setting the `max_features` hyperparameter to 8 and the `n_estimators` hyperparameter to 30. Notice that they are exactly the maximum values in the lists `n_estimators` and `max_features`; so this grid search suggests that the algorithm works better with the highest values. This means that if we include other larger values, they are likely to provide even better estimates; but to be sure it would be necessary

4 Machine learning techniques

```
In [102]: cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)

63669.11631261028 {'max_features': 2, 'n_estimators': 3}
55627.099719926795 {'max_features': 2, 'n_estimators': 10}
53384.57275149205 {'max_features': 2, 'n_estimators': 30}
60965.950449450494 {'max_features': 4, 'n_estimators': 3}
52741.04704299915 {'max_features': 4, 'n_estimators': 10}
50377.40461678399 {'max_features': 4, 'n_estimators': 30}
58663.93866579625 {'max_features': 6, 'n_estimators': 3}
52006.19873526564 {'max_features': 6, 'n_estimators': 10}
50146.51167415009 {'max_features': 6, 'n_estimators': 30}
57869.25276169646 {'max_features': 8, 'n_estimators': 3}
51711.127883959234 {'max_features': 8, 'n_estimators': 10}
49682.273345071546 {'max_features': 8, 'n_estimators': 30}
62895.06951262424 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.176157539405 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.40652318466 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52724.9822587892 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.5691951261 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.495668875716 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Figure 4.40

to check. Anyway, the RMSE score for this combination is 49 682, which is slightly better than the score you got earlier using the default hyperparameter values (which was 50 182). Congratulations, you have successfully fine-tuned your best model!

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, 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. An example is given at lines In[104]–In[105].

4.4.15 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. Let's display these importance scores next to their corresponding attribute names (Fig. 4.41). These numbers specify how much each of the various attributes affects the price of houses. For example, as we have already anticipated, the `median_income` is strongly correlated to the prices. With these information, you may want to try dropping some of the less useful features; for instance, apparently only one `ocean_proximity` category is really useful, so you could try dropping the others.

4.4.16 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 `full_pipeline` to transform the data and evaluate the final model on the test set. The machine gives us a root-mean-square error of 47 730; so it's comparable to the typical error made experts of houses and is just slightly better than before.

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1 % better than the model currently in production?

```
In [107]: extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)

Out[107]: [(0.36615898061813423, 'median_income'),
(0.16478099356159054, 'INLAND'),
(0.10879295677551575, 'pop_per_hhold'),
(0.07334423551601243, 'longitude'),
(0.06290907048262032, 'latitude'),
(0.056419179181954014, 'rooms_per_hhold'),
(0.053351077347675815, 'bedrooms_per_room'),
(0.04114379847872964, 'housing_median_age'),
(0.014874280890402769, 'population'),
(0.014672685420543239, 'total_rooms'),
(0.014257599323407808, 'households'),
(0.014106483453584104, 'total_bedrooms'),
(0.010311488326303788, '<1H OCEAN'),
(0.0028564746373201584, 'NEAR OCEAN'),
(0.0019604155994780706, 'NEAR BAY'),
(6.0280386727366e-05, 'ISLAND')]
```

Figure 4.41

You might want to have an idea of how precise this estimate is. For this, you can compute a 95 % *confidence interval* for the generalization error using `scipy.stats.t.interval`.

In this California housing example, the final performance of the system is not better than the experts' price estimates, which were often off by about 20 %, but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks.

4.5 Training models

So far we have treated machine learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous sections, 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 they actually work. Indeed, in many situations you don't really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what's under the hood will also help you debug issues and perform error analysis more efficiently. For the following list of models refer to the notebook .

4.5.1 Linear regression

First, we will start by looking at the **linear regression** model, one of the simplest models there is. Generally speaking, 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), as shown below:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (4.5.1)$$

4 Machine learning techniques

This model is just a linear function of the input features x_i , while $\theta_0, \theta_1, \dots, \theta_n$ are the model's parameters. It is called "linear" because the input features enter linearly in the equation, but geometrically speaking equation (4.5.1) represents a hyperplane in $n + 1$ dimensions²; e.g., for $n = 1$ it represents a straight line in 2 dimensions. The quantity \hat{y} represents the predicted value, and it can be written much more concisely using a vectorized form:

$$\hat{y} \equiv h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x} = \boldsymbol{\theta}^T \mathbf{x} \quad (4.5.2)$$

In this equation $\boldsymbol{\theta}$ and \mathbf{x} are the model's *parameter vector* and instance's *feature vector*, respectively:

$$\boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \quad (4.5.3)$$

while $\boldsymbol{\theta} \cdot \mathbf{x}$ is the standard scalar product of these vectors and h_{θ} is the hypothesis function, using the model parameters. Notice that we have set $x_0 = 1$ in order to reproduce the bias term.

Now, recall that training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. At the beginning of Sec. 4.4 we saw that the most common performance measure of a regression model is the root-mean-square error (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2} \quad (4.5.4)$$

Therefore, to train a linear regression model, we need to find the value of $\boldsymbol{\theta}$ that minimizes the RMSE. In practice, it is simpler to minimize the mean squared error (MSE):

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})^2 \quad (4.5.5)$$

than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root). N.B.: written in these forms, equations (4.5.4) and (4.5.5) only apply to the linear regression model, since we have already made explicit the functional form of the hypothesis function $h_{\theta}(\mathbf{x})$ as $\boldsymbol{\theta}^T \mathbf{x}$.

To find the value of $\boldsymbol{\theta}$ that minimizes the cost function, it would be sufficient to calculate the derivative of the mean squared error and set the result equal to 0. However, we realize that by introducing the matrix \mathbf{X} of the features values

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(m)})^T \end{pmatrix} = \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \cdots & x_n^{(m)} \end{pmatrix} \quad (4.5.6)$$

this matrix is generally not a square matrix. In fact it is a $m \times (n + 1)$ matrix; therefore, if $m \neq n + 1$, this matrix is not square. Accordingly, when computing the derivative, the

²A hyperplane is a subspace whose dimension is one less than that of its ambient space.

procedure can become a bit tricky. For this reason, we just quote the closed-form solution, i.e. a mathematical equation that gives the result directly, called the **normal equation** and given by

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (4.5.7)$$

So, in this equation $\hat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimizes the cost function (the error) and \mathbf{y} is the vector of target values:

$$\mathbf{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \quad (4.5.8)$$

Note that the dimensions of $\hat{\boldsymbol{\theta}}$ are well defined because the product gives

$$\begin{aligned} \hat{\boldsymbol{\theta}} &= \left([m \times (n+1)]^T [m \times (n+1)] \right)^{-1} [m \times (n+1)]^T [m \times 1] \\ &= \left([(n+1) \times m] [m \times (n+1)] \right)^{-1} [(n+1) \times m] [m \times 1] \\ &= [(n+1) \times (n+1)]^{-1} [(n+1) \times 1] \\ &= [(n+1) \times (n+1)] [(n+1) \times 1] \\ &= [(n+1) \times 1] \end{aligned} \quad (4.5.9)$$

Let's generate some linear-looking data to test this equation; this type of data are sometimes called "mock data" because they are simulated samples that mimic the behavior of real objects in controlled ways, most often exploited as part of a testing initiative. To do this, we have first generated 100 random numbers x_i uniformly distributed between 0 and 2, and then we have created the corresponding y_i according to the function $y_i = 4 + 3x_i + \text{Gaussian noise}$ (Fig. 4.42).

```
In [2]: import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

Figure 4.42

Now let's compute $\hat{\boldsymbol{\theta}}$ using the normal equation (Fig. 4.43). Looking at what the equation found, we would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$ instead of $\theta_0 = 4.215$ and $\theta_1 = 2.770$. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

Anyway, now we can make predictions using $\hat{y} = \hat{\boldsymbol{\theta}} \cdot \mathbf{x} = \hat{\boldsymbol{\theta}}^T \mathbf{x}$, i.e. equation (4.5.2). In Figure 4.44 are shown the initial data (blue points) and the best linear model optimized with the parameters dictated by $\hat{\boldsymbol{\theta}}$ (red line).

In the previous lines we computed the best fit "manually", directly exploiting the expression of the normal equation. But in general, performing linear regression is much more simple using Scikit-Learn because it already has an implemented function dedicated to it: `scipy.linalg.lstsq()` (Fig. 4.45). However, the normal equation expects to invert a

4 Machine learning techniques

```
In [4]: X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance  
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)  
  
In [5]: theta_best  
  
Out[5]: array([4.21509616],  
              [2.77011339]))
```

Figure 4.43

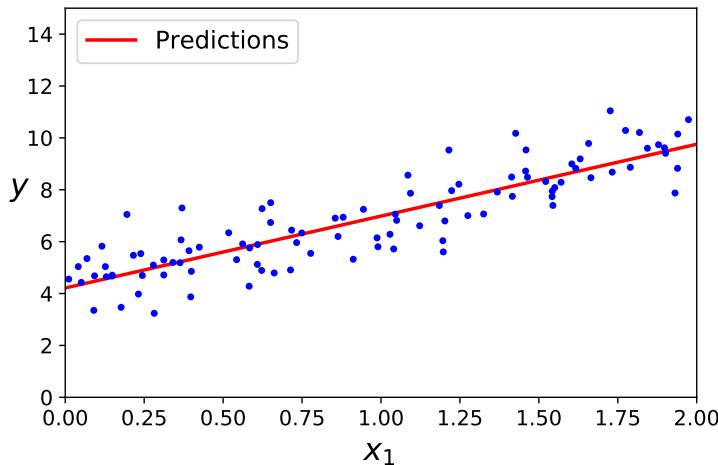


Figure 4.44

matrix, and this can be particularly delicate or unstable for a variety of reasons. Therefore, this function actually computes $\theta = \mathbf{X}^+ \mathbf{y}$, where \mathbf{X}^+ is the Moore–Penrose inverse or *pseudoinverse* of \mathbf{X} (\mathbf{X}^+ is a specific notation; it does not mean Hermitian conjugate). The pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix \mathbf{X} into the matrix multiplication of three matrices: $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^+ = \mathbf{V}\Sigma^+\mathbf{U}^T$. To compute the matrix Σ^+ , the algorithm takes Σ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their reciprocals, and finally it transposes the resulting matrix. This approach is more efficient than computing the normal equation, plus it handles edge cases nicely: indeed, the normal equation may not work if the matrix $\mathbf{X}^T\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined. Naturally, for invertible matrices the pseudoinverse equals the usual inverse.

```
In [9]: from sklearn.linear_model import LinearRegression  
  
lin_reg = LinearRegression()  
lin_reg.fit(X, y)  
lin_reg.intercept_, lin_reg.coef_  
  
Out[9]: (array([4.21509616]), array([[2.77011339]]))
```

Figure 4.45

The normal equation computes the inverse of $\mathbf{X}^T\mathbf{X}$, which is an $(n + 1) \times (n + 1)$ 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 $2^{2.4} \approx 5.3$ to $2^3 = 8$. Instead, the SVD approach used by Scikit-Learn's `LinearRegression` class is about $O(n^2)$. If you double the number of features, you multiply the computation time by roughly 4.

Keep in mind that both the normal equation and the SVD approach get very slow when the number of features grows large (e.g., 100 000). On the positive side, both are linear with regard to the number of instances in the training set, $O(m)$, so they handle large training sets efficiently, provided they can fit in memory.

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

4.5.2 Gradient descent

Gradient descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems (we have already seen it when we spoke about neural networks). The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function (Fig. 4.46). This technique measures the local gradient of the error function with regard to the parameter vector θ , and it goes in the direction of descending gradient. Once the gradient is zero, we have reached a minimum! Concretely, we start by filling θ with random values (this is called random initialization). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum:

$$\theta^{\text{next}} = \theta - \eta \nabla_{\theta} \text{MSE} \quad (4.5.10)$$

(computing ∇_{θ} MSE leads to $\frac{2}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y})$).

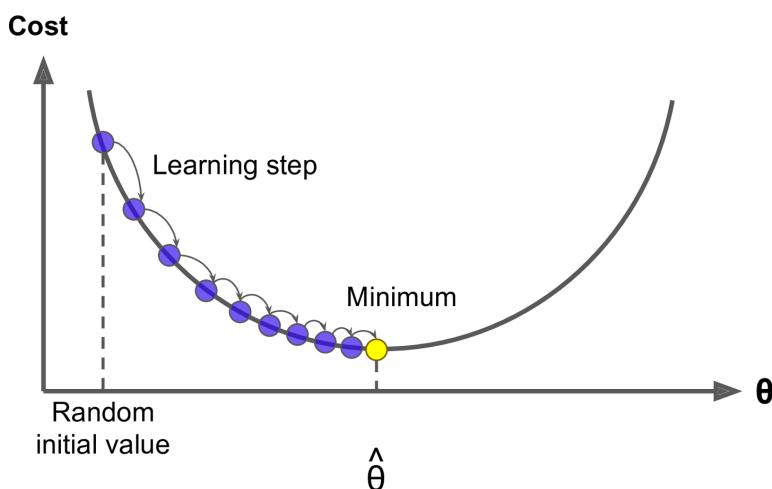


Figure 4.46

An important parameter in gradient descent is the size of the steps, determined by the *learning rate* hyperparameter η . 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. 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. Figure 4.47 shows the first 10 steps of gradient descent using three different learning rates (the dashed line represents the starting point). 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.

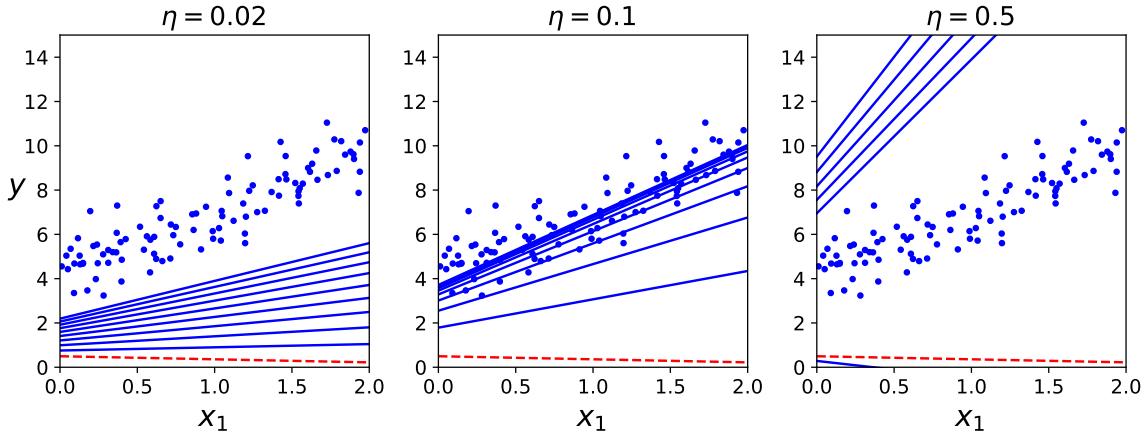


Figure 4.47: Gradient descent with various learning rates.

Finally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaux, and all sorts of irregular terrains, making convergence to the minimum difficult. Figure 4.48 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.

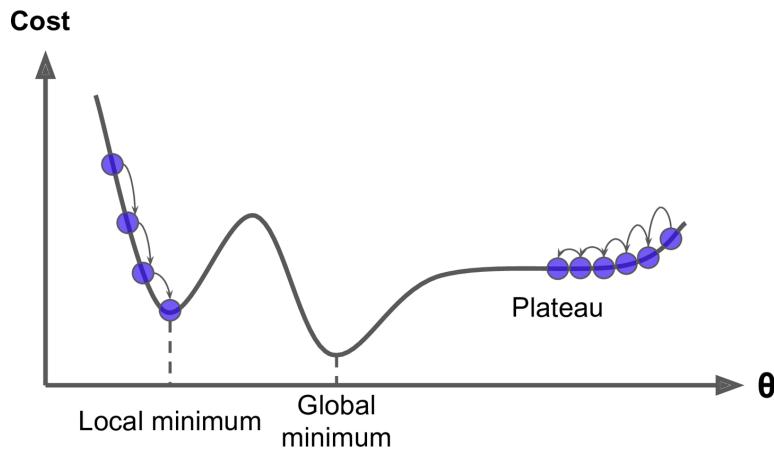


Figure 4.48

Fortunately, the MSE cost function for a linear regression model 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 abruptly. 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 cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure 4.49 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). As you can see, on the left the gradient descent algorithm goes straight toward 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. Therefore, when using gradient descent, you should *ensure that all features have a similar scale*, or else it will take much longer to converge.

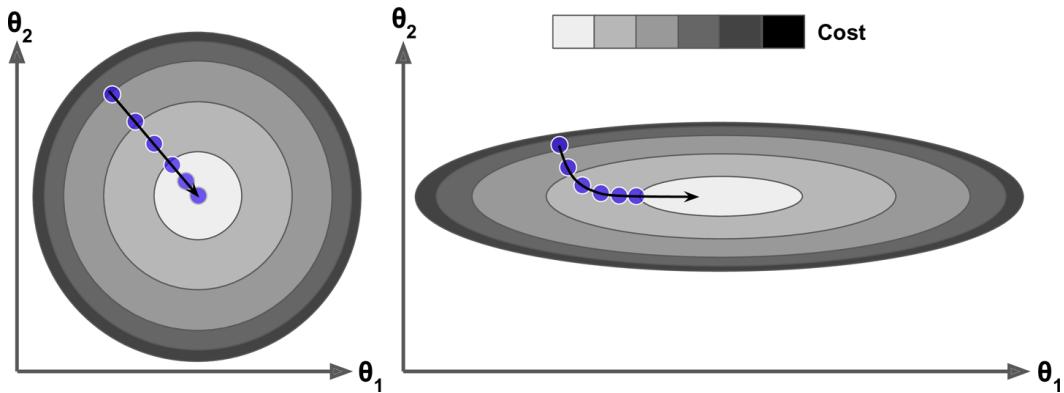


Figure 4.49: Gradient Descent with (left) and without (right) feature scaling.

4.5.2.1 Stochastic gradient descent

Notice that formula (4.5.10) 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 (actually, full gradient descent would probably be a better name). As a result it is terribly slow on very large training sets.

A possible solution is **stochastic gradient descent**, which picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because 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.

On the other hand, due to its stochastic (i.e., 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 Fig. 4.50). So once the algorithm stops, the final parameter values are good, but not optimal.

Moreover, when the cost function is very irregular (as in Figure 4.48), 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, a process akin to *simulated annealing*. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. The function that

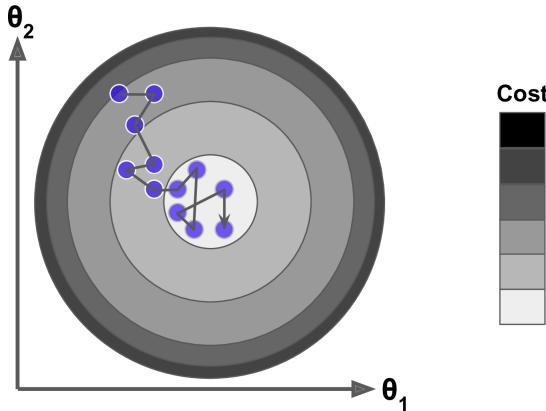


Figure 4.50

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 halfway 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. The code in Fig. 4.51 implements stochastic gradient descent using a simple learning schedule:

$$\text{Learning schedule} = \frac{t_0}{t + t_1} \quad (4.5.11)$$

where t_0 and t_1 are hyperparameters of the model, while t indicates the time. By convention we iterate by rounds of m iterations; each round is called an *epoch*. While the batch gradient descent code iterated 1000 times through the whole training set, this code goes through the training set only 50 times and reaches a pretty good solution.

```
In [19]: n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

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

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20: # not shown in the book
            y_predict = X_new_b.dot(theta) # not shown
            style = "b-" if i > 0 else "r--" # not shown
            plt.plot(X_new, y_predict, style) # not shown
        random_index = np.random.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
        theta_path_sgd.append(theta) # not shown
```

Figure 4.51

4.5.2.2 Mini-batch gradient descent

The last gradient descent algorithm we will look at is called **mini-batch gradient descent**. It is simple to understand once you know batch and stochastic gradient descent: at each

step, instead of computing the gradients based on the full training 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 algorithm's progress in parameter space is less erratic than with stochastic GD, especially with fairly large mini-batches. As a result, mini-batch GD will end up walking around a bit closer to the minimum than stochastic GD. But it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike linear regression). Figure 4.52 shows the paths taken by the three 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.

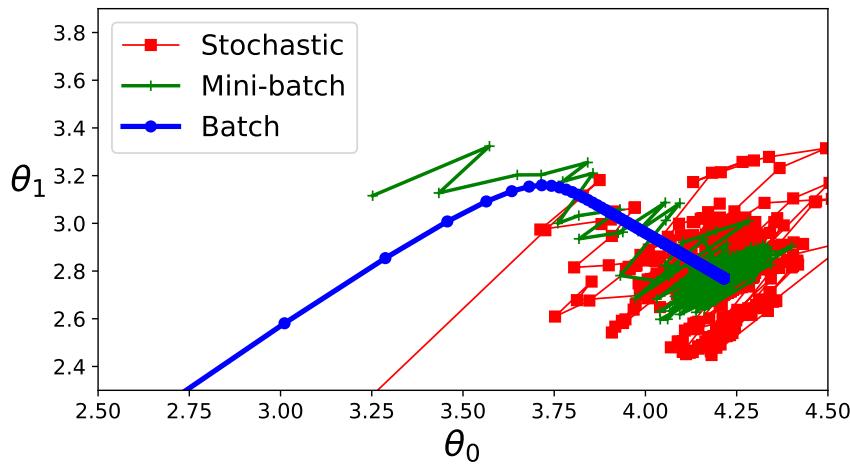


Figure 4.52

4.5.3 Polynomial regression

What if your data is more complex than a straight line? Surprisingly, you can still use a linear model to fit nonlinear 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**. For instance, suppose we have just one feature and many instances; if we expect a quadratic relation between x and y :

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 \quad (4.5.12)$$

then we take the initial sample

$$\left\{ x_1^{(1)}, \dots, x_1^{(m)} \right\} \quad (4.5.13)$$

and we replace it with the extended

$$\begin{aligned} & \left\{ x_1^{(1)}, (x_1^{(1)})^2, \dots, x_1^{(m)}, (x_1^{(m)})^2 \right\} \\ & \qquad \downarrow \\ & \left\{ \bar{x}_1^{(1)}, \bar{x}_2^{(1)}, \dots, \bar{x}_1^{(m)}, \bar{x}_2^{(m)} \right\} \end{aligned} \quad (4.5.14)$$

such that we can apply a linear regression model of the type

$$\hat{y} = \theta_0 + \theta_1 \bar{x}_1 + \theta_2 \bar{x}_2 \quad (4.5.15)$$

4 Machine learning techniques

with two features instead of just one.

Let's look at an example (Figs. 4.53 and 4.54). First, let's generate some nonlinear data, based on a simple quadratic equation (plus some noise). Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `PolynomialFeatures` class to transform our training data, adding the square (second-degree polynomial) of each feature in the training set as a new feature (in this case there is just one feature³). `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. The result are not bad: the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $\hat{y} = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

```
In [28]: m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

```
In [30]: from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

```
Out[30]: array([-0.75275929])
```

```
In [31]: X_poly[0]
```

```
Out[31]: array([-0.75275929,  0.56664654])
```

```
In [32]: lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
Out[32]: (array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

Figure 4.53

Notice that this procedure can be done only for a certain class of functions, like polynomials. For example, if we modeled with a function like e^{ax}/x there would be no possibility of being able to reduce it to a linear model.

4.5.4 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 4.55 applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances. This high-degree polynomial regression 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

³Note 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. E.g. combinations like x_1x_2 .

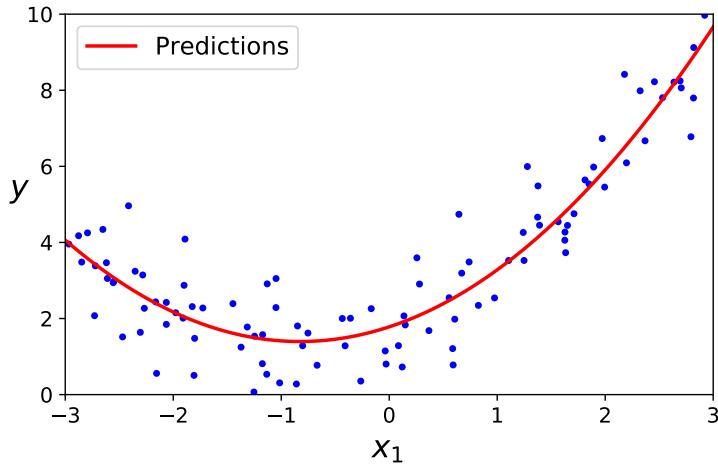


Figure 4.54: Polynomial regression model predictions (In[33]).

quadratic model, which makes sense because 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?

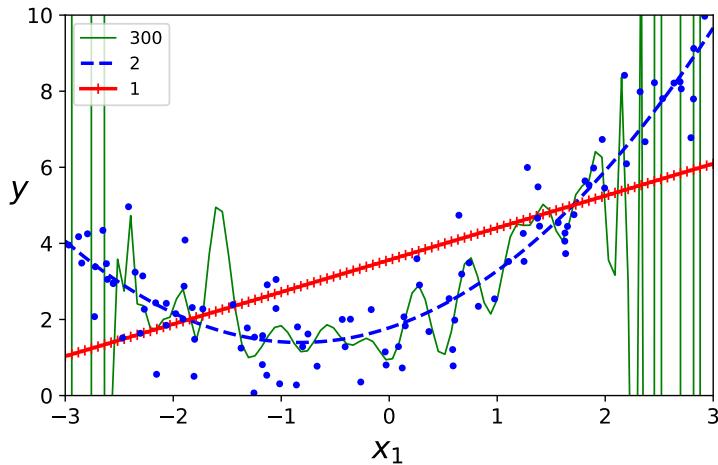


Figure 4.55

In Section 4.4.13 we 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 to tell 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 (or the training iteration). To generate the plots, train the model several times on different sized subsets of the training set.

Let's look at the learning curves of the plain linear regression model (a straight line; see Fig. 4.56). This model that's underfitting 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 (one, and only one, line passes through two points). But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly with a

straight line, 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 which 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 a model that's underfitting. Both curves have reached a plateau; they are close and fairly high.

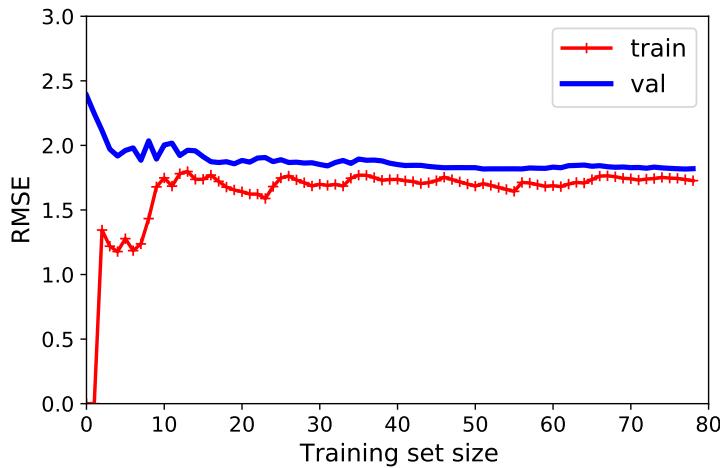


Figure 4.56: Learning curves for the linear regression model.

Now let's look at the learning curves of a 10-th-degree polynomial model on the same data (Fig. 4.57). These learning curves look a bit like the previous ones, but there are two very important differences:

- The error on the training data is much lower than with the linear regression model.
- 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. If you used a much larger training set, however, the two curves would continue to get closer.

4.5.4.1 On the nature of errors

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 are linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.

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 overfit the training data.

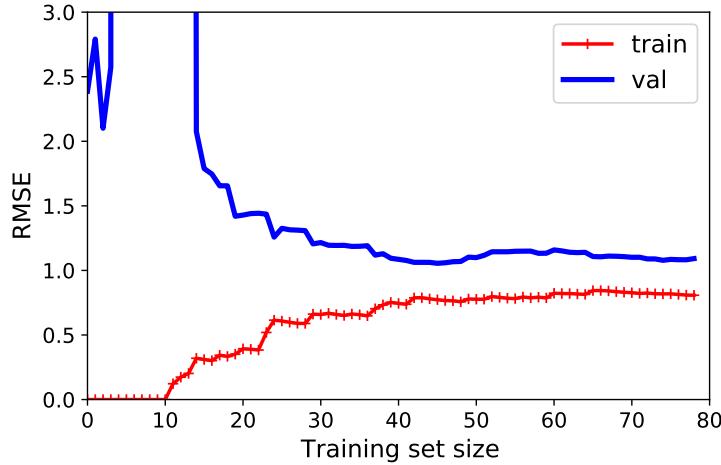


Figure 4.57: Learning curves for the 10-th-degree polynomial model.

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 (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

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 trade-off.

4.5.5 Regularized linear models

As we saw in Section 4.2, a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. 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.

4.5.5.1 Ridge regression

Ridge regression (also called *Tikhonov regularization*) is a regularized version of linear regression based on a regularization term equal to $\alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$ added to the cost function:

$$J(\boldsymbol{\theta}) = \text{MSE} + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2 \quad (4.5.16)$$

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). Anyway, ridge regression forces the learning algorithm to not only fit the data but also 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 use the unregularized performance measure to evaluate the model’s performance.

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.

Figure 4.58 shows several ridge models trained on some linear data using different α values. 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 it is 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 (i.e., less oscillatory, more reasonable) predictions, thus reducing the model's variance but increasing its bias.

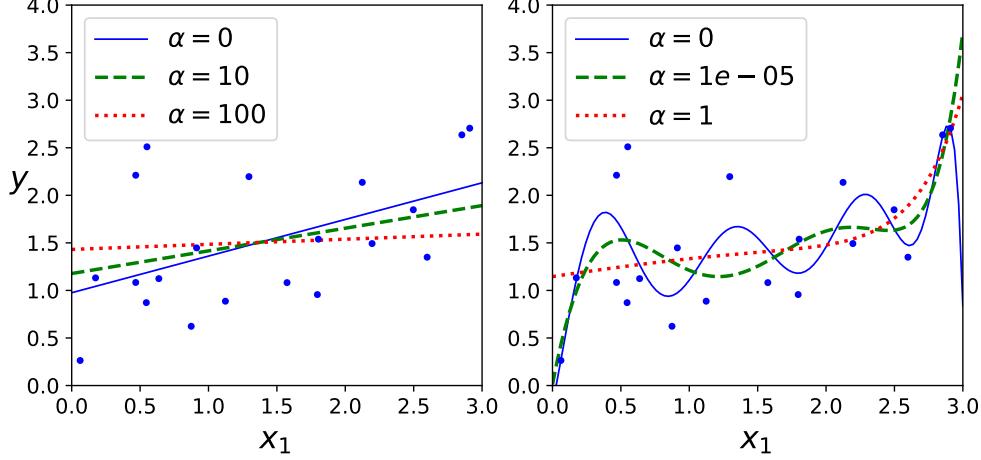


Figure 4.58

4.5.5.2 Lasso regression

Least absolute shrinkage and selection operator regression, usually 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 norm⁴:

$$J(\boldsymbol{\theta}) = \text{MSE} + \alpha \sum_{i=1}^n |\theta_i| \quad (4.5.18)$$

Figure 4.59 shows the same thing as Figure 4.58 but replaces ridge models with lasso models and uses smaller α values.

An important characteristic of lasso regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the right-hand plot in Figure 4.59 (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* (i.e., with few nonzero feature weights).

4.5.5.3 Early stopping

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**

⁴Remember that the ℓ_p norm of a vector $\mathbf{x} = (x_1, \dots, x_n)$ is defined as

$$\|\mathbf{x}\|_{\ell_p} := \left(\sum_{i=1}^n |x_i|^p \right)^{1/p} \quad (4.5.17)$$

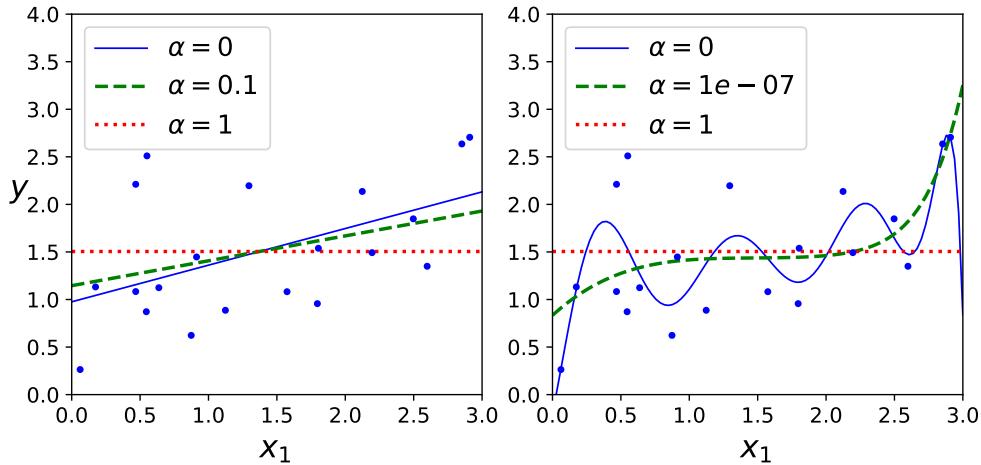


Figure 4.59

stopping. Figure 4.60 shows a complex model (in this case, a high-degree polynomial regression model) being trained with batch gradient descent. As the epochs go by the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set. After a while though, the validation error stops decreasing and 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.

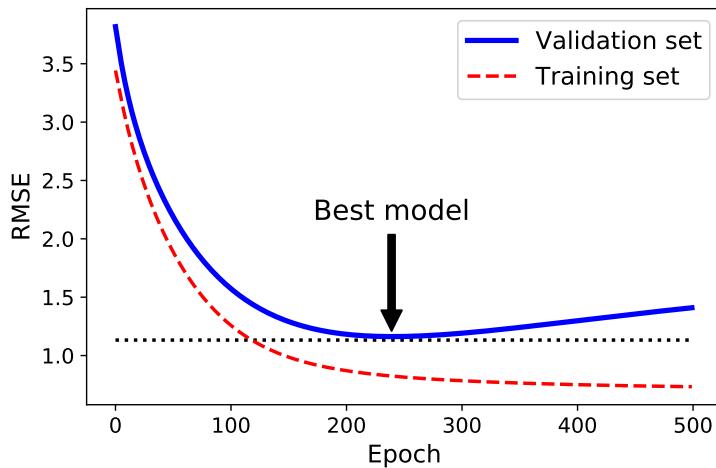


Figure 4.60

4.5.6 Logistic regression

As we discussed in Section 4.1, some regression algorithms can be used for classification (and vice versa). **Logistic regression** is commonly used to estimate the probability that an instance belongs to a particular class (e.g., 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*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”). This makes it a binary classifier.

4.5.6.1 Estimating probabilities

So how does logistic regression 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

$$\hat{p} = \sigma(\mathbf{x}^T \boldsymbol{\theta}) \quad (4.5.19)$$

where

$$\sigma(t) = \frac{1}{1 + e^{-t}} \quad (4.5.20)$$

Here σ constitutes the so-called logistic, which is a sigmoidal function (i.e., S-shaped) that outputs a number between 0 and 1 (see Fig. 4.61). The logistic function gives a measure of the *probability* for a specific outcome, and not the exact prediction of the model. Therefore, in order to compute a determined output we need a rule to “convert” the logistic function into a step function. So, once the logistic regression model has estimated the probability $\hat{p} = \sigma(\mathbf{x}^T \boldsymbol{\theta})$ that an instance \mathbf{x} belongs to the positive class, it can make its prediction \hat{y} easily as

$$\hat{y} = \begin{cases} 0, & \text{if } \hat{p} < 0.5 \\ 1, & \text{if } \hat{p} \geq 0.5 \end{cases} \quad (4.5.21)$$

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 $\mathbf{x}^T \boldsymbol{\theta}$ is positive and 0 if it is negative: $\hat{y} = \Theta(\mathbf{x}^T \boldsymbol{\theta})$.

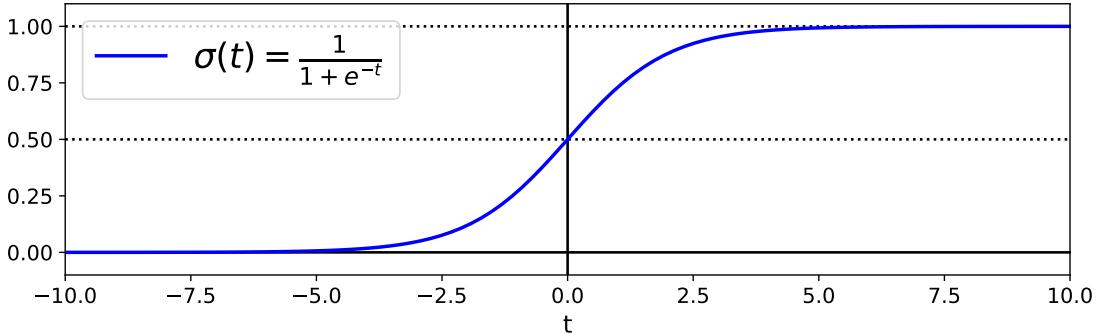


Figure 4.61

Note the close similarity between logistic regression and stochastic neural networks (Sec. 3.2.6). Also in that case we were used to adopt a deterministic law for the description of the neuron response (sign), but then we introduced a stochastic law based essentially on a Fermi–Dirac distribution. There, the connection between deterministic and stochastic was based on the dependence on a pseudo-temperature, in such a way that for $T \rightarrow 0$ we would lead back to the deterministic case.

4.5.6.2 Training and cost function

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 $\boldsymbol{\theta}$ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$), i.e. in order to make the smallest possible errors in the classification task. This idea is captured by the *cost function* for a single training instance \mathbf{x} :

$$c(\boldsymbol{\theta}) = \begin{cases} -\ln \hat{p}, & \text{if } y = 1 \\ -\ln (1 - \hat{p}), & \text{if } y = 0 \end{cases} \quad (4.5.22)$$

which measures how far we are from what we want to obtain⁵. This cost function makes sense because $-\ln \hat{p}$ grows very large when \hat{p} approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance ($y = 1$), and it will also be very large if the model estimates a probability close to 1 for a negative instance ($y = 0$). On the other hand, $-\ln \hat{p}$ is close to 0 when \hat{p} 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 the average cost over all training instances:

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \ln \hat{p}^{(i)} + (1 - y^{(i)}) \ln (1 - \hat{p}^{(i)}) \right] \quad (4.5.23)$$

In fact, if $y^{(i)} = 1$ the second term vanishes and the cost for that specific positive instance is $-\ln \hat{p}^{(i)}$, while if $y^{(i)} = 0$ it is the first term to cancel itself out and the cost for that specific negative instance is $-\ln (1 - \hat{p}^{(i)})$. Here we have written the global cost in a single expression, which is usually called the *log loss*. The bad news is that there is no known closed-form equation to compute the value of $\boldsymbol{\theta}$ that minimizes this cost function; in other words, there is no equivalent of the normal equation. However, the good news is that this cost function is convex, so gradient descent⁶ (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).

4.5.6.3 Decision boundaries

Let's use the Iris dataset to illustrate logistic regression, trying to understand if a certain flower belongs to a specific class. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor* and *Iris virginica* (see Figure 4.62).

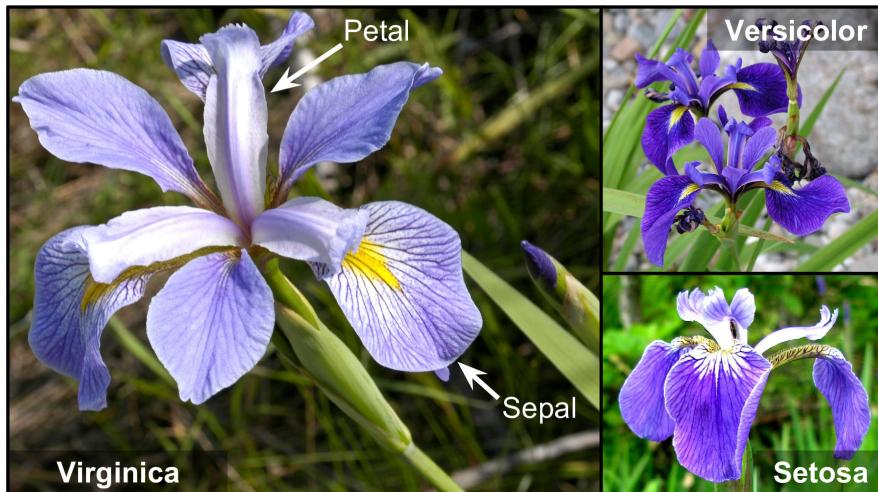


Figure 4.62

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, and then let's train a logistic regression model with

⁵It is quite common to use logarithm functions in logistic tasks.

⁶ $\nabla J(\boldsymbol{\theta}) = \frac{1}{m} \mathbf{X}^T (\boldsymbol{\sigma} - \mathbf{y})$, with $\boldsymbol{\sigma}^{(i)} = \sigma(\boldsymbol{\theta}^T \mathbf{x}^{(i)})$.

4 Machine learning techniques

the Scikit-Learn's `LogisticRegression()`. Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm (Figure 4.63). 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 for 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, 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*, and otherwise it will predict that it is not (even if it is not very confident).

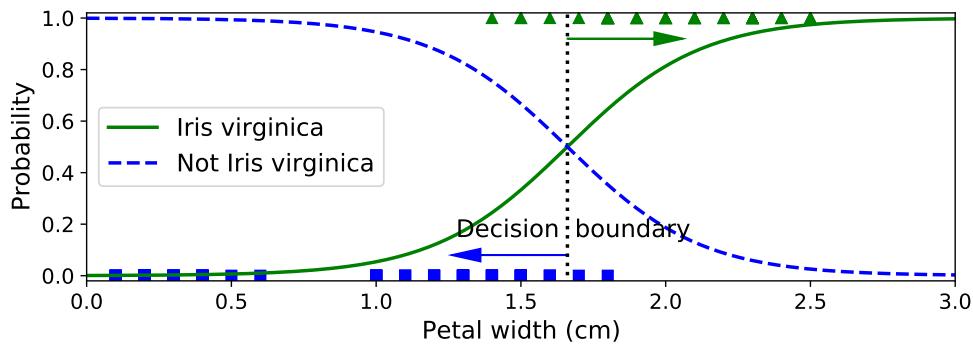


Figure 4.63

Figure 4.64 shows the same dataset, but this time displaying two features: petal width and length. Once trained, the logistic regression classifier can, based on these two features, estimate the probability that a new flower is an *Iris virginica*. 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 boundary. 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 an over 90% chance of being *Iris virginica*, according to the model.

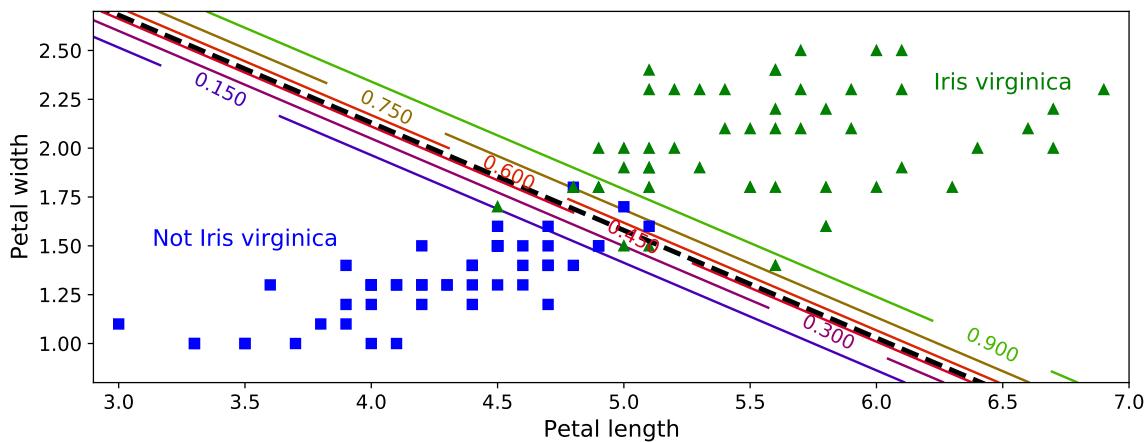


Figure 4.64

4.5.6.4 Softmax regression

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called multinomial classifiers) can distinguish between more than two classes. The logistic regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers. This is called *softmax regression*, or *multinomial logistic regression*. The idea is simple: when given an instance \mathbf{x} , the softmax regression model first computes a score $s_k(\mathbf{x})$ for each class k , then estimates the probability of belonging to each class by applying the softmax function (also called the normalized exponential) to the scores.

The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for linear regression prediction

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)} \quad (4.5.24)$$

Note that each class has its own dedicated parameter vector $\boldsymbol{\theta}^{(k)}$. All these vectors are typically stored as rows in a parameter matrix $\boldsymbol{\Theta}$. 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*

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))} \quad (4.5.25)$$

where K is the total number of classes, $\mathbf{s}(\mathbf{x})$ is a vector containing the scores $s_j(\mathbf{x})$ of each class for the instance \mathbf{x} , and $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance \mathbf{x} belongs to class k , given the scores of each class for that instance. As we can see, the function computes the exponential of every score, then normalizes them dividing by the sum of all the exponentials.

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):

$$\hat{y} = \operatorname{argmax}_k \hat{p}_k = \operatorname{argmax}_k (s_k(\mathbf{x})) = \operatorname{argmax}_k (\mathbf{x}^T \boldsymbol{\theta}^{(k)}) \quad (4.5.26)$$

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. Bear in mind that the softmax regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only 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 cost function

$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \ln(\hat{p}_k^{(i)}) \quad (4.5.27)$$

called the *cross entropy*⁷, 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 matches the target classes. In this equation

⁷Cross entropy originated from information theory. 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).

$y_k^{(i)}$ is the target probability that the i -th instance belongs to class k ; in general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not. Notice also that when there are just two classes ($K = 2$), this cost function is equivalent to the logistic regression's cost function (4.5.23).

Let's use softmax regression to classify the iris flowers into all three classes. Scikit-Learn's `LogisticRegression` uses one-versus-the-rest 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. Figure 4.65 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 versicolor* class, represented by the curved lines (e.g., the line labeled 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 %.

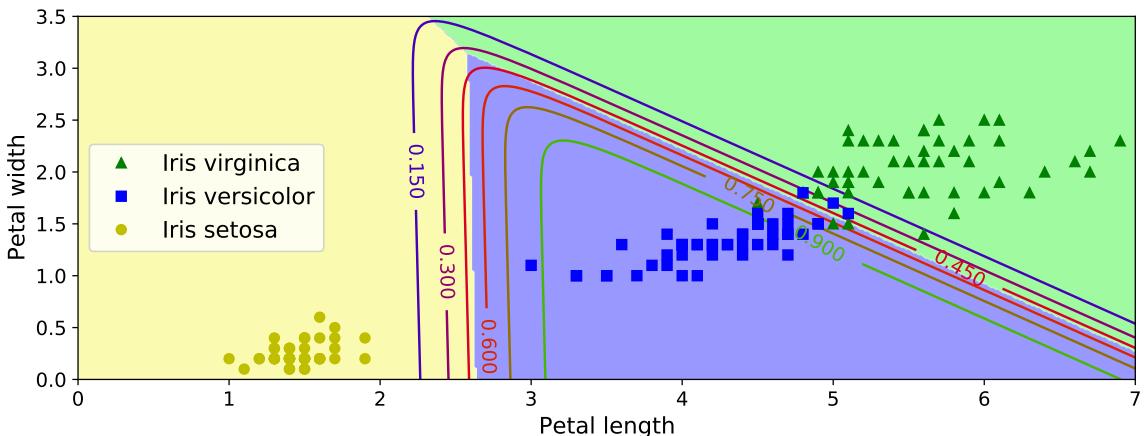


Figure 4.65

4.6 Support vector machines

Support vector machine (SVM) is a powerful and versatile machine learning model, capable of performing linear or nonlinear 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 it in their toolbox. SVMs are particularly well suited for classification of complex small- or medium-sized datasets. This section will explain the core concepts of SVMs, how to use them, and how they work. For the code refer to [→](#).

4.6.1 Linear SVM classification

The fundamental idea behind SVMs is best explained with some pictures. Figure 4.66 shows part of the iris dataset that was introduced at the end of the previous section. The two classes can clearly be separated easily with a straight line (they are *linearly separable*⁸). 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

⁸We have seen something very similar when treating the AND and OR operations in the context of simple perceptrons (Sec. 3.3.1). Also in that case we were able to separate the two classes of outputs (Boolean functions) with just one line, whereas for the XOR this was not possible.

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*. 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 4.66).

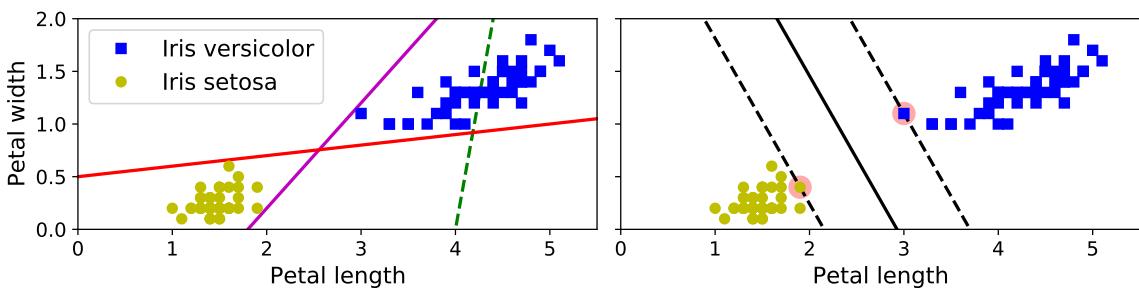


Figure 4.66

SVMs are sensitive to the feature scales, as you can see in Figure 4.67: in 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 (e.g., using Scikit-Learn’s `StandardScaler`), the decision boundary in the right plot looks much better.

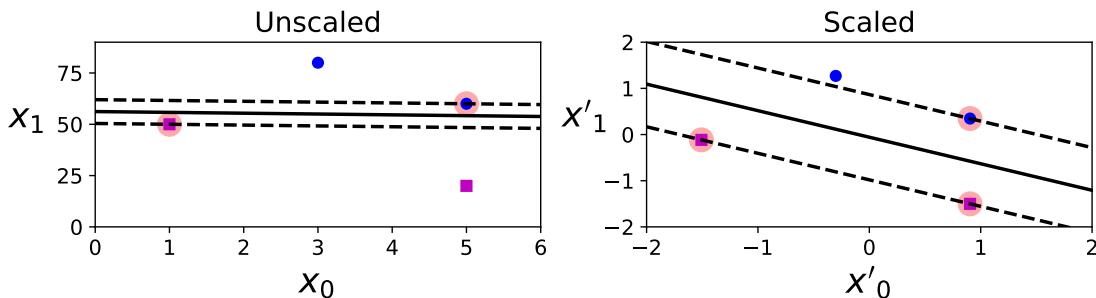


Figure 4.67

4.6.1.1 Soft margin classification

If we strictly impose that all instances must 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 are linearly separable. Second, it is sensitive to outliers. Figure 4.68 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin; on the right, the decision boundary ends up very different from the one we saw in Figure 4.66 without the outlier, and it will probably not generalize as well.

To avoid these issues, 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 (i.e., instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

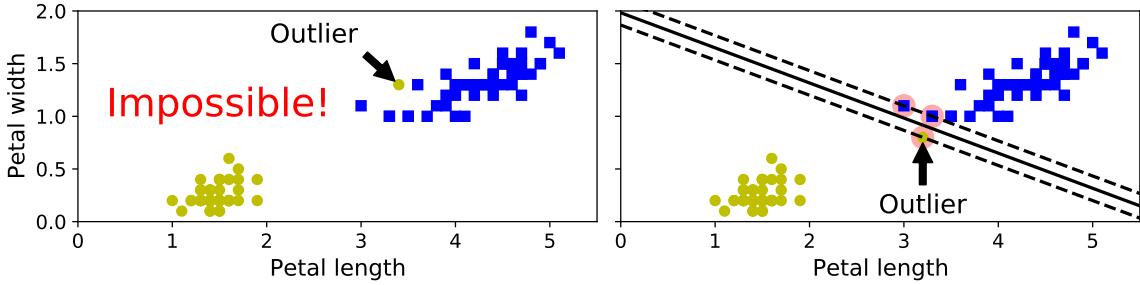


Figure 4.68

When creating an SVM model using Scikit-Learn (e.g. using the function `LinearSVC`), we can specify a number of hyperparameters. C is one of those hyperparameters. If we set it to a low value, then we end up with the model on the left of Figure 4.69. With a high value, we get the model on the right. Margin violations are bad. It's usually better to have only few of them. However, in this case the model on the left has a lot of margin violations but will probably generalize better. If your SVM model is overfitting, you can try regularizing it by reducing C .

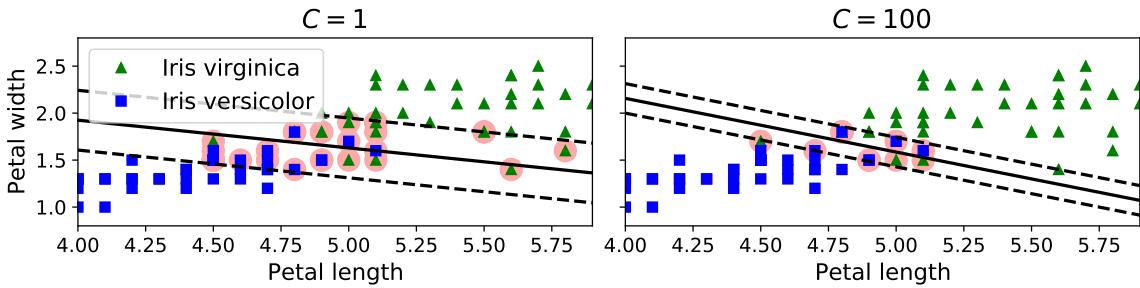


Figure 4.69

4.6.2 Nonlinear 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 nonlinear datasets is to add more features, such as polynomial features (as we did in Sec. 4.5.3); in some cases this can result in a linearly separable dataset. Consider the left plot in Figure 4.70: it represents a simple dataset with just one feature, x_1 . This dataset is clearly 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. Note that we are talking about “linear” separability, because so far we have treated classes described by just two features; but actually, these “lines” must be considered as hyperplanes in a larger ambient space. More generally, we say that an ensemble of instances characterized by n features is linearly separable if two classes can be separated by an $(n - 1)$ -dimensional hyperplane. Therefore, for the left panel the hyperplane is just a dot, while for the right one is a straight line.

To implement this idea using Scikit-Learn, we create a `Pipeline` containing a `PolynomialFeatures` transformer (discussed in “Polynomial regression”) of order 3, followed by a `StandardScaler` and a `LinearSVC` (Fig. 4.71). Let's test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles (see Figure 4.70). You can generate this dataset using the `make_moons()` function. We also added some random noise.

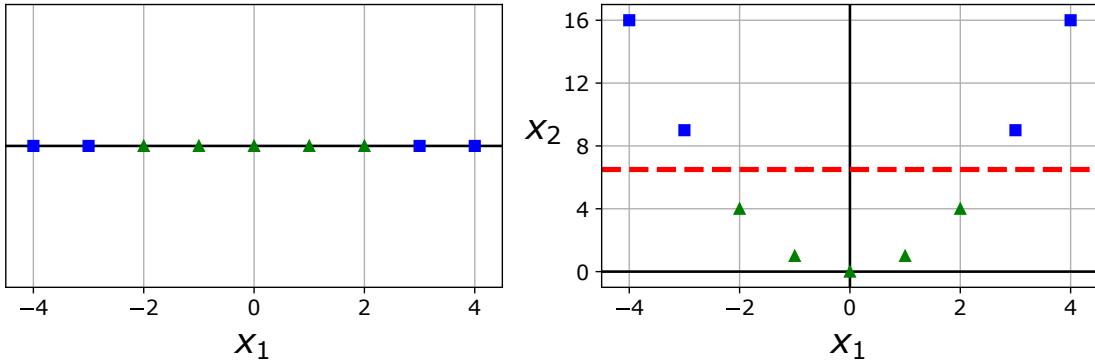


Figure 4.70

```
In [13]: 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", random_state=42))
])

polynomial_svm_clf.fit(X, y)
```

Figure 4.71

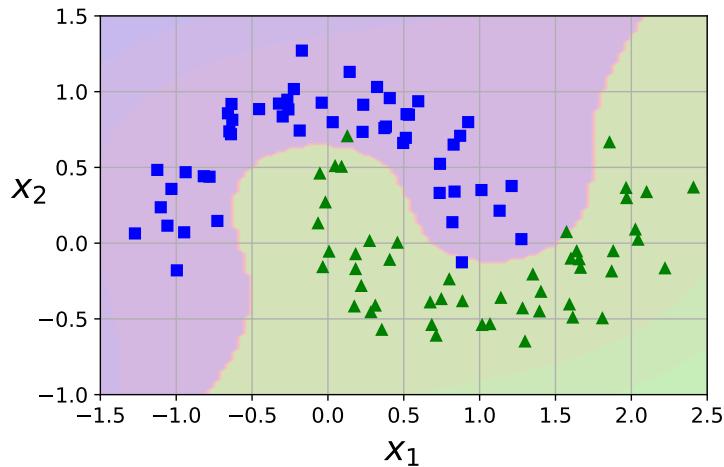


Figure 4.72

4.6.2.1 Polynomial kernel

Adding polynomial features is simple to implement and can work great with all sorts of machine learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow. Fortunately, when using SVMs we can apply an almost miraculous mathematical technique called the *kernel trick* (explained in a moment). The kernel trick makes it possible to get the same result as if you had 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

4 Machine learning techniques

features because you don't actually add any features. This trick is implemented by the `SVC` class.

Let's test it on the moons dataset (Fig. 4.73). The left panel represents an SVM classifier using a third-degree polynomial kernel. On the right is another SVM classifier using instead a 10th-degree polynomial kernel. As we can notice, the plot on the right takes into account all the outliers; there the boundary is more complicated, and hence we risk to overfit and to generalize badly. 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` (r in the figure) controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

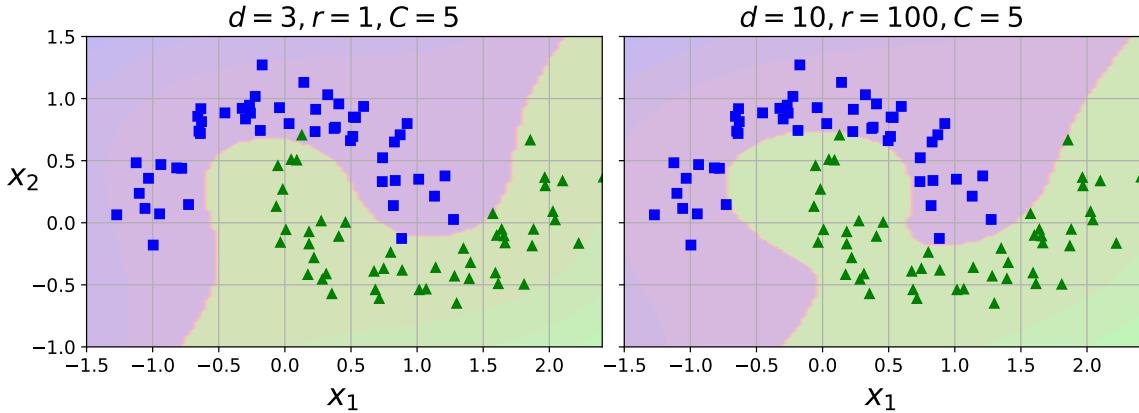


Figure 4.73

4.6.3 Mathematical formulation

In this section we will explain how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. First, a word about notations. So far 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 paragraph we will use a convention that is more convenient (and more common) when dealing with SVMs: the bias term will be called b , and the feature weights vector will be called w . No bias feature will be added to the input feature vectors.

4.6.3.1 Decision function and predictions

The linear SVM classifier model predicts the class of a new instance x by simply computing the *decision function* $w^T x + b = w_1 x_1 + \dots + w_n x_n + b$. If the result is positive, the predicted class \hat{y} is the positive class (1), and otherwise it is the negative class (0):

$$\hat{y} = \begin{cases} 0, & \text{if } w^T x + b < 0 \\ 1, & \text{if } w^T x + b \geq 0 \end{cases} \quad (4.6.1)$$

Again, notice the similarity between SVMs and perceptrons: also in that case we first checked if a certain expression was positive or negative, and then we associated to the neurons the corresponding response. The only difference is that in neural networks we preferred to use -1 and $+1$ to indicate the activity of a neurons, while now we are using 0 and 1. However, this is not a problem because we can easily map one convention to another in a unique way.

Figure 4.74 shows the decision function that corresponds to the model in the left in Figure 4.69: it is a 2D plane because this dataset has two 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). The dashed lines represent the points \mathbf{x} where the decision function is equal to 1 or -1 : they are parallel and at equal distance to the decision boundary, and they form a margin around it. Training a linear SVM classifier means finding the values of \mathbf{w} and b that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

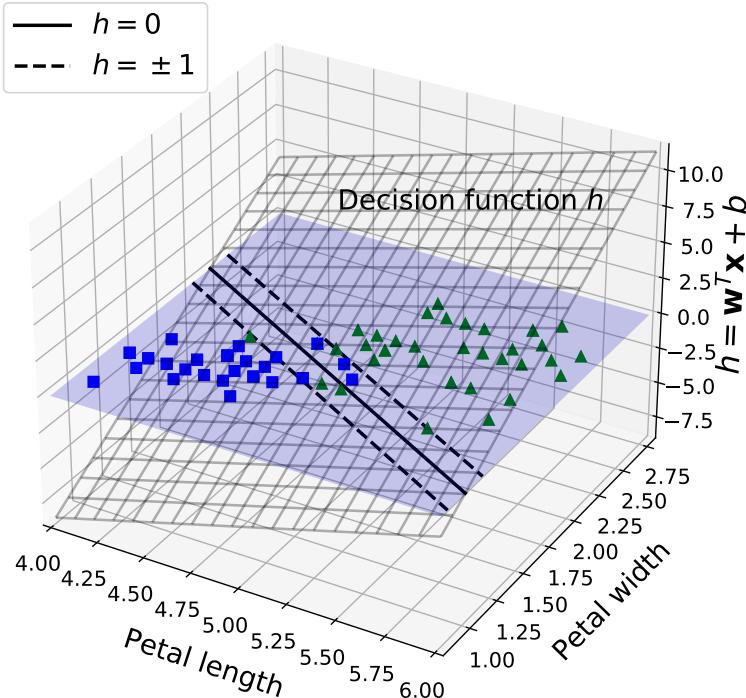


Figure 4.74

4.6.3.2 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 as far away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. This may be easier to visualize in 2D, as shown in Figure 4.75. The smaller the weight vector \mathbf{w} , the larger the margin.

So we want to minimize $\|\mathbf{w}\|$ to get a large margin. If we also want to avoid any margin violations (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 $t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ for all instances. We can therefore express the hard margin linear SVM classifier objective as the constrained optimization problem in

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} \\ & \text{subject to} \quad t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

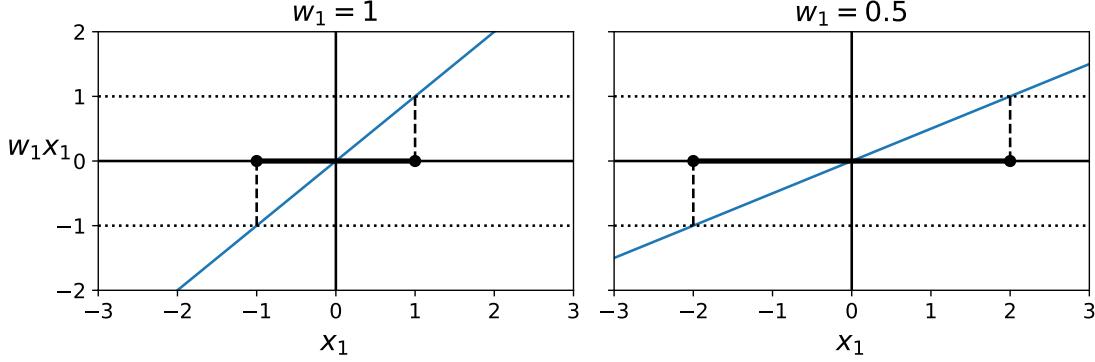


Figure 4.75

We are minimizing $\frac{1}{2}\mathbf{w}^T\mathbf{w}$, which is equal to $\frac{1}{2}\|\mathbf{w}\|^2$, rather than minimizing $\|\mathbf{w}\|$. Indeed, $\frac{1}{2}\|\mathbf{w}\|^2$ has a nice, simple derivative (it is just \mathbf{w}), while $\|\mathbf{w}\|$ is not differentiable at $\mathbf{w} = 0$. 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 instance: $\zeta^{(i)}$ measures how much the i -th instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ as small as possible to increase the margin. This is where the C hyperparameter (amount of non-violation) comes in: it allows us to define the tradeoff between these two objectives. This gives us the constrained optimization problem

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} \quad t^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

4.6.3.3 The dual problem

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. 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 have the same solution as the primal problem. Luckily, the SVM problem happens to meet these conditions, so you can choose to solve the primal problem or the dual problem; both will have the same solution. It can be demonstrated that the dual form of the linear SVM objective is

$$\begin{aligned} & \underset{\alpha}{\text{minimize}} \quad \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)}{}^T \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ & \text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Once we find the vector $\hat{\boldsymbol{\alpha}}$ that minimizes this equation ($\boldsymbol{\alpha}$ is a free parameter), we exploit the following equations to compute $\hat{\mathbf{w}}$ and \hat{b} that minimize the primal problem:

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \tag{4.6.2}$$

$$\hat{b} = \frac{1}{m} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \right) \tag{4.6.3}$$

The dual problem is faster to solve than the primal one when the number of training instances is smaller than the number of features. More importantly, the dual problem makes the kernel trick possible, while the primal does not. So what is this kernel trick, anyway?

4.6.4 Kernelized SVMs

In the previous sections we have said that not all datasets are linearly separable and a possible solution was adding more features, such as polynomial features. Practically, we apply a polynomial mapping to the initial set of features and the resulting values are the new features that we add to the initial dataset. Clearly this works, but for high-order polynomials we create a huge number of features because we must not simply consider the powers of the individual features, but also the combined products between different features (e.g. x_1x_2). Accordingly, the number of features increases combinatorially, thus making the algorithm computationally heavy. However, if we focus on the dual problem instead of the primal one, such a problem is automatically eliminated. This is called the *kernel trick*.

Let's see why mathematically the kernel trick actually solves our problems. Suppose you want to apply a second-degree polynomial transformation to a two-dimensional training set (such as the moons training set), then train a linear SVM classifier on the transformed training set. The following equation shows the second-degree polynomial mapping function Φ that you want to apply:

$$\Phi(\mathbf{x}) = \Phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix} \quad (4.6.4)$$

Notice that the transformed vector (i.e. containing the new features) is 3D instead of 2D. Now let's look at what happens to a couple of 2D vectors, \mathbf{a} and \mathbf{b} , if we apply this second-degree polynomial mapping and then compute the dot product of the transformed vectors:

$$\begin{aligned} \Phi(\mathbf{a})^T \Phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} \\ &= a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 \\ &= (a_1b_1 + a_2b_2)^2 \\ &= \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 \\ &= (\mathbf{a}^T \mathbf{b})^2 \end{aligned} \quad (4.6.5)$$

How about that? The dot product of the transformed vectors is equal to the square of the dot product of the original vectors: $\Phi(\mathbf{a})^T \Phi(\mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$. Here is the key insight: if you apply the transformation Φ to all training instances, then the dual problem

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)}^T \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ \text{subject to} \quad & \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

will contain the dot product $\Phi(\mathbf{x}^{(i)})^T \Phi(\mathbf{x}^{(j)})$, instead of $\mathbf{x}^{(i)}^T \mathbf{x}^{(j)}$. But if Φ is the second-degree polynomial transformation defined in equation (4.6.4), then you can replace this

dot product of transformed vectors simply by $(\mathbf{x}^{(i)T} \mathbf{x}^{(j)})^2$. So, *you don't need to transform the training instances at all*; just replace the dot product by its square. The result will be strictly the same as if you had gone through the trouble of transforming the training set then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient. This trick can be applied to whatever degree, resulting in a power p in the final result.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ is a second-degree polynomial kernel. In machine learning, a *kernel* is a function capable of computing the dot product $\Phi(\mathbf{a})^T \Phi(\mathbf{b})$, based only on the original vectors \mathbf{a} and \mathbf{b} , without having to compute (or even to know about) the transformation Φ . This is the essence of *Mercer's theorem*.

4.7 Dimensionality reduction

Many machine learning problems involve thousands or even millions of features for each training instance. For instance, this occurs when we try to digitize an image: we divide the real image in a grid of pixels and each pixel's intensity, from 0 (white) to 255 (black), simply represents a different feature. So, it is obvious that for an image with a resolution of 100×100 pixels we already have 10 000 features, which is a huge number! Not only do all these features make training extremely slow, but they 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 digitized images: 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. These pixels are utterly unimportant for the classification task. Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information. So, identifying the most important feature helps against the problem of dimensionality. Nevertheless, keep in mind that reducing dimensionality does cause some information loss (just like compressing a raw image to JPEG can degrade its quality), so even though it will speed up training, it may make your system perform slightly worse.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization. Reducing the number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters.

In this section we will discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then, we will consider 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 example are taken from [→](#).

4.7.1 The curse of dimensionality

We are so used to living in three spatial dimensions that our intuition fails us when we try to imagine a high-dimensional space. 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 1×1 square), it will have only about a 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, this probability is greater than 99.999999 %. Most points in a high-dimensional hypercube are very close to the border (we already seen something similar in Sec. 2.3.3).

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 1 000 000-dimensional hypercube? The average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1\ 000\ 000}/6$)! This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, there's just plenty of space in high dimensions. As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. 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 short, the more dimensions the training set has, the greater the risk of overfitting it.

4.7.2 Main approaches for dimensionality reduction

4.7.2.1 Projection

In most real-world problems, training instances are not spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, *all training instances 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 4.76a you can see a 3D dataset represented by circles. Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space. If we project every training instance perpendicularly onto this subspace (as represented by the short lines connecting the instances to the plane), we get the new 2D dataset shown in Figure 4.76b. By doing this, 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).

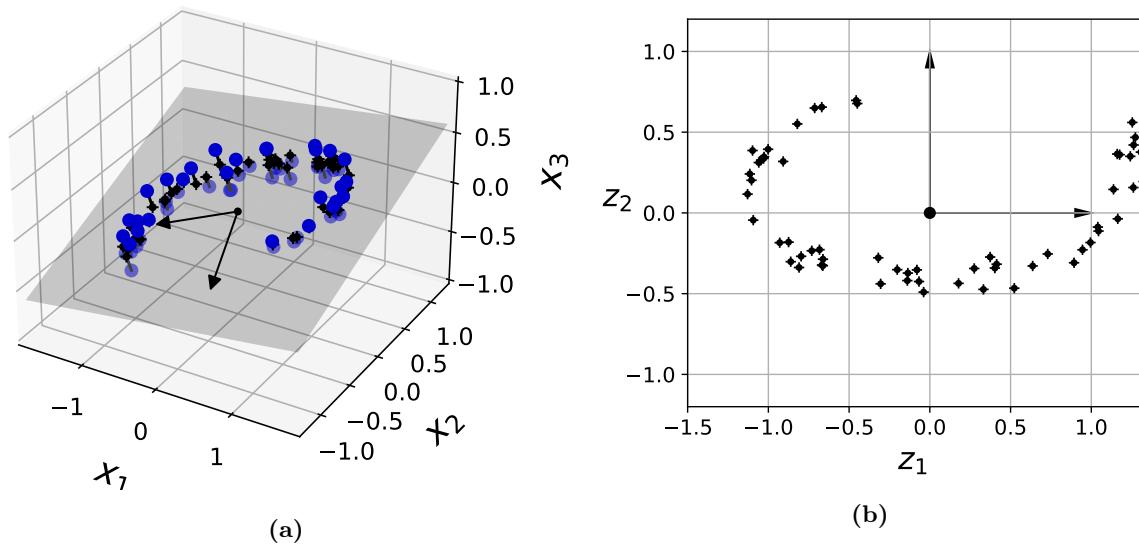


Figure 4.76

However, projection is not always the best approach to dimensionality reduction. In

4 Machine learning techniques

many cases the subspace may twist and turn, such as in the famous *Swiss roll* toy dataset represented in Figure 4.77. In this case we have four features: x_1 , x_2 , x_3 and the color of the points. Simply projecting onto a plane (e.g., by dropping x_3 , i.e. seeing the picture from above) would squash different layers of the Swiss roll together, as shown on the left side of Figure 4.78. What you really want is to “unroll” the Swiss roll to obtain the 2D dataset on the right side.

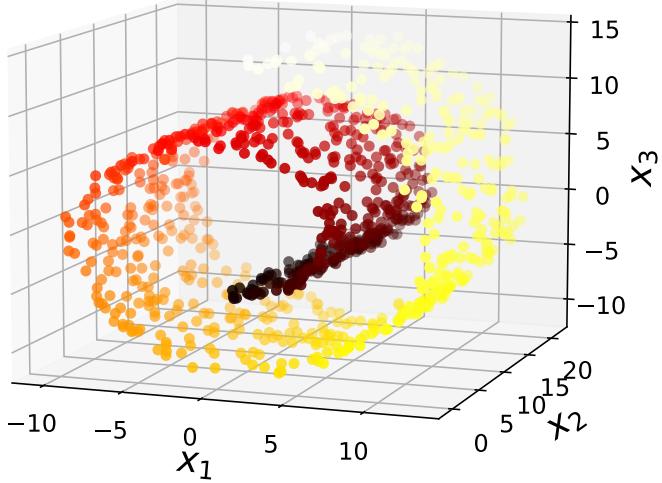


Figure 4.77

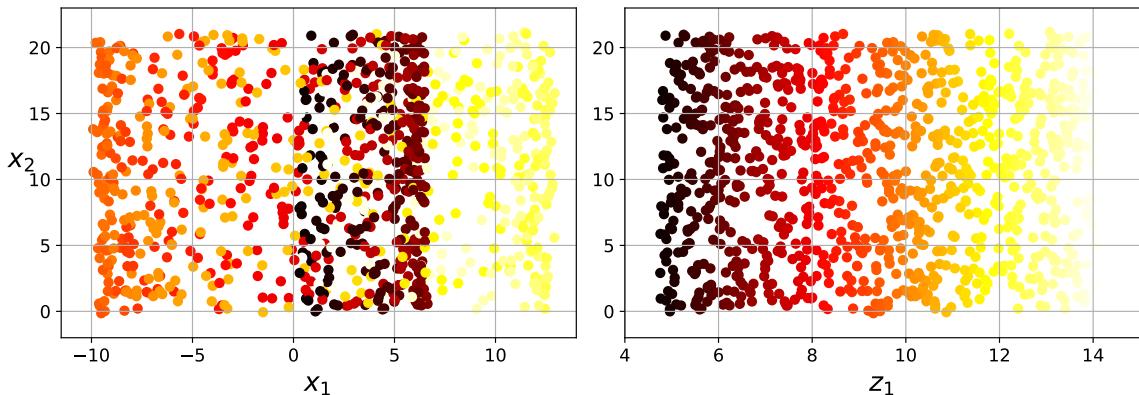


Figure 4.78: Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right).

4.7.2.2 Manifold learning

The Swiss roll is an example of a 2D manifold. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. 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, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure 4.79 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 just a straight line. However, this implicit assumption does not always hold. For example, in the bottom row of Figure 4.79, the decision boundary is located at $x = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

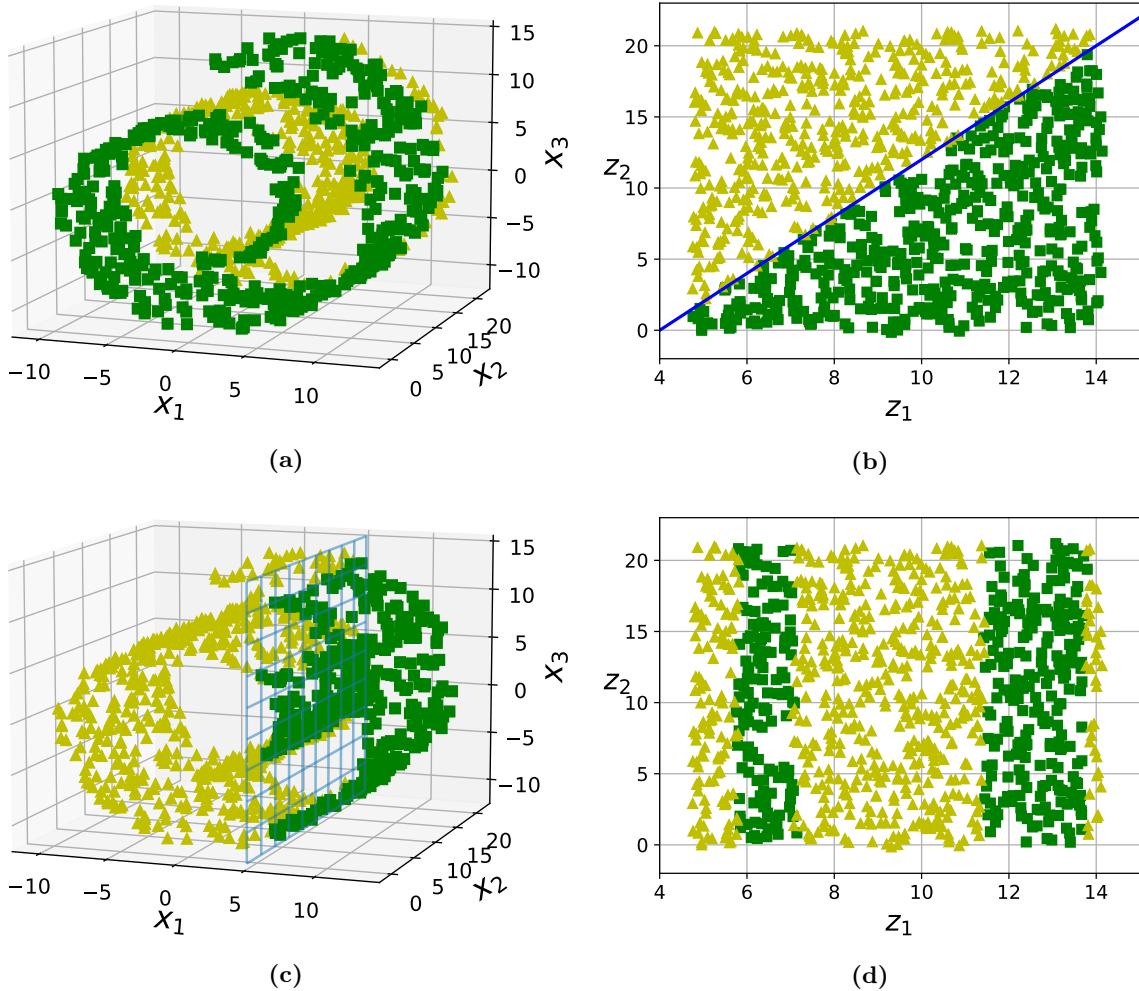


Figure 4.79: The decision boundary may not always be simpler with lower dimensions.

In short, reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

4.7.3 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, and then it projects the data onto it. 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 in Figure 4.80, along with three different axes (i.e., 1D 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 variance, while the projection onto the dotted line preserves very little variance and the projection onto the dashed line

preserves an intermediate amount of variance. 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.

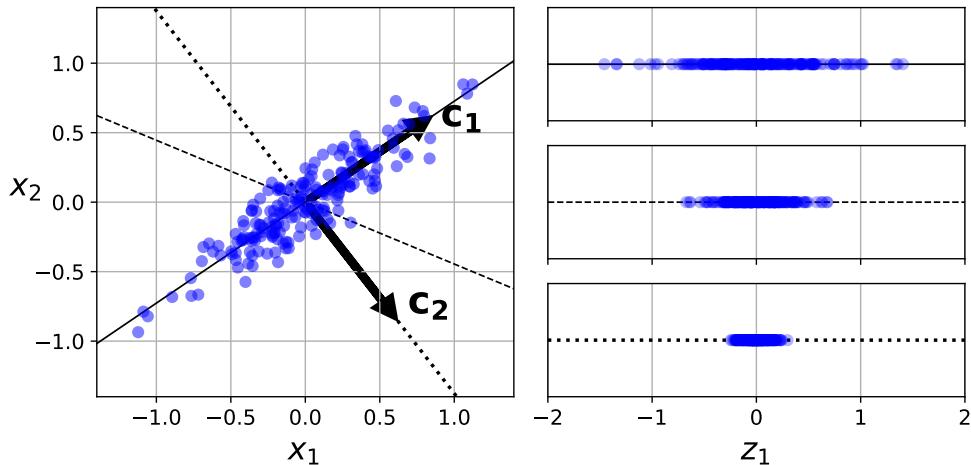


Figure 4.80

PCA identifies the axis that accounts for the largest amount of variance in the training set. In the previous figure, it is the solid line. It also finds a second axis, orthogonal to the first one, that 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 i -th axis is called the i -th *principal component* (PC) of the data. In Figure 4.80, the first PC is the axis on which vector \mathbf{c}_1 lies, and the second PC is the axis on which vector \mathbf{c}_2 lies.

4.7.3.1 Determination of the principal components

Now, how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called **singular value decomposition** (SVD) which do the trick. The singular value decomposition is a factorization of a real or complex matrix that generalizes the eigendecomposition of a square normal matrix to any $m \times n$ matrix via an extension of the polar decomposition. In contrast to the eigenvalue decomposition, the SVD of a matrix always exists. Specifically, the singular value decomposition of an $m \times n$ complex matrix \mathbf{X} is a factorization of the form $\mathbf{U}\Sigma\mathbf{V}^\dagger$, where \mathbf{U} is an $m \times m$ complex unitary matrix, Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and \mathbf{V} is an $n \times n$ complex unitary matrix (e.g. for an Hermitian matrix $H = VDV^\dagger$). In our case the training set matrix \mathbf{X} is real, hence factorizations where \mathbf{U} and \mathbf{V} are real orthogonal matrices exist. Accordingly, all the matrices involved in the decomposition are real and we end up with the simpler form

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T \quad (4.7.1)$$

where \mathbf{V} contains the unit vectors that define all the principal components that we are looking for

$$\mathbf{V} = \begin{pmatrix} & & & \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ & & & \end{pmatrix} \quad (4.7.2)$$

Notice that we have said that Σ is diagonal, but in general it may not be a square matrix. So what does it mean to be diagonal? A rectangular matrix Σ is said to be diagonal if all the entries are 0 but for the ones of the main diagonal, i.e. except for the entries $\sigma_i := \Sigma_{i,i}$. For the sake of clarity, the following are two examples of rectangular diagonal matrices:

$$\begin{pmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \end{pmatrix} \quad \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \\ 0 & 0 & 0 \end{pmatrix} \quad (4.7.3)$$

Such non-null values are called *singular values* and play the same role as the eigenvalues for a square matrix. So, if for square matrix eigendecomposition the diagonal matrix contained the eigenvalues and the unitary transformation matrix contained the corresponding eigenvectors, now the matrix Σ contains the singular values (typically ordered) and \mathbf{V} the vectors defining the principal components. From the geometrical point of view, the SVD can be seen as a rotation or reflection (\mathbf{V}^T), followed by a coordinate-by-coordinate scaling (Σ), followed by another rotation or reflection (\mathbf{U}). So \mathbf{V} contains the information of the rotation to maximize the variance.

Anyway, in Python singular value decomposition can be implemented using the NumPy function `np.linalg.svd()`. This function gives us the three matrices involved in the decomposition; to obtain all the principal components $\mathbf{c}_1, \mathbf{c}_2$ of the training set, we then extracts the two unit vectors from the latter. Note that PCA assumes that the dataset is *centered* around the origin by subtracting the mean to each value (Fig. 4.81). So, if you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first. Another possibility is to use the `PCA` function already implemented in Scikit-Learn, which automatically takes care of centering the data for you.

```
In [3]: X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

Figure 4.81: NumPy implementation.

In Figure 4.82 are shown the results of this analysis using Scikit-Learn and Numpy. The result are identical, apart for the sign. In fact, for each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC. Since two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is *not stable*: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap (if the variances along these two axes are close), but the plane they define will generally remain the same.

Another useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For the i -th

```
In [9]: X2D[:5]
Out[9]: array([[ 1.26203346,  0.42067648],
   [-0.08001485, -0.35272239],
   [ 1.17545763,  0.36085729],
   [ 0.89305601, -0.30862856],
   [ 0.73016287, -0.25404049]])
```



```
In [10]: X2D_using_svd[:5]
Out[10]: array([[-1.26203346, -0.42067648],
   [ 0.08001485,  0.35272239],
   [-1.17545763, -0.36085729],
   [-0.89305601,  0.30862856],
   [-0.73016287,  0.25404049]])
```

Figure 4.82: Scikit-Learn (`In[9]`) and Numpy (`In[10]`) results.

component it is computed by using the entries σ_i of the rectangular diagonal matrix Σ :

$$\alpha_i = \frac{\sigma_i^2}{\sum_{j=1}^{\min\{m,n\}} \sigma_j^2} \quad (4.7.4)$$

In other words, the explained variance ratio gives us the relative contributions of the various principal components to the total variance. For instance, the output of the notebook tells us that 84.2 % of the dataset's variance lies along the first PC, while 14.6 % lies along the second PC. This leaves less than 1.2 % for the third PC (orthogonal to the other), so it is reasonable to assume that the third PC probably carries little information.

4.7.3.2 Choosing the right number of dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95 %), unless, of course, you are reducing dimensionality for data visualization.

An option is to plot the explained variance as a function of the number of dimensions. Practically, we take the array of the explained variance ratios and we check what happens if we sum the first two elements ($n = 2$ dimensions), the first three ($n = 3$ dimensions), and so on; this is achieved in NumPy using first a cumulative sum `np.cumsum` and then the argmax operator `np.argmax`. The goal is to find a range of dimensions such that the sum of the corresponding explained variance ratios is bigger than 0.95. As we can see from Fig. 4.83, there will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

4.7.3.3 PCA for compression

After dimensionality reduction, the training set takes up much less space. As an example, try applying PCA to the MNIST dataset (handwritten digits) 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 size reduction can speed up a classification algorithm (such as an SVM classifier) tremendously.

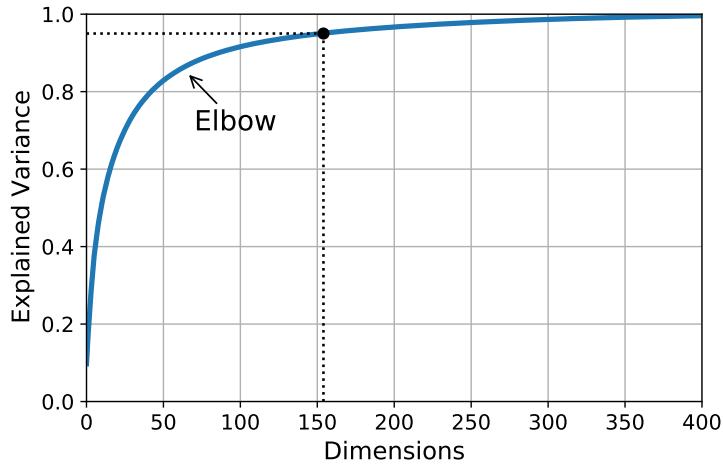


Figure 4.83

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. However, 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 close to the original data.

Figure 4.84 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, i.e. a sort of blurred halo, but the digits are still mostly intact.

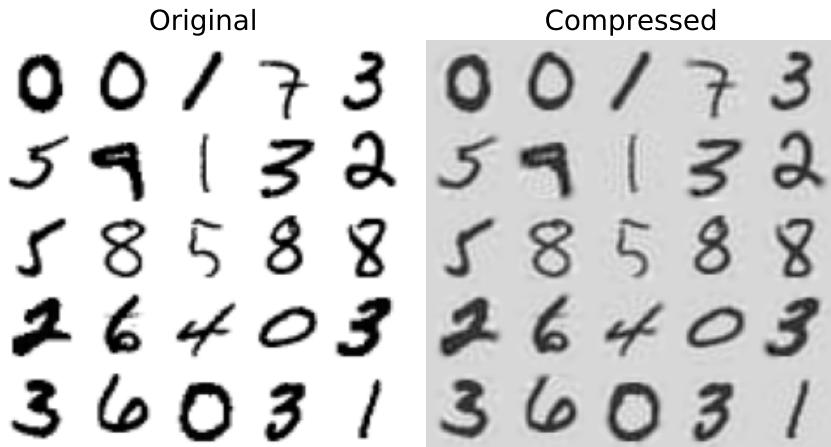


Figure 4.84

4.7.4 Kernel PCA

In Section 4.6.4 we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with support vector machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space.

It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called **kernel PCA** (kPCA). It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

4.7.5 LLE

Locally linear embedding (LLE) is another powerful nonlinear dimensionality reduction technique. It is a manifold learning technique that does not rely on projections, like the previous algorithms do. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

For example, the 2D dataset resulting from the Swiss roll seen before in Figure 4.77 is shown in Figure 4.85. 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 stretched, while the right part is squeezed. Nevertheless, LLE did a pretty good job at modeling the manifold.

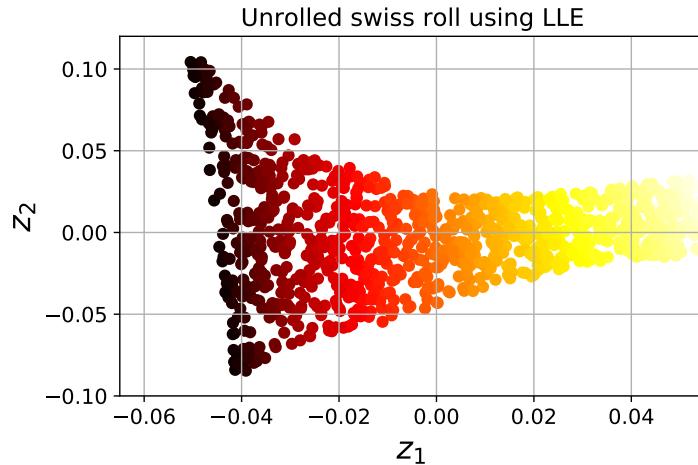


Figure 4.85

Here's how LLE works: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k closest neighbors, then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it finds the weights w_{ij} such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^m w_{ij}\mathbf{x}^{(j)}$ is as small as possible, assuming $w_{ij} = 0$ 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

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \left[\sum_{i=1}^m \left(\mathbf{x}^{(i)} - \sum_{j=1}^m w_{ij} \mathbf{x}^{(j)} \right)^2 \right]$$

subject to

$$\begin{cases} w_{ij} = 0, & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{ij} = 1, & \text{for } i = 1, 2, \dots, m \end{cases}$$

where \mathbf{W} is the weight matrix containing all the weights w_{ij} . The second constraint simply normalizes the weights for each training instance. After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the optimal weights \widehat{w}_{ij}) encodes the local linear relationships between the training instances. 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 $\sum_{j=1}^m \hat{w}_{ij} \mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \left[\sum_{i=1}^m \left(\mathbf{z}^{(i)} - \sum_{j=1}^m \hat{w}_{ij} \mathbf{z}^{(j)} \right)^2 \right] \quad (4.7.5)$$

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 $\mathbf{z}^{(i)}$.

4.8 Unsupervised learning techniques

In the previous section we looked at the most common unsupervised learning task: dimensionality reduction. In this section we will look at a few more unsupervised learning tasks and algorithms: clustering, anomaly detection and density estimation .

4.8.1 Clustering

The clustering is the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances. Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task. Consider Figure 4.86: on the left is the Iris dataset, where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as logistic regression or SVMs classifiers are well suited; they are trained on a labeled set and then they are able to generalize to new instances. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore because we don't know to which specific class an instance belongs to. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two distinct sub-clusters. That said, the dataset has two additional features (sepal length and width), not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).

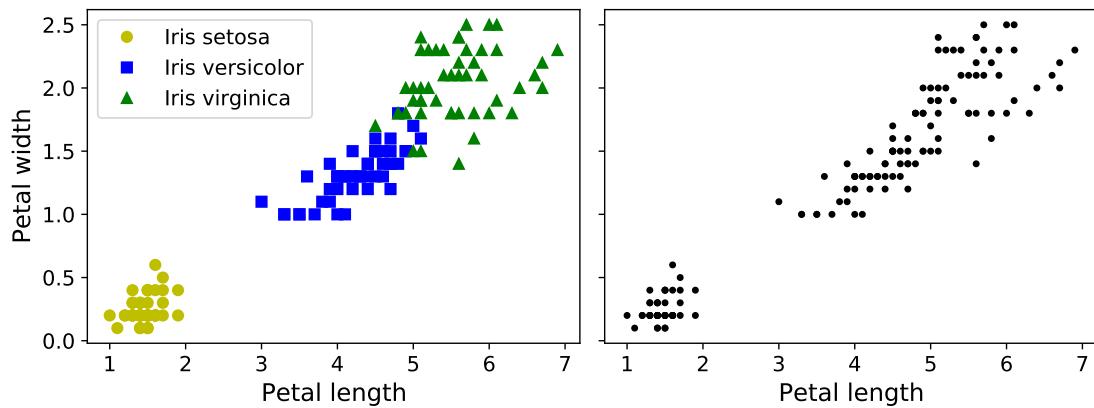


Figure 4.86

4.8.1.1 K-means

Consider the unlabeled dataset represented in Figure 4.87: you can clearly see five blobs of instances. The **K-means** algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. The starting point is that you *have to specify the number of clusters k that the algorithm must find*. In this example, it is pretty obvious from looking at the data that k should be set to 5, but in general it is not that easy. We will discuss this shortly.

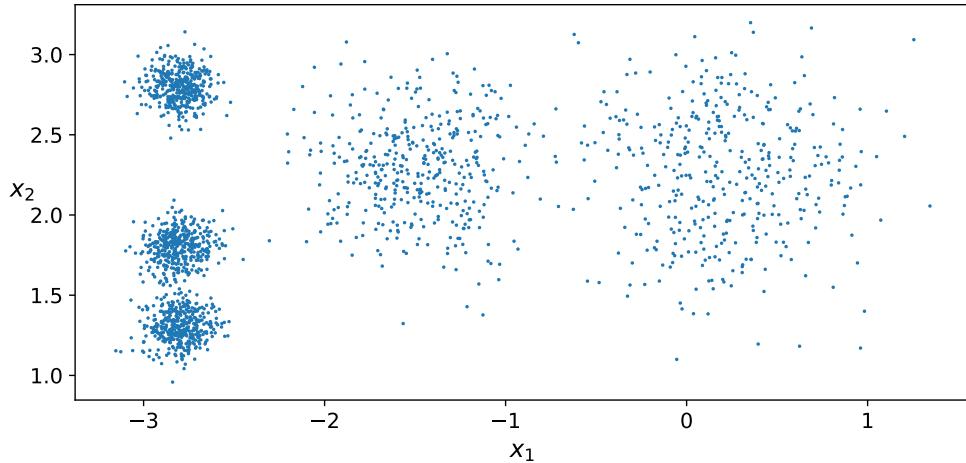


Figure 4.87

Let's train a K-means clusterer on this dataset. The basic idea is to find each blob's center, thereafter called *centroid*, and assign each instance to the closest blob. In this way, each instance is assigned a label corresponding to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster that this instance gets assigned to by the algorithm: this is not to be confused with the class labels in classification (remember that clustering is an unsupervised learning task). If you plot the cluster's decision boundaries, i.e. delimiting the regions consisting of all points of the plane closer to that seed centroid than to any other, you get a *Voronoi tessellation* (see for example Figure 4.88, where each centroid is represented with an **X**).

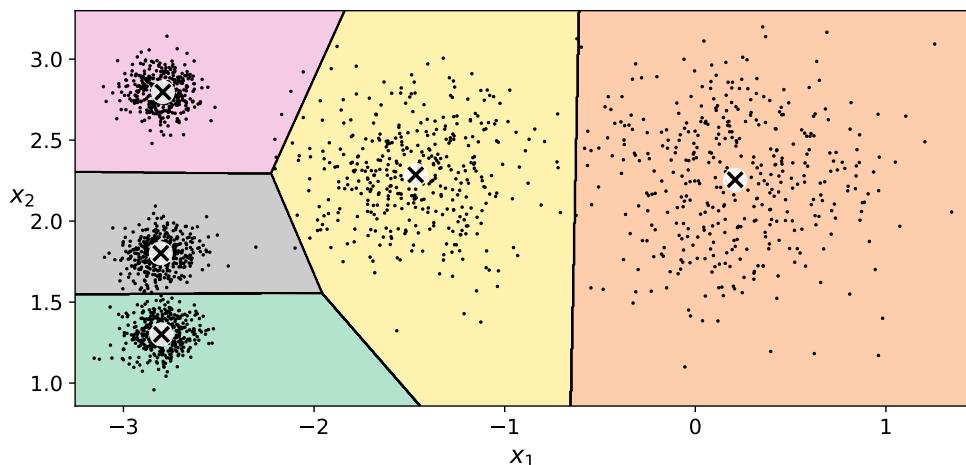


Figure 4.88

The vast majority of the instances were clearly assigned to the appropriate cluster, but

a few instances were probably mislabeled (especially near the boundary between the top-left cluster and the central cluster). Indeed, the K-means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to give each instance a score per cluster, which is called *soft clustering*. The score can be the distance between the instance and the centroid; conversely, it can be a similarity score (or affinity). In the Scikit-Learn’s `KMeans` class, the `transform()` method measures the distance from each instance to every centroid.

The algorithm So, how does the algorithm work? Well, suppose you were given the centroids. You could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate all the centroids by computing the mean of the instances for each cluster. But you are given neither the labels nor the centroids, so how can you proceed? Well, just start by placing the centroids randomly (e.g., by picking k instances at random and using their locations as centroids). Then label the instances according to the distance, update the centroids computing the mean, label again the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small); it will not oscillate forever.

You can see the algorithm in action in Figure 4.89: the centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can see, in just three iterations, the algorithm has reached a clustering that seems close to optimal.

In the original K-means algorithm, the centroids are just initialized randomly, and the algorithm simply runs a single iteration to gradually improve the centroids, as we saw above. However, one major problem with this approach is that if you run K-means multiple times (or with different random seeds), it can converge to very different solutions. In other words, although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization. Figure 4.90 shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

Let’s look at a few ways you can mitigate this risk by improving the centroid initialization.

Centroid initialization methods If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set those points for the centroid initialization.

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the `n_init` hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier runs 10 times and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? It uses a *performance metric*! That metric is called the model’s *inertia*, which is the mean squared distance between each instance and its closest centroid. It is roughly equal to 223.3 for the model on the left in Figure 4.90, 237.5 for the model on the right in Figure 4.90, and 211.6 for the model in Figure 4.88. The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia. In this example, the model in Figure 4.88 will be selected (unless we are very unlucky with `n_init` consecutive random initializations). If you are curious, a model’s inertia is accessible via the `inertia_` instance variable.

4 Machine learning techniques

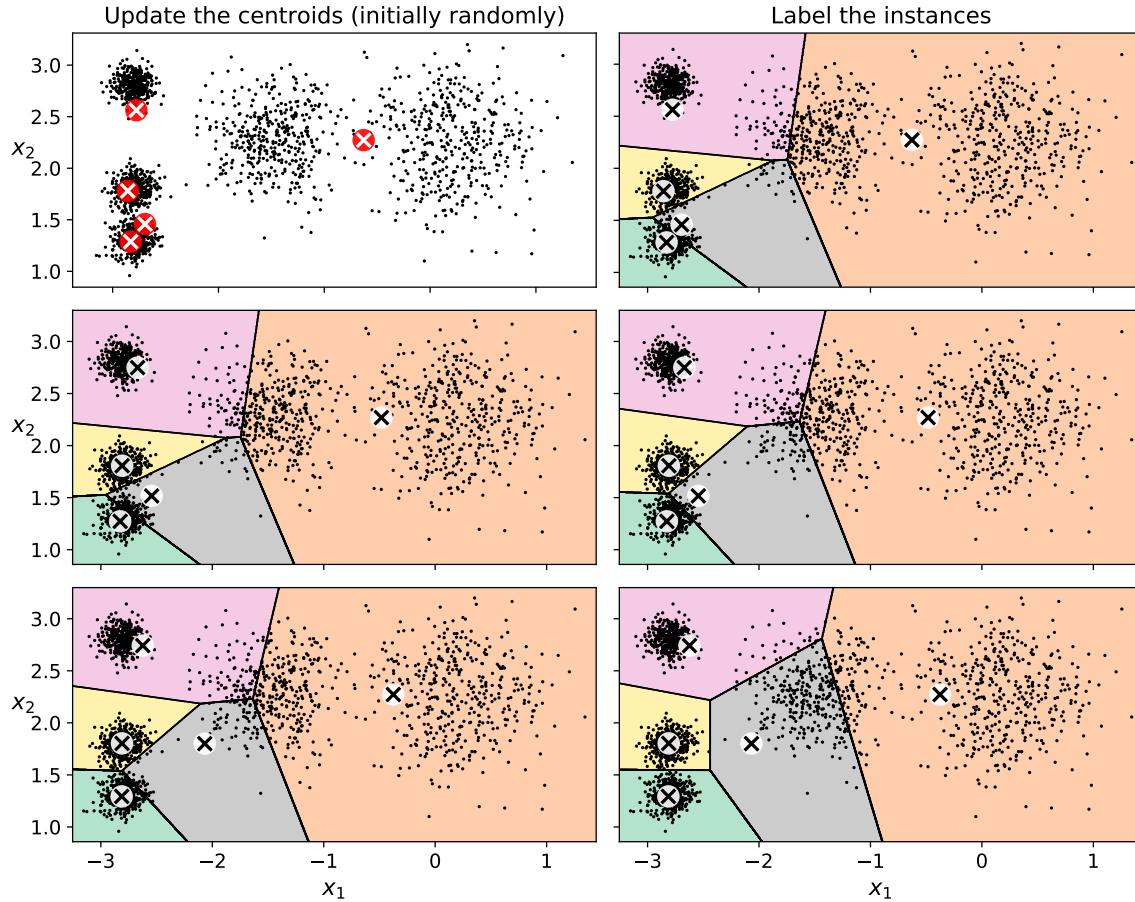


Figure 4.89

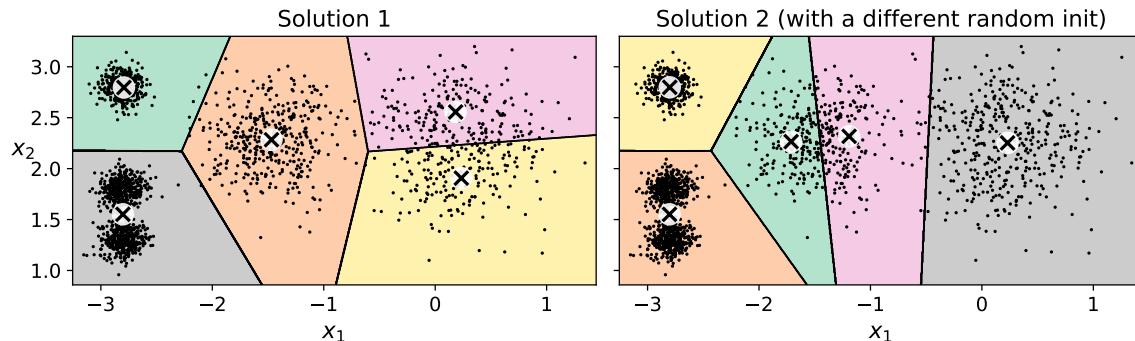


Figure 4.90

The `score()` method returns the negative inertia. Why negative? Because a predictor's `score()` method must always respect Scikit-Learn's “greater is better” rule: if a predictor is better than another, its `score()` method should return a greater score (Fig. 4.91).

K-means++ An important improvement to the Kmeans algorithm, *K-means++*, was proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii. They introduced a smarter initialization step that tends to select centroids that are distant from one another, and this improvement makes the K-means algorithm much less likely to converge to a suboptimal solution. They showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the

```
In [32]: kmeans.inertia_
Out[32]: 211.5985372581683

In [33]: X_dist = kmeans.transform(X)
          np.sum(X_dist[np.arange(len(X_dist)), kmeans.labels_]**2)
Out[33]: 211.5985372581684

In [34]: kmeans.score(X)
Out[34]: -211.5985372581683
```

Figure 4.91

number of times the algorithm needs to be run to find the optimal solution. Here is the K-means++ initialization algorithm:

- Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset $\{\mathbf{x}^{(i)}\}$.
- Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability

$$\Pr(\mathbf{x}^{(i)}) = \frac{D(\mathbf{x}^{(i)})^2}{\sum_{j=1}^m D(\mathbf{x}^{(j)})^2} \quad (4.8.1)$$

where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid among the ones that were already chosen. In other words, for each of the instances of the dataset we first determine the closest centroid from the ensemble of centroids already at disposal; then, we compute and associate to every instance a probability according to equation (4.8.1); finally, we extract a random number uniformly distributed between 0 and 1, and according to its value we insert the corresponding instance in the set of centroids already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely be selected as centroids.

- Repeat the previous step until all k centroids have been chosen.

The rest of the K-means++ algorithm is just regular K-means. The `KMeans` class uses this initialization method by default. If you want to force it to use the original method (i.e., picking k instances randomly to define the initial centroids), then you can set the `init` hyperparameter to "random". You will rarely need to do this.

Finding the optimal number of clusters So far, we have set the number of clusters k to 5 because it was obvious by looking at the data that this was the correct number of clusters. But in general, it will not be so easy to know how to set k , and the result might be quite bad if you set it to the wrong value. As you can see in Figure 4.92, setting k to 3 or 8 results in fairly bad models. When k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces.

You might be thinking that we could just pick the model with the lowest inertia, right? Unfortunately, it is not that simple. The inertia for $k = 3$ is 653.2, which is much higher than for $k = 5$ (which was 211.6). But with $k = 8$, the inertia is just 119.1. So, the inertia is not a good performance metric when trying to choose k because it keeps getting lower as

4 Machine learning techniques

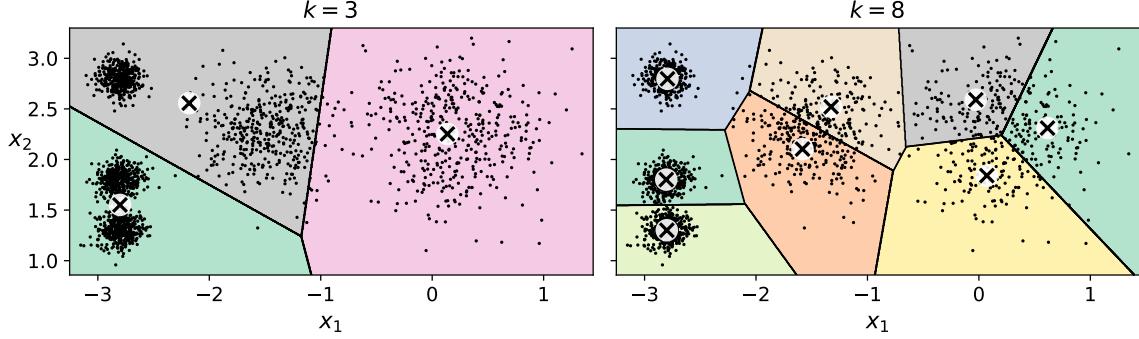


Figure 4.92

we increase k . Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be.

Let's plot the inertia as a function of k (see Fig. 4.93). As you can see, the inertia drops very quickly as we increase k up to 4, but then it decreases much more slowly as we keep increasing k . Finally, for $k = m$ it will necessarily reach zero because in that case we would have a number of clusters equal to the number of instances, i.e. every instance will be a different centroid, and so the inertia will vanish. This curve has roughly the shape of an arm, and there is an "elbow" at $k = 4$. So, if we did not know better, 4 would be a good choice: any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

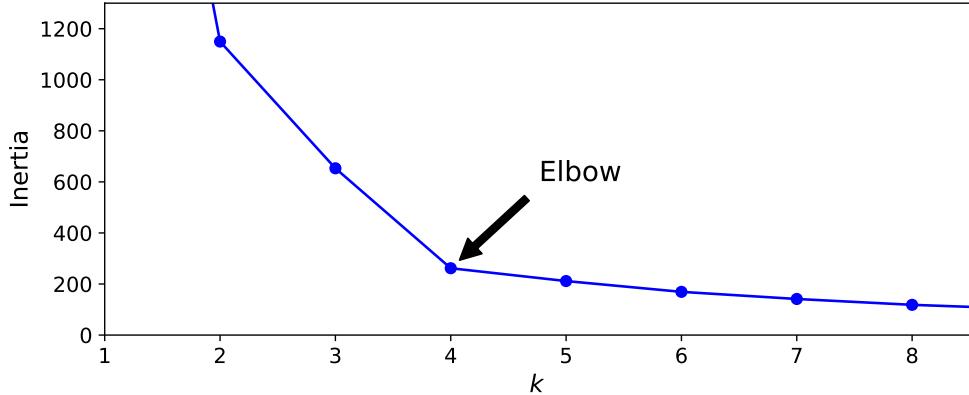


Figure 4.93

This technique for choosing the best value for the number of clusters is rather coarse. A more precise approach (but also more computationally expensive) is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to

$$\text{Silhouette coefficient} = \frac{b - a}{\max \{a, b\}} \quad (4.8.2)$$

where a is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and b is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes b , excluding the instance's own cluster). The silhouette coefficient can vary between -1 and $+1$. A coefficient close to $+1$ means that the instance is well inside its own cluster and far from other clusters (i.e. small a and big b), while a coefficient close to 0 means that it is close to

a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned. Let's compare the silhouette scores for different numbers of clusters k (see Fig. 4.94). As you can see, this visualization is much richer than the previous one: although it confirms that $k = 4$ is a very good choice, it also underlines the fact that $k = 5$ is quite good as well, and much better than $k = 6$ or 7 , for instance. This was not visible when comparing inertias.

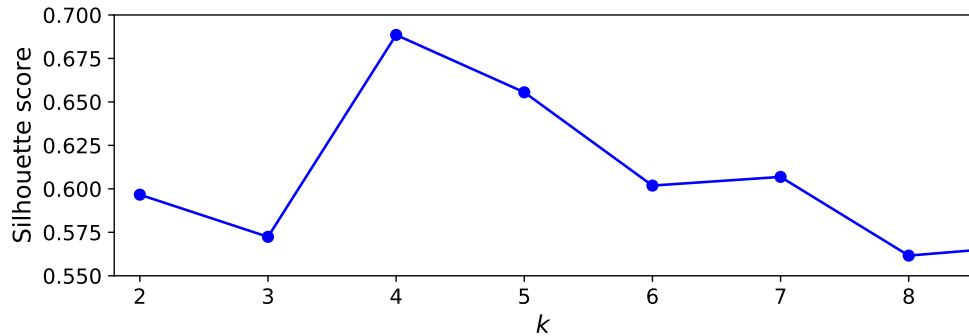


Figure 4.94

An even more informative visualization is obtained when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient. This is called a *silhouette diagram* (see Figure 4.95). Each diagram contains one knife shape per cluster. The shape's height reflects the number of instances the cluster contains and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better). In other words, if we fix a silhouette coefficient on the x axis, the height of the knife shapes in correspondence of that abscissa indicates the number of instances reaching (at least) that silhouette coefficient in every cluster. The dashed line indicates the mean silhouette coefficient. The vertical dashed lines represent the silhouette score for each number of clusters. When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. We can see that when $k = 3$ and when $k = 6$, we get bad clusters. But when $k = 4$ or $k = 5$, the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0. When $k = 4$, the cluster at index 1 (the third from the top) is rather big. When $k = 5$, all clusters have similar sizes. So, even though the overall silhouette score from $k = 4$ is slightly greater than for $k = 5$, it seems like a good idea to use $k = 5$ to get clusters of similar sizes.

Limits of K-means Despite its many merits, most notably being fast and scalable, K-means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover, K-means does not behave very well when the clusters have varying sizes, different densities or nonspherical shapes. For example, Figure 4.96 shows how K-means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities and orientations. As you can see, neither of these solutions is any good. The solution on the left is better, but it still chops off 25 % of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better.

4 Machine learning techniques

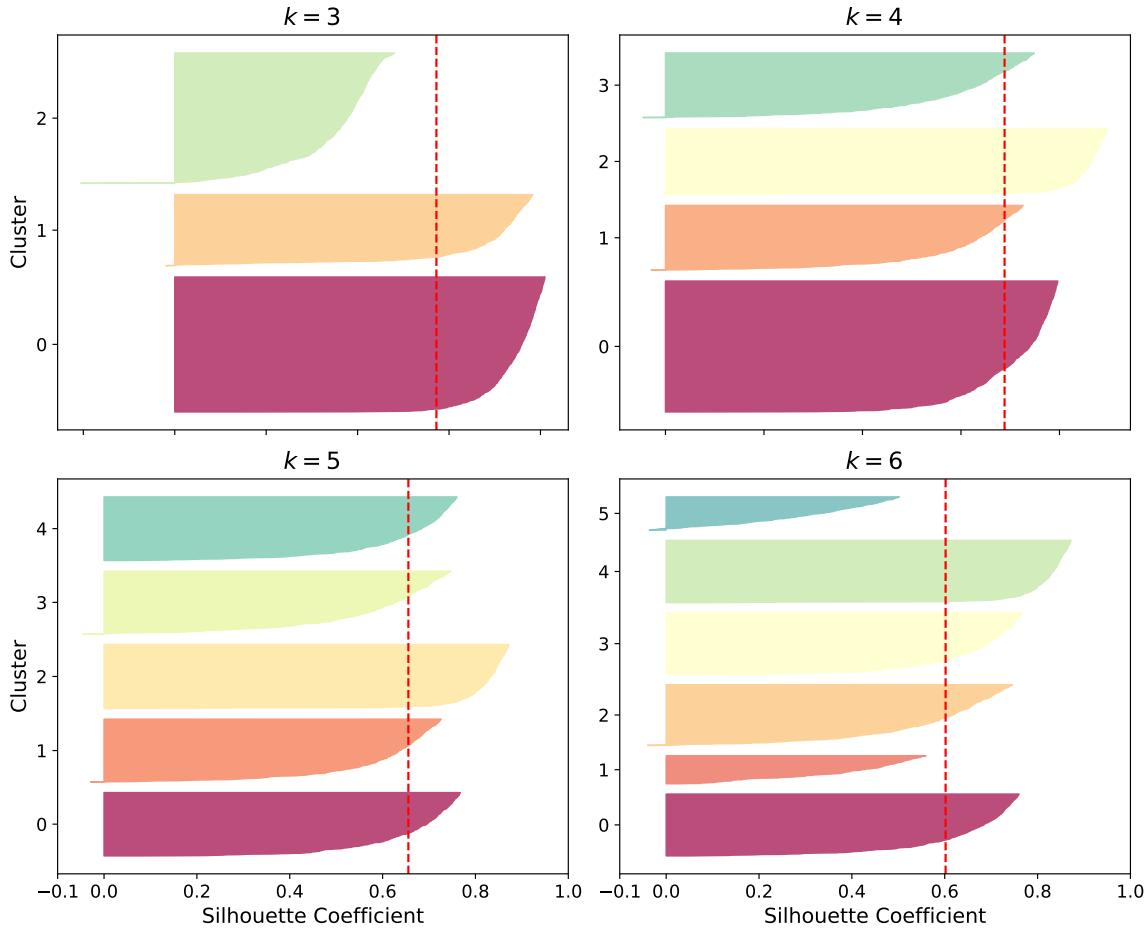


Figure 4.95

On these types of elliptical clusters, Gaussian mixture models work great.

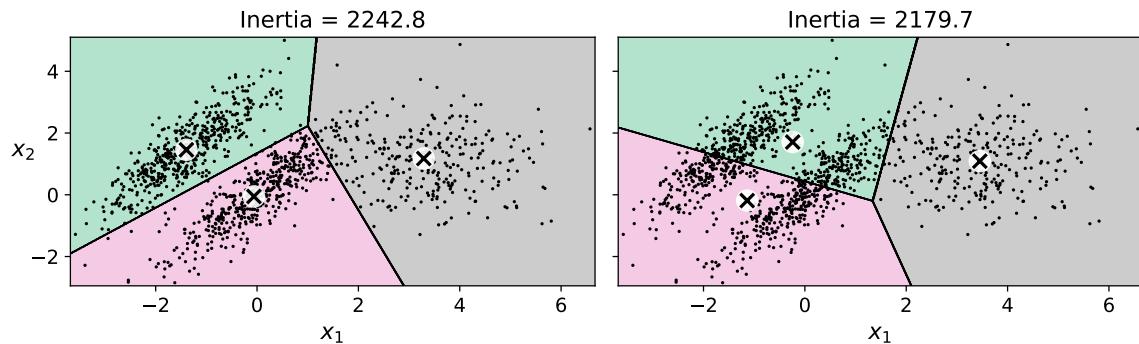


Figure 4.96

At a first glance, it is clear the K-means suffers of the need of a preliminary rescaling of the input features, or the clusters may be very stretched and K-means will perform poorly. In fact, if the order of magnitude of a feature is much higher than the others, then it will dominate over the others when computing the position of the centroids. E.g., if x_1 is much smaller than x_2 , then only x_2 will be determinant in comparing the various distances. However, scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally improves things.

4.8.2 Gaussian mixtures

Before introducing the topic of this section it is better to recall a bit of notation. A one-dimensional random variable X is said to be normally distributed if its probability distribution function is given by

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (4.8.3)$$

where x represents one of the possible outcomes of X . Instead, the *multivariate normal distribution* is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions. One definition is that a random vector variable $\mathbf{X} = (X_1, \dots, X_k)$ is said to be k -variate normally distributed if every linear combination of its k components has a univariate normal distribution. Mathematically speaking, its distribution function is given by the following expression:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^k \det \boldsymbol{\Sigma}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})} \quad (4.8.4)$$

where

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_{X_1} \\ \vdots \\ \mu_{X_k} \end{pmatrix} \equiv \begin{pmatrix} \text{E}(X_1) \\ \vdots \\ \text{E}(X_k) \end{pmatrix} \quad (4.8.5)$$

and the entries of the matrix $\boldsymbol{\Sigma}$, called the covariance matrix, are given by

$$\Sigma_{i,j} = \text{Cov}(X_i, X_j) \quad (4.8.6)$$

The multivariate normal distribution is often used to describe, at least approximately, any set of (possibly) correlated real-valued random variables each of which clusters around a mean value.

A **Gaussian mixture model** (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown:

$$w^{(1)}\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}^{(1)}, \boldsymbol{\Sigma}^{(1)}) + \dots + w^{(k)}\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}^{(k)}, \boldsymbol{\Sigma}^{(k)}) \quad (4.8.7)$$

All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density and orientation, just like in Figure 4.97. When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one and you do not know what the parameters of these distributions are!

In general, there are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class of Scikit-Learn, you must know in advance the number k of Gaussian distributions, i.e. of clusters. The dataset \mathbf{X} is assumed to have been generated through the following probabilistic process:

- For each instance $\mathbf{x}^{(i)}$, a cluster is picked randomly from among k clusters. The probability of choosing the j -th cluster is defined by the cluster's weight, $w^{(j)}$. The index of the cluster chosen for the i -th instance is noted $z^{(i)}$.
- If $z^{(i)} = j$, meaning the i -th instance has been assigned to the j -th cluster, the location $\mathbf{x}^{(i)}$ of this instance is sampled randomly from the Gaussian distribution with mean $\boldsymbol{\mu}^{(j)}$ and covariance matrix $\boldsymbol{\Sigma}^{(j)}$. This is noted $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$.

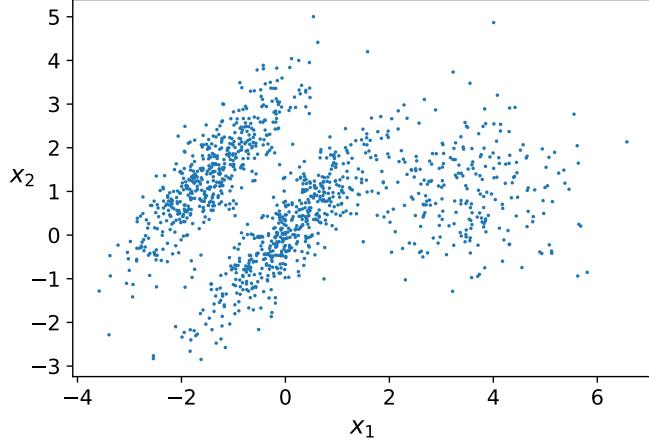


Figure 4.97

In other words, each variable $z^{(i)}$ is drawn from the *categorical distribution*⁹ with weights $w^{(1)}, \dots, w^{(k)}$, while each variable $\mathbf{x}^{(i)}$ is drawn from the normal distribution, with the mean and covariance matrix defined by its cluster $z^{(i)}$.

So, what can you do with such a model? Well, given the dataset \mathbf{X} , you typically want to start by estimating the weights $w^{(1)}, \dots, w^{(k)}$ and all the distribution parameters $\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$. Scikit-Learn's `GaussianMixture` class makes this super easy. Let's look at the parameters that the algorithm estimated (Fig. 4.98).

Great, it worked fine! Indeed, the weights that were used to generate the data in Figure 4.97 were 0.2, 0.4 and 0.4; and similarly, the means and covariance matrices were very close to those found by the algorithm. But how? This class relies on the *expectation-maximization* (EM) algorithm, which has many similarities with the K-means algorithm: it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the *expectation step*) and then updating the clusters (this is called the *maximization step*). Sounds familiar, right? In the context of clustering, you can think of EM as a generalization of K-means that not only finds the cluster centers ($\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$), but also their size, shape and orientation ($\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$), as well as their relative weights ($w^{(1)}$ to $w^{(k)}$). Unlike K-means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters):

$$\pi_j^{(i)} \equiv \pi(\mathbf{x}^{(i)} \in C_j) = \frac{w^{(j)} \mathcal{N}(\mathbf{x}^{(i)}; \boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})}{\sum_{l=1}^k w^{(l)} \mathcal{N}(\mathbf{x}^{(i)}; \boldsymbol{\mu}^{(l)}, \boldsymbol{\Sigma}^{(l)})} \quad (4.8.8)$$

for all $i = 1, \dots, m$ and $j = 1, \dots, k$. Then, during the maximization step, each Gaussian cluster is updated using *all* the instances in the dataset, with each instance weighted by

⁹A categorical distribution (also called a generalized Bernoulli distribution) is a discrete probability distribution that describes the possible results of a random variable that can take on one of k possible categories, with the probability of each category separately specified.

```
In [141]: from sklearn.mixture import GaussianMixture
In [142]: gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
Out[142]: GaussianMixture(n_components=3, n_init=10, random_state=42)

In [143]: gm.weights_
Out[143]: array([0.39054348, 0.2093669 , 0.40008962])

In [144]: gm.means_
Out[144]: array([[ 0.05224874,  0.07631976],
   [ 3.40196611,  1.05838748],
   [-1.40754214,  1.42716873]])

In [145]: gm.covariances_
Out[145]: array([[[[ 0.6890309 ,  0.79717058],
   [ 0.79717058,  1.21367348]],
  [[[ 1.14296668, -0.03114176],
   [-0.03114176,  0.9545003 ]],
  [[[ 0.63496849,  0.7298512 ],
   [ 0.7298512 ,  1.16112807]]]])
```

Figure 4.98

the estimated probability that it belongs to that cluster:

$$w_{\text{new}}^{(j)} = \sum_{i=1}^m \frac{\pi_j^{(i)}}{m} \quad (4.8.9)$$

$$\boldsymbol{\mu}_{\text{new}}^{(j)} = \frac{\sum_{i=1}^m \pi_j^{(i)} \mathbf{x}^{(i)}}{\sum_{l=1}^m \pi_j^{(l)}} \quad (4.8.10)$$

$$\boldsymbol{\Sigma}_{\text{new}}^{(j)} = \frac{\sum_{i=1}^m \pi_j^{(i)} (\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)})^T (\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)})}{\sum_{l=1}^m \pi_j^{(l)}} \quad (4.8.11)$$

for all $j = 1, \dots, k$. These probabilities (the $w^{(i)}$) are called the *responsibilities* of the clusters for the instances. During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.

Unfortunately, just like K-means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to 10. Be careful: by default `n_init` is set to 1.

Now that you have an estimate of the location, size, shape, orientation and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). A Gaussian mixture model is a *generative model*, meaning you can also sample new instances from it.

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method

estimates the logarithm of the probability density function (PDF) at that location. The greater the score, the higher the density. Naturally, if you compute the exponential of these scores, you get the value of the PDF at the location of the given instances. These are not probabilities, but probability *densities*: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within a particular region, you would have to integrate the PDF over that region (if you do so over the entire space of possible instance locations, the result will be 1).

Figure 4.99 shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model.

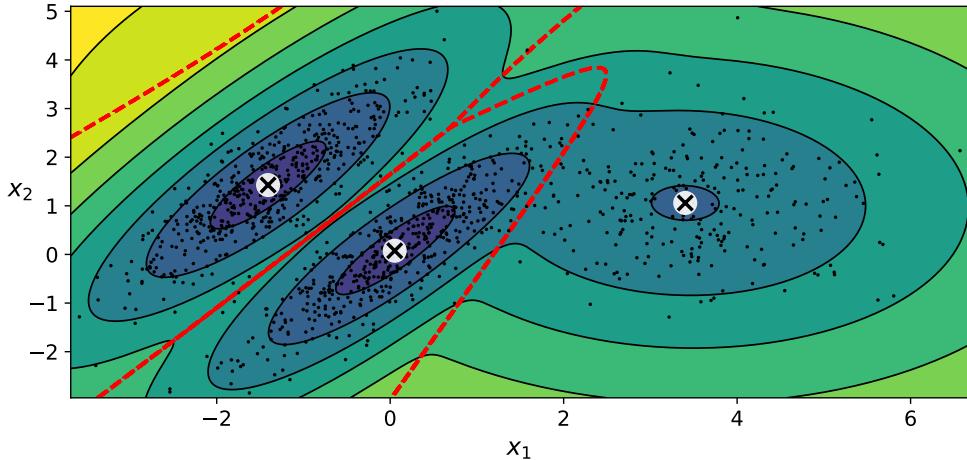


Figure 4.99

4.8.2.1 Anomaly detection

Anomaly detection (also called outlier detection) is the task of detecting instances that deviate strongly from the norm. These instances are called anomalies, or *outliers*, while the normal instances are called *inliers*. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing or removing outliers from a dataset before training another model (which can significantly improve the performance of the resulting model).

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use. If you notice that you get too many false positives (i.e., inliers that are flagged as outliers), you can lower the threshold. Conversely, if you have too many false negatives (i.e., outliers that the system does not flag as anomalous), you can increase the threshold. This is the usual precision/recall trade-off. Figure 4.100 shows how you would identify the outliers of the previous example using the fourth percentile lowest density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies). In this picture these anomalies are represented with stars.

4.8.2.2 Selecting the number of clusters

With K-means, you could use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a theoretical information criterion,

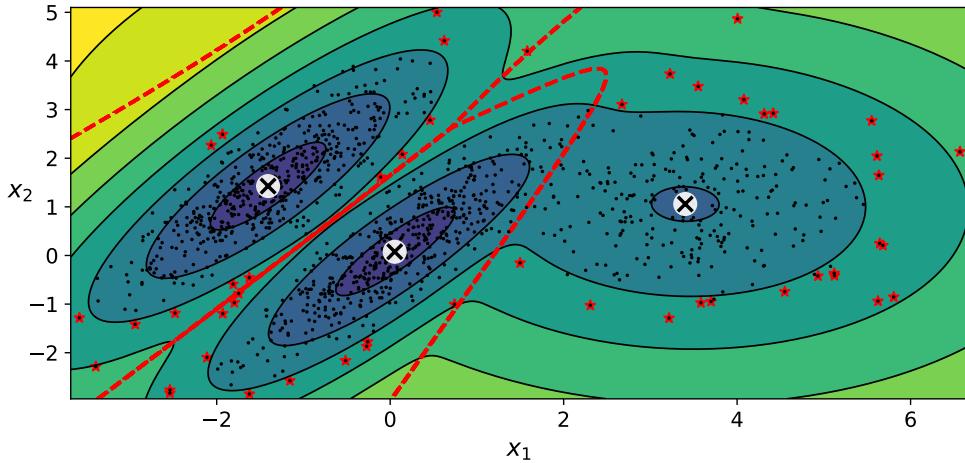


Figure 4.100

such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined as

$$\text{BIC} = p \ln m - 2 \ln \hat{L} \quad (4.8.12)$$

$$\text{AIC} = 2p - 2 \ln \hat{L} \quad (4.8.13)$$

where m is the number of instances, as always, p is the number of parameters learned by the model, and \hat{L} is the maximized value of the *likelihood function* of the model.

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

Likelihood function The terms “probability” and “likelihood” are often used interchangeably in the English language, but they have very different meanings in statistics. Given a statistical model with some parameters θ , the word “probability” is used to describe how plausible a future outcome x is (knowing the parameter values θ), while the word “likelihood” is used to describe how plausible a particular set of parameter values θ are, after the outcome x is known.

Consider a 1D mixture model of two Gaussian distributions centered at -4 and $+1$. For simplicity, this toy model has a single parameter θ that controls the standard deviations of both distributions:

$$f(x; \theta) = w^{(1)} e^{-\frac{(x+4)^2}{2\theta^2}} + w^{(2)} e^{-\frac{(x-1)^2}{2\theta^2}} \quad (4.8.14)$$

The top-left contour plot in Figure 4.101 shows the entire model $f(x; \theta)$ as a function of both x and θ . To estimate the probability distribution of a future outcome x , you need to set the model parameter θ . For example, if you set θ to 1.3 (the horizontal line), you get the probability density function $f(x; \theta = 1.3)$ shown in the lower-left plot. Say you want to estimate the probability that x will fall between -2 and $+2$. You must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). But what if you don’t know θ , and instead if you have observed a single instance $x = 2.5$ (the vertical line in the upper-left plot)? In this case, you get the likelihood function $\mathcal{L}(\theta|x = 2.5) = f(x = 2.5; \theta)$, represented in the upper-right plot.

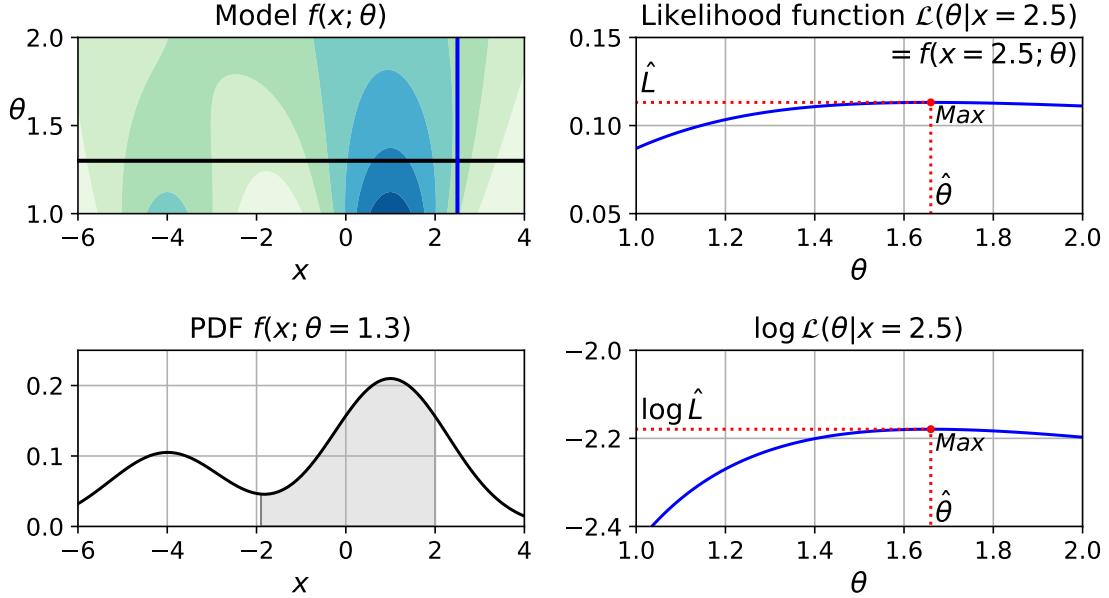


Figure 4.101: A model’s parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right).

In short, the PDF is a function of x (with θ fixed), while the likelihood function is a function of θ (with x fixed). It is important to understand that the likelihood function is not a probability distribution: if you integrate a probability distribution over all possible values of x , you always get 1; but if you integrate the likelihood function over all possible values of θ , the result can be any positive value.

Given a dataset \mathbf{X} , a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given \mathbf{X} . In this example, if you have observed a single instance $x = 2.5$, the *maximum likelihood estimate* (MLE) of θ is $\hat{\theta} = 1.5$. If a prior probability distribution $g(\theta)$ over θ exists, it is possible to take it into account by maximizing $\mathcal{L}(\theta|x)g(\theta)$ rather than just maximizing $\mathcal{L}(\theta|x)$. This is called *maximum a-posteriori* (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

Notice that maximizing the likelihood function is equivalent to maximizing its logarithm (represented in the lower-right-hand plot in Figure 4.101). Indeed the logarithm is a strictly increasing function, so if θ maximizes the log likelihood, it also maximizes the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if you observed several independent instances $x^{(1)}$ to $x^{(m)}$, you would need to find the value of θ that maximizes the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums: $\log(ab) = \log a + \log b$.

Once you have estimated $\hat{\theta}$, the value of θ that maximizes the likelihood function, then you are ready to compute $\hat{L} = \mathcal{L}(\hat{\theta}|x)$, which is the value used to compute the AIC and BIC; you can think of it as a measure of how well the model fits the data.

4.9 Artificial neural networks with Keras

The first part of this section summarizes what we have learned in Chapter 3 about artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to multilayer perceptrons (MLPs), which are heavily used today (other architectures

will be explored in the next sections). In the second part, we will look at how to implement neural networks using the popular Keras application programming interface (API). This is a beautifully designed and simple high-level API for building, training, evaluating and running neural networks. But don't be fooled by its simplicity: it is expressive and flexible enough to let you build a wide variety of neural network architectures. In fact, it will probably be sufficient for most of your use cases. For the code refers to →.

4.9.1 Summary of artificial neural networks

In Chapter 3 we have briefly illustrated the main characteristics of the neurons, the cells of our brain, and we have seen that it seemed quite logical to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANNs): an ANN is a machine learning model inspired by the networks of biological neurons found in our brains. Artificial neural networks 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.

Hebbian learning

In the first simplified model we have studied each *artificial neuron* was in principle linked to all the other neurons simultaneously and had one binary (on/off) output. In order to simulate the finite regenerative period of real neurons, changes in the state of the network are supposed to occur in discrete time steps. At the next time step every artificial neuron activates its output when more than a certain number of its inputs are active at the previous step: the new state of a certain neural unit is determined by the influence of all other neurons, as expressed by a linear combination of their output values:

$$s_i(t+1) = \text{sgn} [h_i(t)] = \text{sgn} \left[\sum_{j=1}^N w_{ij} s_j(t) - \vartheta_i \right] \quad (4.9.1)$$

where the step function accounts for the strong non-linearity of the connections.

In that case training the network meant finding the optimal values of the synapses w_{ij} such that the network evolves from the presented pattern s_i into the most similar stored pattern by virtue of its own inherent dynamics. Our initial attempt was largely inspired by the biological Hebb's rule: when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. This rule is known as *Hebb's rule* (or Hebbian learning):

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \quad (4.9.2)$$

Then we also introduced special learning rules and corrections in order for it to assume the most distinctive characteristics of an associative memory.

Simple perceptron

Despite the important results obtained, these first models were all aimed at storing and recalling given information. Of course, our brains do something else as well, such providing an optimal reaction or answer to external stimuli, or generalizing what we are facing. For this reason we introduced a more complex class of artificial neural networks generally called *perceptrons*. The most trivial structure of a perceptron, called *simple perceptron*, consists of

4 Machine learning techniques

just two layers of neurons closely connected to each other: the *input layer* is composed of the sensory *input neurons* (σ_k), which receive external stimuli, whereas the *output layer* is composed of *output neurons* (S_i), whose goal is to cause the desired reaction. In summary, the neurons S_i of the output layer receive synaptic signals from those of the input layer σ_k , but not vice versa, and the neurons within one layer do not communicate with each other. The flow of information is thus strictly directional; hence one speaks of a *feed-forward network*. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a *fully connected layer* or a *dense layer*. The activation of the output neurons by the input layer is determined by the nonlinear *activation function* $f(h)$ according to

$$S_i^\mu = f(h_i^\mu) = f\left(\sum_{k=1}^{N_{\text{in}}} w_{ik} \sigma_k^\mu - \vartheta_i\right) \quad (4.9.3)$$

for a specific pattern μ , i.e. for a particular combination of input values. The function $f(h)$ may be considered either a stochastic law, where it determines the probability of the values, or a continuous function, if the neurons are assigned analog values (deterministic network). In the previous equation w_{ik} are the synaptic connections from the input k to the output i , while ϑ_i is a threshold (or bias) the output neurons must exceed in order to fire. Be careful that in [9] the nomenclature is slightly different: in that book the first index of the synaptic connections refers to the input, while the second refers to the output (whereas for equation (4.9.3) is reversed); moreover the patterns are called “instances” and each neuron in the input layer represent a different “feature”.

The task now is to choose the synaptic connections w_{ik} in such a way that a certain input σ_k leads to the desired reaction ζ_i , specified by the correct states of the neurons in the output layer. Usually, in order to achieve this, we introduce a notion of distance (loss or cost function) which gives a measure of the error associated to a particular input. In our case, the distance was a sort of mean squared error:

$$D = \frac{1}{2} \sum_{\mu=1}^p \sum_{i=1}^{N_{\text{out}}} (S_i^\mu - \zeta_i^\mu)^2 \quad (4.9.4)$$

evaluated for every pattern simultaneously. At that point, the optimal choice of weights is the one that minimizes this distance. To find the minimum of such a function we have introduced an iterative procedure based on gradient descent. Essentially, we start with a random initialization and at each iteration we move (in the parameter space) in the direction opposite to the gradient. The method works because the gradient always points in the direction of strongest change of the function; hence the negative gradient follows the direction of steepest descent. By exploiting the chain rule of the derivatives, we can the progressive update of the weights and biases as

$$w_{ik}^{(\text{next})} = w_{ik} - \varepsilon \frac{\partial D}{\partial w_{ik}} = w_{ik} + \varepsilon \sum_{\mu=1}^p \Delta_i^\mu \sigma_k^\mu \quad (4.9.5)$$

$$\vartheta_i^{(\text{next})} = \vartheta_i - \varepsilon \frac{\partial D}{\partial \vartheta_i} = \vartheta_i - \varepsilon \sum_{\mu=1}^p \Delta_i^\mu \quad (4.9.6)$$

where

$$\Delta_i^\mu = [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \quad (4.9.7)$$

This is what in [9] is called batch gradient descent because it uses all the instances at the same time (remember Sec. 4.5.2). The parameter ε is the *learning rate* and determines the size of the step towards the minimum: it must be not too large to avoid continuous bouncing from one side to the other of the minimum, and not too small in order to avoid endless times of convergence. Furthermore, the algorithm exploits the derivative of the activation function; hence, if $f(h)$ is a step function (meaning its derivative is not “well” defined in 0), we replace it with a fixed number.

Scikit-Learn provides a `Perceptron` class that implements simple feed-forward neural networks. It can be used pretty much as you would expect—for example, on the Iris dataset introduced in the previous sections (Fig. 4.102). Remember that every instance in the dataset has 4 features (sepal length, sepal width, petal length, petal width) and the class it belongs to (*Iris setosa*, *Iris versicolor*, *Iris virginica*), while here we are neglecting the features concerning the sepal size. Therefore the structure of our simple perceptron is just 2 input neurons and 1 output neuron, providing the prediction for the class. In Figure 4.103 are shown the decision boundaries where the predictions are based on a hard threshold.

```
In [2]: import numpy as np
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)

per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

Figure 4.102

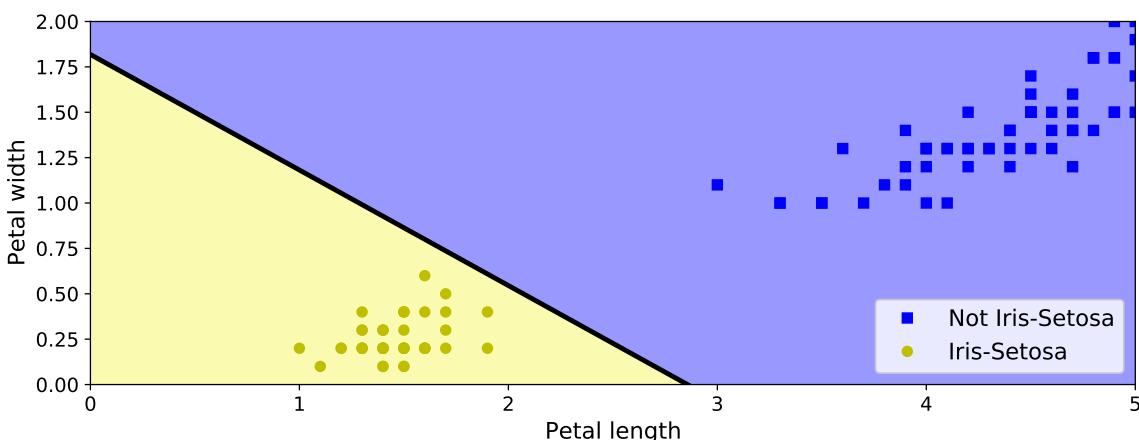


Figure 4.103

Although very powerful, the simple perceptron has a number of serious weaknesses; in particular, the fact that they are incapable of solving some trivial problems, e.g. the exclusive-OR (XOR) classification problem: the XOR function is called linearly inseparable, since there is no way of dividing the space of input variables into regions of equal output by a single linear condition. This is true of any other linear classification model, but researchers had expected much more from perceptrons, and some were so disappointed that

they dropped neural networks altogether in favor of higher-level problems such as logic, problem solving and search.

Multilayer perceptron

However, it turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting artificial neural network is called a *multilayer perceptron* (MLP). A multilayer perceptron has the same basic structure of a simple perceptron, with the only difference that in between the input and output layers there are one or more *hidden layers of hidden neurons*. We have then demonstrated that for a suitable choice of the synapses the multilayer perceptron can solve the XOR problem, as you can verify by computing the output.

In this case the learning procedure is based on a very powerful generalization of the gradient descent, called *error back-propagation*. In just two passes through the network (one forward, one backward), the back-propagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter.

- Each pattern is passed to the network's input layer, which sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*.
- Next, the algorithm measures the network's output error. Then it computes analytically (through the chain rule) how much each output connection contributed to the error. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this *reverse pass* efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm). Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

For example, in a perceptron with two hidden layers the algorithm can be summarized as

$$\bar{s}_k = f(\bar{h}_k), \quad \bar{h}_k = \sum_{l=1}^{N_{\text{in}}} \bar{w}_{kl} \sigma_l - \bar{\vartheta}_k \quad (4.9.8)$$

$$s_j = f(\bar{h}_j), \quad \bar{h}_j = \sum_{k=1}^{N_{\text{hid},1}} \bar{w}_{jk} \bar{s}_k - \bar{\vartheta}_j \quad (4.9.9)$$

$$S_i = f(h_i), \quad h_i = \sum_{j=1}^{N_{\text{hid},2}} w_{ij} s_j - \vartheta_i \quad (4.9.10)$$

and

$$\delta w_{ij} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu s_j^\mu, \quad \delta \vartheta_i = -\varepsilon \sum_{\mu=1}^p \Delta_i^\mu, \quad \Delta_i^\mu = [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \quad (4.9.11)$$

$$\delta \bar{w}_{jk} = \varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu \bar{s}_k^\mu, \quad \delta \bar{\vartheta}_j = -\varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu, \quad \bar{\Delta}_j^\mu = \left(\sum_{i=1}^{N_{\text{out}}} \Delta_i^\mu w_{ij} \right) f'(\bar{h}_j^\mu) \quad (4.9.12)$$

$$\delta \bar{\bar{w}}_{kl} = \varepsilon \sum_{\mu=1}^p \bar{\bar{\Delta}}_k^\mu \sigma_l^\mu, \quad \delta \bar{\bar{\vartheta}}_k = -\varepsilon \sum_{\mu=1}^p \bar{\bar{\Delta}}_k^\mu, \quad \bar{\bar{\Delta}}_k^\mu = \left(\sum_{i=1}^{N_{\text{out}}} \bar{\Delta}_j^\mu \bar{w}_{jk} \right) f'(\bar{\bar{h}}_k^\mu) \quad (4.9.13)$$

It is important to initialize all the hidden layers' connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus back-propagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you break the symmetry and allow back-propagation to train a diverse team of neurons.

When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN). The field of deep learning studies deep neural networks, and more generally models containing deep stacks of computations. Even so, many people talk about deep learning whenever neural networks are involved (even shallow ones).

Note that in the 1980s there was no indication that using multiple hidden layers could in any way increase the performances (recall that we had seen that a function of more than one variable could be reasonably represented with just two hidden layers). Nevertheless, multilayer perceptrons with many layers are even recommended when it is necessary to achieve complicated tasks, because in that way we can "unpack" the total task in group of sub-operations, each performed by a different layer in a hierachic way (first the elementary operations, until they are put together to obtain the most complex ones). To understand why, suppose you are asked to draw a forest using some drawing software. It would take an enormous amount of time: you would have to draw each tree individually, branch by branch, leaf by leaf. If you could instead draw one leaf, copy and paste it to draw a branch, then copy and paste that branch to create a tree, and finally copy and 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 deep neural networks automatically take advantage of this fact: lower hidden layers model low-level structures, intermediate hidden layers combine these low-level structures to model intermediate-level structures, and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures. Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets.

Regression or classification

In order for the back-propagation algorithm to work properly, the choice of the activation function $f(h)$ also plays an important role. In particular, depending on the problem we want to study, one choice may be preferred over another, i.e. we are not restricted to step functions. Typically, step functions works very well when the output must binary (on/off), but if our goal is, for instance, to reproduce a continuous function, it is clear that the output neurons must be able to produce whatever value. Therefore, while the choice of the activation function for the inner layers remains rather arbitrary, various types of

4 Machine learning techniques

non-bounded alternatives have been hypothesized over the years. For example, one who has found a lot of luck is the rectified linear unit (ReLU), given by $x\Theta(x)$, which is indeed limitless for positive values. In practice, it works very well and has the advantage of being fast to compute, so it has become the default. Most importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent. In any case, we will return to the choice of the activation function later, also considering the advantages and disadvantages of each. The most popular activation functions and their derivatives are represented in Figure 4.104.

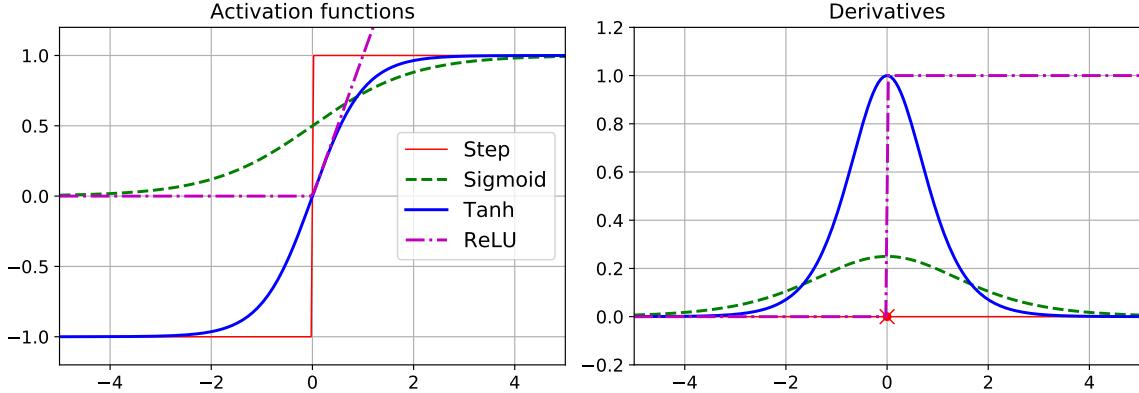


Figure 4.104

First, MLPs can be used for *regression tasks*. If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. Instead, for multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. MLPs can also be used for *classification tasks*. For a binary classification problem, you just need a single output neuron using the logistic activation function (the sigmoid): the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. If each instance can belong only to a single class, out of more than two possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))} \quad (4.9.14)$$

with $s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$ (already introduced in Sec. 4.5.6.4), for the whole output layer. According to the notation of equation (4.9.10), here \mathbf{x} is the vector of the input neurons (s_j) plus the bias feature ($x_0 = 1$), while $\boldsymbol{\theta}^{(i)}$ contains the weights associated the i -th output neuron (w_{ij}) plus the bias ($-\vartheta_i$). The softmax function will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called *multiclass classification*.

The loss function

So far we have never talked too much about the choice of the loss function (or cost function). We have learned that the loss function is a measure of the deviation between the network's output and the function/result it is trying to approximate. The elementary expression of the cost function, $C_{\mathbf{x}}(\boldsymbol{\theta})$, is sample-specific, meaning that it is an indication of the error that would be made in running the neural network for just a single instance \mathbf{x} ($\boldsymbol{\theta}$ represents the ensemble of parameters of the net). However, in order to have a valid network, the perceptron must be trained over a large number of instances (huge datasets); therefore, during training the sample cost function will be actually averaged over all sets of training samples: $C(\boldsymbol{\theta}) = \langle C_{\mathbf{x}}(\boldsymbol{\theta}) \rangle_{\mathbf{x}} = \frac{1}{m} \sum_{i=1}^m C_{\mathbf{x}^{(i)}}(\boldsymbol{\theta})$.

As for the activation function, there is a lot of theory behind cost functions and each one would perform better on a problem with respect to another. Here are shown some of the most famous:

- Mean squared error:

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 \quad (4.9.15)$$

where $\hat{y}^{(i)}$ is the sample-specific outcome and $y^{(i)}$ is the corresponding desired output (i.e. the target).

- Binary cross entropy, sometimes called log loss:

$$\text{BCE} = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \ln \hat{p}^{(i)} + (1 - y^{(i)}) \ln (1 - \hat{p}^{(i)}) \right] \quad (4.9.16)$$

typically used for logistic tasks (binary classifiers). Here $y^{(i)} = 0, 1$ is as usual the expected result, while $\hat{p}^{(i)}$ is the probability to obtain the positive class; accordingly $1 - \hat{p}^{(i)}$ is the probability to obtain the negative class. Notice that, as already said in Sec. 4.5.6.2, this type of loss function estimates high probabilities for positive instances ($y^{(i)} = 1$) and low probabilities for negative instances ($y^{(i)} = 0$).

- Cross entropy:

$$\text{CE} = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \ln (\hat{p}_k^{(i)}) \quad (4.9.17)$$

which is a generalization of the log loss for more classes (softmax).

- Kullback–Leibler divergence:

$$\text{KL}(P\|Q) = \sum_{i=1}^m P(x_i) \log_2 \left(\frac{P(x_i)}{Q(x_i)} \right) \quad (4.9.18)$$

- Hellinger distance:

$$\text{HD} = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^m \left(\sqrt{\hat{y}^{(i)}} - \sqrt{y^{(i)}} \right)^2} \quad (4.9.19)$$

4.9.2 Building an image classifier using Keras

Keras is a high-level deep learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design. To perform the heavy computations required by neural networks, it relies on a computation backend. At present, you can choose from some popular open source deep learning libraries, like TensorFlow, Microsoft Cognitive Toolkit (CNTK), etc. However, as of version 2.4, TensorFlow itself now comes bundled with its own Keras implementation, `tf.keras`. It only supports TensorFlow as the backend, but it has the advantage of offering some very useful extra features: for example, it supports TensorFlow's Data API, which makes it easy to load and preprocess data efficiently. For this reason, we will use `tf.keras` in the following.

All right, it's time to code! As `tf.keras` is bundled with TensorFlow, let's start by installing TensorFlow. At the following links you can find all the information about the installation: <https://www.tensorflow.org/install> and <https://docs.anaconda.com/anaconda/user-guide/tasks/tensorflow/> (the latter for Anaconda). The suggestion is to create a virtual environment for TensorFlow; if you do so, remember that you need to activate it before installing TensorFlow.

Now let's use `tf.keras`! We'll start by building a simple image classifier (supervised learning). First, we need to load a dataset. In this section we will tackle Fashion MNIST, which is a drop-in replacement of MNIST (introduced in Section 4.7). It has the exact same format as MNIST (70 000 grayscale images of 28×28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits (see Fig. 4.105), so each class is more diverse, and the problem turns out to be significantly more challenging than MNIST.



Figure 4.105

4.9.2.1 Using Keras to load the dataset

Keras provides some utility functions to fetch and load common datasets, including MNIST, Fashion MNIST and the California housing dataset we used in Section 4.4. Let's load Fashion MNIST. When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, one important difference is that every image is represented as a 28×28 array rather than a 1D array of size 784. Moreover, the pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0).

Let's take a look at the shape and data type of the training set. Note that the dataset is already split into a training set (60 000) and a test set (10 000), but there is no validation set, so we'll create one now by taking the first 5000 instances of the full training set. Additionally, since we are going to train the neural network using gradient descent, we must scale the input features. For simplicity, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0 (see Fig. 4.106); this also converts them to floats.

```
In [12]: fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

In [13]: X_train_full.shape
Out[13]: (60000, 28, 28)

In [14]: X_train_full.dtype
Out[14]: dtype('uint8')

In [15]: X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

Figure 4.106

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with. For example, the first image in the training set represents a coat (Fig. 4.107).

```
In [17]: y_train
Out[17]: array([4, 0, 7, ..., 3, 0, 5], dtype=uint8)

In [18]: class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
                    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

In [19]: class_names[y_train[0]]
Out[19]: 'Coat'
```

Figure 4.107

4.9.2.2 Creating the model using the sequential API

Now let's build the neural network! The code in Figure 4.108 shows the implementation of a classification multilayer perceptron with two hidden layers. Let's go through this code line by line:

- The first line creates a sequential model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially (the outputs of a layer are the inputs of the next one). This is called the sequential API.

4 Machine learning techniques

- Next, we build the first layer (input) and add it to the model. It is a `Flatten` layer whose role is to convert each input image into a 1D array: if it receives input data `X`, it computes `X.reshape(-1, 1)`. This layer does not have any parameters; it is just there to do some simple preprocessing. Since it is the first layer in the model, you should specify the `input_shape`, which doesn't include the batch size, only the shape of the instances. Alternatively, you could add a `keras.layers.InputLayer` as the first layer, setting `input_shape=[28, 28]`.
- Next we add a `Dense` hidden layer with 300 neurons. It will use the ReLU activation function. Each `Dense` layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron). When it receives some input data, it computes equation (4.9.8).
- Then we add a second `Dense` hidden layer with 100 neurons, also using the ReLU activation function.
- Finally, we add a `Dense` output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

Instead of adding the layers one by one as we just did, you can pass a list of layers when creating the `Sequential` model (`In[25]` of Figure 4.108).

```
In [23]: model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

```
In [25]: model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Figure 4.108

The model's `summary()` method (Fig. 4.109) displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (`None` means the batch size can be anything), and its number of parameters (weight + biases). The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters (we will see examples of non-trainable parameters later on). Note that `Dense` layers often have a lot of parameters. For example, the first hidden layer has 784×300 connection weights, plus 300 bias terms, which adds up to 235 500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods (Fig. 4.110). For a `Dense` layer, this includes both the connection weights and the bias terms. Notice that the `Dense` layer initialized the connection weights randomly (which is needed to break symmetry, as we discussed earlier), and the biases were initialized

```
In [27]: model.summary()

Model: "sequential"
-----
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010

```
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

Figure 4.109

to zeros, which is fine. If you ever want to use a different initialization method, you can set `kernel_initializer` (`kernel` is another name for the matrix of connection weights) or `bias_initializer` when creating the layer.

```
In [31]: weights, biases = hidden1.get_weights()

In [32]: weights
```

```
Out[32]: array([[ 0.02448617, -0.00877795, -0.02189048, ... , -0.02766046,
   0.03859074, -0.06889391],
   [ 0.00476504, -0.03105379, -0.0586676 , ... ,  0.00602964,
   -0.02763776, -0.04165364],
   [-0.06189284, -0.06901957,  0.07102345, ... , -0.04238207,
   0.07121518, -0.07331658],
   ... ,
   [-0.03048757,  0.02155137, -0.05400612, ... , -0.00113463,
   0.00228987,  0.05581069],
   [ 0.07061854, -0.06960931,  0.07038955, ... , -0.00384101,
   0.00034875,  0.02878492],
   [-0.06022581,  0.01577859, -0.02585464, ... , -0.00527829,
   0.00272203, -0.06793761]], dtype=float32)
```

```
In [33]: weights.shape
```

```
Out[33]: (784, 300)
```

Figure 4.110

4.9.2.3 Compiling the model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can specify a list of extra metrics to compute during training and evaluation. The code (Fig. 4.111) requires some explanation. First, we use the "`sparse_categorical_crossentropy`" loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. `[0., 0., 0., 1., 0., 0., 0., 0., 0.]` to represent

4 Machine learning techniques

class 3), then we would need to use the "`categorical_crossentropy`" loss instead. If we were doing binary classification (with one or more binary labels), then we would use the "`sigmoid`" (i.e., logistic) activation function in the output layer instead of the "`softmax`" activation function (generalization of the sigmoid for more classes), and we would use the "`binary_crossentropy`" loss. Regarding the optimizer, "`sgd`" means that we will train the model using simple stochastic gradient descent. In other words, Keras will perform the back-propagation algorithm described earlier (i.e., reverse-model autodiff plus gradient descent). Finally, since this is a classifier, it's useful to measure its "`accuracy`" (percentage of right guessings) during training and evaluation.

```
In [36]: model.compile(loss="sparse_categorical_crossentropy",
                      optimizer="sgd",
                      metrics=["accuracy"])
```

Figure 4.111

4.9.2.4 Training and evaluating the model

Now the model is ready to be trained. For this we simply need to call its `fit()` method. We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional). Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs. If the performance on the training set is much better than on the validation set, your model is probably overfitting the training set (or there is a bug, such as a data mismatch between the training set and the validation set).

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set (Fig. 4.112). You can see that the training loss went down, which is a good sign, and the validation accuracy reached 89.20 % after 30 epochs. That's not too far from the training accuracy, so there does not seem to be much overfitting going on.

```
In [37]: history = model.fit(X_train, y_train, epochs=30,
                           validation_data=(X_valid, y_valid))

Epoch 1/30
1719/1719 [=====] - 2s 1ms/step
- loss: 1.0187 - accuracy: 0.6807 - val_loss: 0.5207 - val_accuracy: 0.8234
Epoch 2/30
1719/1719 [=====] - 2s 921us/step
- loss: 0.5028 - accuracy: 0.8260 - val_loss: 0.4345 - val_accuracy: 0.8538
Epoch 3/30
1719/1719 [=====] - 2s 881us/step
- loss: 0.4485 - accuracy: 0.8423 - val_loss: 0.5334 - val_accuracy: 0.7982
[...]
Epoch 29/30
1719/1719 [=====] - 2s 917us/step
- loss: 0.2284 - accuracy: 0.9177 - val_loss: 0.3053 - val_accuracy: 0.8896
Epoch 30/30
1719/1719 [=====] - 2s 916us/step
- loss: 0.2252 - accuracy: 0.9211 - val_loss: 0.3004 - val_accuracy: 0.8920
```

Figure 4.112

4.10 Training deep neural networks (credit: Francesco Di Clemente)

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). If you use this dictionary to create a pandas DataFrame and call its `plot()` method, you get the learning curves shown in Figure 4.113.

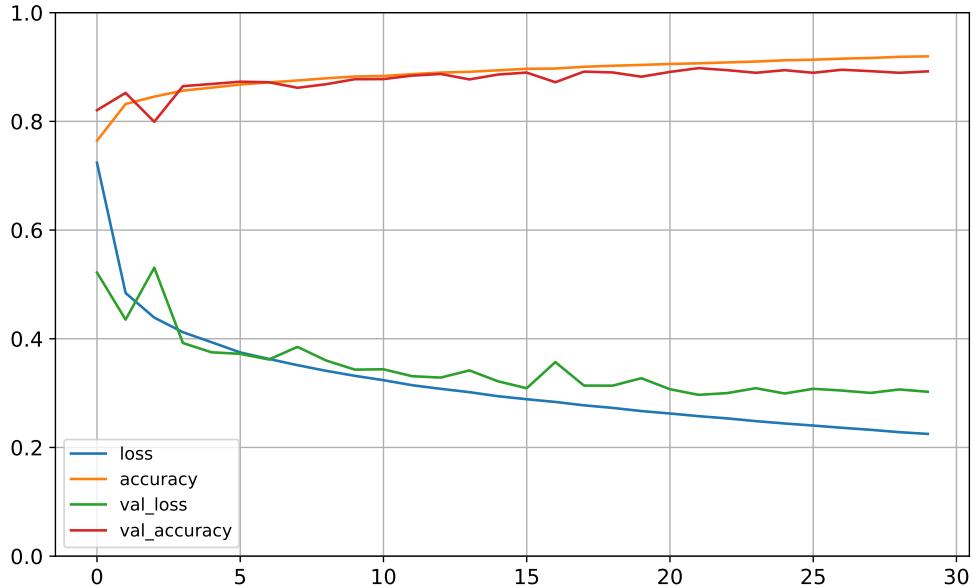


Figure 4.113

You can see that both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease. Good! Moreover, the validation curves are close to the training curves, which means that there is not too much overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training. But that's not the case: indeed, the validation error is computed at the end of each epoch, while the training error is computed using a running mean during each epoch. So the training curve should be shifted by half an epoch to the left. If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training.

The training set performance ends up beating the validation performance, as is generally the case when you train for long enough. You can tell that the model has not quite converged yet, as the validation loss is still going down, so you should probably continue training. It's as simple as calling the `fit()` method again, since Keras just continues training where it left off. Furthermore, if you are not satisfied with the performance of your model, you should go back and tune the hyperparameters (learning rate, optimizer, number of layers, number of neurons per layer, different types of activation functions to use for each hidden layer, etc.).

4.10 Training deep neural networks (credit: Francesco Di Clemente)

In the previous section we summarized artificial neural networks and trained our first deep neural networks with Keras. But they were shallow nets, with just a few hidden layers.

4 Machine learning techniques

What if you need to tackle a 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 10 layers or many more, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep DNN isn't a walk in the park and there are some problems you could run into:

- Training may be extremely slow.
- You may be faced with the tricky vanishing gradients problem or the related exploding gradients problem. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In the following we will go through each of these problems and present techniques to solve them.

4.10.1 Learning rate scheduling

Finding a good learning rate is very important. If you set it much too high, training may diverge (as we discussed in “Gradient descent”). 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 really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution. So the choice of the learning rate depends on the shape of the loss function. The loss function gives us an idea on how the net is actually learning and we have to manually adjust the learning rate to maximize learning (i.e. reducing the loss).

As we discussed many times, you can find a good learning rate by training the model for a few hundred iterations, exponentially increasing the learning rate from a very small value to a very large value, and then looking at the learning curve and picking a learning rate slightly lower than the one at which the learning curve starts shooting back up. You can then reinitialize your model and train it with that learning rate.

But you can do better than a constant learning rate: if you start with a large learning rate and then reduce it once training 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. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called *learning schedules* (we briefly introduced this concept in Sec. 4.5.2).

4.10.2 The vanishing/exploding gradients problems

As we discussed in Section 4.9.1, the back-propagation algorithm works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard 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 layers' connection weights virtually unchanged, and training never converges to a good solution. We call this the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients

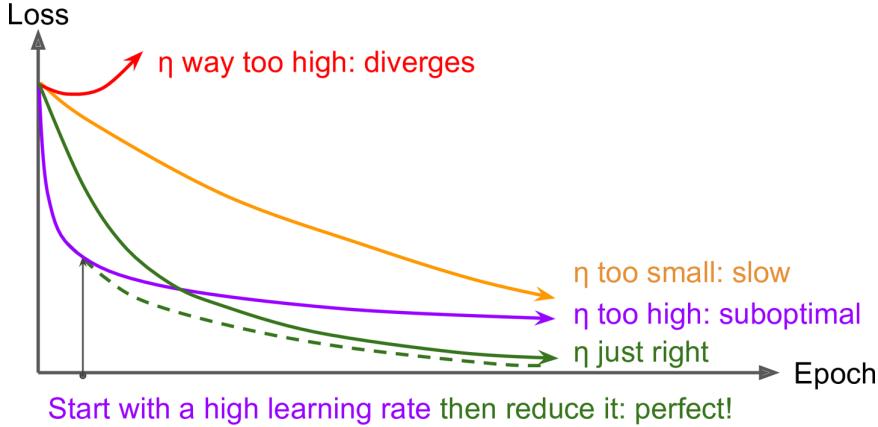


Figure 4.114: Learning curves for various learning rates η .

can grow bigger and bigger until 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.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn't clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a 2010 paper by Xavier Glorot and Yoshua Bengio, which noticed that the problems were in part due to a poor choice of activation function. One of the most popular activation functions is the **sigmoid** (or logistic):

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (4.10.1)$$

which is very useful in classification problem together with BCE. The sigmoid produces a smooth gradient, preventing “jumps” in the output values, which are bound between 0 and 1, thus normalizing the output of each neuron. However, by looking at the function (Fig. 4.115a) 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 back-propagation kicks in it has virtually no gradient to propagate back through the network; and what little gradient exists keeps getting diluted as back-propagation progresses down through the top layers, so there is really nothing left for the lower layers. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction. Moreover it is computationally expensive.

This saturation is actually made worse by the fact that the logistic function has a mean of 0.5, not 0. Instead, the **hyperbolic tangent** function

$$f(x) = \tanh x \quad (4.10.2)$$

has a mean of 0 and behaves slightly better than the logistic function in deep networks. However, a part for this, it suffers of the same disadvantages of the sigmoid activation function.

Therefore around 2010 it turns out that other activation functions behave much better in deep neural networks. In particular, the **ReLU** activation function

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \quad (4.10.3)$$

4 Machine learning techniques

mostly because it does not saturate for positive values (Fig. 4.115c). Moreover, its simplicity makes it computationally efficient, allowing the network to converge very quickly. It has better performance with respect to the previous activation functions and it is widely used in feed-forward neural networks. Unfortunately, also 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 other 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. A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set. When this happens, it just keeps outputting zeros, and gradient descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative. So, due to this, sometimes the network cannot perform back-propagation and cannot learn.

To mitigate this problem of getting stuck in dead states, you may want to use a variant of the ReLU function, such as the **leaky ReLU**. This function is defined as

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x, & \text{if } x < 0 \end{cases} \quad (4.10.4)$$

This variation of ReLU has a small positive slope in the negative area, so it does enable back-propagation, even for negative input values. This small slope (which defines how much the function “leaks”) ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. Unfortunately, even leaky ReLUs tend to produce not consistent results: they do not provide consistent predictions for negative input values.

Another possible choice of the activation function is the **Swish**, or **SiLU**:

$$f(x) = x\sigma(x) = \frac{x}{1 + e^{-x}} \quad (4.10.5)$$

This activation function discovered by researchers at Google. According to their paper, “[...] this simple, suboptimal procedure resulted in Swish consistently outperforming ReLU and other activation functions. [...] The simplicity of Swish and its similarity to ReLU means that replacing ReLUs in any network is just a simple one line code change.” The simplicity of Swish and its similarity to ReLU make it easy for practitioners to replace ReLUs with Swish units in any neural network (especially for hidden layers). Moreover, the simple fact of being non-monotonic means that the function is self-regularized, i.e. it can regularize networks (prevent overfitting). In summary, the SiLU performs better than ReLU with a similar level of computational efficiency: in experiments on image datasets or machine translation it performs $\sim 1\%$ better than ReLU. Swish outperforms ReLU also in batch training, suggesting that the performance difference between the two activation functions remains even when varying the batch size. Being very recently discovered, to date there are no known disadvantages, except for the theoretical fact that ReLU is much better known than SiLU.

We conclude by proposing also the **Mish** activation function:

$$f(x) = x \tanh [\ln (1 + e^x)] \quad (4.10.6)$$

which is a combination of the hyperbolic tangent and the *softplus* activation function $f(x) = \ln (1 + e^x)$ (the softplus is also equal to the primitive of the sigmoid). The Mish function is very similar to Swish but it is found that it performs slightly better.

Other variants that you can find in literature are the RReLU, the ELU, the SELU, etc.

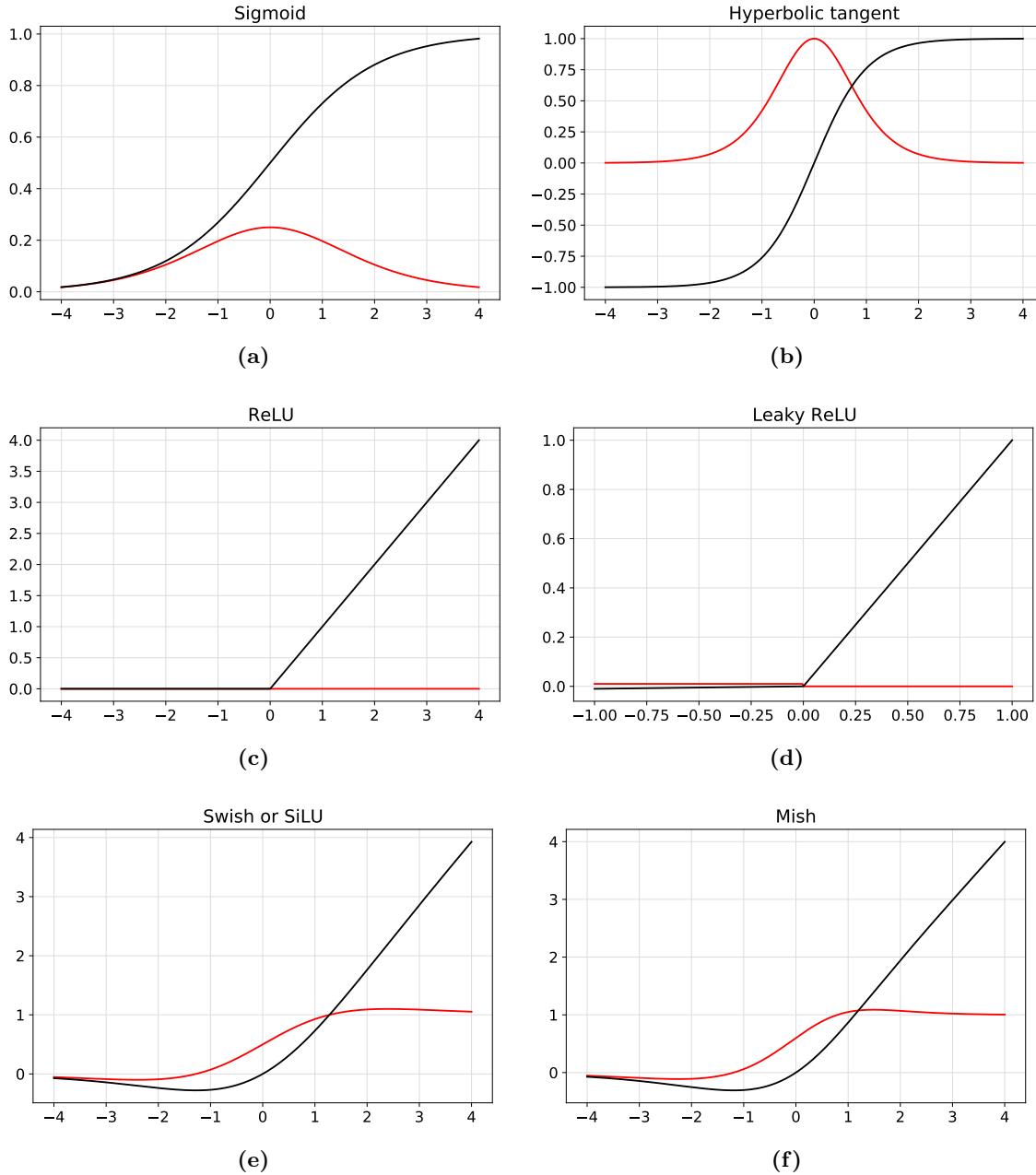


Figure 4.115: Some of the most common activation functions (black) with their derivatives (red).
Code in [→](#).

4.10.3 Faster optimizers

Training a very large deep neural network can be painfully slow. So far we have seen how choosing good activation functions can increase the performance. Another huge speed boost in training DNNs comes from using a faster optimizer than the regular gradient descent optimizer. In this section we will present the most popular algorithms (every year dozens of new algorithms are proposed with the aim of increasing performances).

4.10.3.1 Momentum optimization

Imagine a 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 1964. In contrast, regular gradient descent will simply take small, regular steps down the slope, so the algorithm will take much more time to reach the bottom.

Recall that gradient descent updates the weights θ by directly subtracting the gradient of the cost function $J(\theta)$ with regard to the weights ($\nabla_\theta J(\theta)$) multiplied by the learning rate η . The equation is: $\theta \rightarrow \theta - \eta \nabla_\theta J(\theta)$. 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 subtracts the local gradient from the momentum vector \mathbf{m} (multiplied by the learning rate η), and it updates the weights by adding this momentum vector:

$$\mathbf{m} \rightarrow \beta \mathbf{m} - \eta \nabla_\theta J(\theta) \quad (4.10.7)$$

$$\theta \rightarrow \theta + \mathbf{m} \quad (4.10.8)$$

In other words, the gradient is used for acceleration, not for speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β , called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $\frac{1}{1-\beta}$ (ignoring the sign). 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 plateaux much faster than gradient descent. We have seen in Section 4.5.2 that when the inputs have very different scales, the cost function will look like an elongated bowl (see Figure 4.49). 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 valley faster and faster until it reaches the bottom (the optimum).

For a better understanding of this optimization algorithm, the website <https://distill.pub/2017/momentum/> presents a very nice interactive simulation that you can play with by changing the values of some parameters (learning rate, momentum, starting point).

4.10.3.2 Nesterov accelerated gradient

One small variant to momentum optimization, proposed by Yurij Nesterov in 1983, is almost always faster than vanilla momentum optimization. The **Nesterov accelerated gradient** (NAG) method, also known as Nesterov momentum optimization, measures the gradient of the cost function not at the local position θ but slightly ahead in the direction of the momentum, at $\theta + \beta \mathbf{m}$:

$$\mathbf{m} \rightarrow \beta \mathbf{m} - \eta \nabla_\theta J(\theta + \beta \mathbf{m}) \quad (4.10.9)$$

$$\theta \rightarrow \theta + \mathbf{m} \quad (4.10.10)$$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original

position, as you can see in Figure 4.116 (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 $\theta + \beta m$).

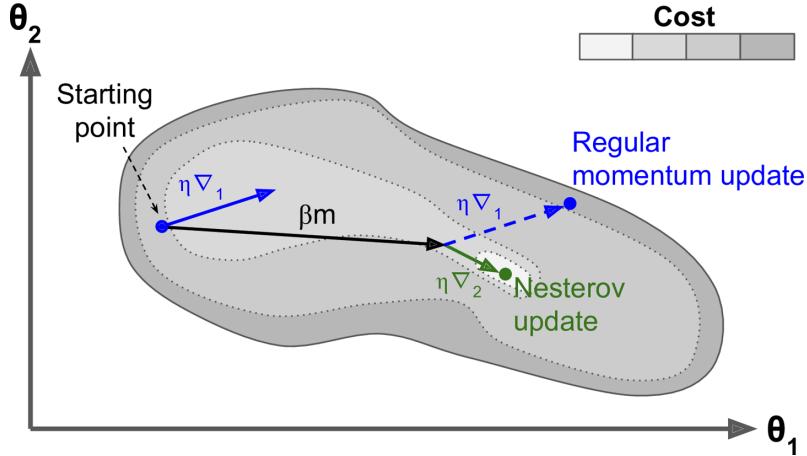


Figure 4.116

4.10.3.3 AdaGrad

Consider the elongated bowl problem again: gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The **AdaGrad** algorithm achieves this correction by scaling down the gradient vector along the steepest dimensions:

$$\mathbf{s} \rightarrow \beta \mathbf{s} + \nabla_{\theta} J(\theta) \odot \nabla_{\theta} J(\theta) \quad (4.10.11)$$

$$\theta \rightarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon} \quad (4.10.12)$$

The first step accumulates the square of the gradients into the vector \mathbf{s} (recall that the \odot symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \rightarrow s_i + (\partial J(\theta)/\partial \theta_i)^2$ for each element s_i of the vector \mathbf{s} ; in other words, each \mathbf{s} accumulates the squares of the partial derivative of the cost function with regard to parameter θ . 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 $\sqrt{\mathbf{s} + \varepsilon}$ (the \oslash symbol represents the element-wise division, and ε is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to simultaneously computing $\theta_i \rightarrow \theta_i - \eta (\partial J(\theta)/\partial \theta_i) / \sqrt{s_i + \varepsilon}$ for all parameters.

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 toward the global optimum. One additional benefit is that it requires much less tuning of the learning rate hyperparameter η .

AdaGrad frequently performs well for simple quadratic problems, but 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

4 Machine learning techniques

Keras has an **Adagrad** optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as linear regression, though).

4.10.3.4 RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The **RMSProp** algorithm fixes this by accumulating only 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:

$$\mathbf{s} \rightarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \odot \nabla_{\theta} J(\theta) \quad (4.10.13)$$

$$\theta \rightarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon} \quad (4.10.14)$$

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.

Figure 4.117: Comparison between different optimization algorithms (click to run the animation).

4.10.3.5 Adam

Adam, 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 keeps track of an

4.10 Training deep neural networks (credit: Francesco Di Clemente)

exponentially decaying average of past squared gradients:

$$\mathbf{m} \rightarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta) \quad (4.10.15)$$

$$\mathbf{s} \rightarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \odot \nabla_{\theta} J(\theta) \quad (4.10.16)$$

$$\hat{\mathbf{m}} \rightarrow \frac{\mathbf{m}}{1 - \beta_1^t} \quad (4.10.17)$$

$$\hat{\mathbf{s}} \rightarrow \frac{\mathbf{s}}{1 - \beta_2^t} \quad (4.10.18)$$

$$\theta \rightarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon} \quad (4.10.19)$$

In this equation, t represents the iteration number (starting at 1). So, the updates (4.10.17) and (4.10.18) are performed by using the values of \mathbf{m} and \mathbf{s} already updated in the first steps, (4.10.15) and (4.10.16), of the same iteration cycle.

If you just look at steps 1, 2 and 5, you will notice Adam's close similarity to 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). Steps 3 and 4 are somewhat of a technical detail: since \mathbf{m} and \mathbf{s} are initialized at 0, they will be biased toward 0 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^{-7} . 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.

4.10.4 Avoiding overfitting through regularization

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. We need regularization.

4.10.4.1 Dropout

Dropout is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper by Geoffrey Hinton in 2012 and further detailed in a 2014 paper by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks get 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 always excluding the output neurons¹⁰) 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 4.118). The hyperparameter p is called the *dropout rate*, and it is typically set between 10 % and 50 %. After training, neurons don't get dropped anymore, i.e. in the test phase we keep all the neurons.

¹⁰In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).

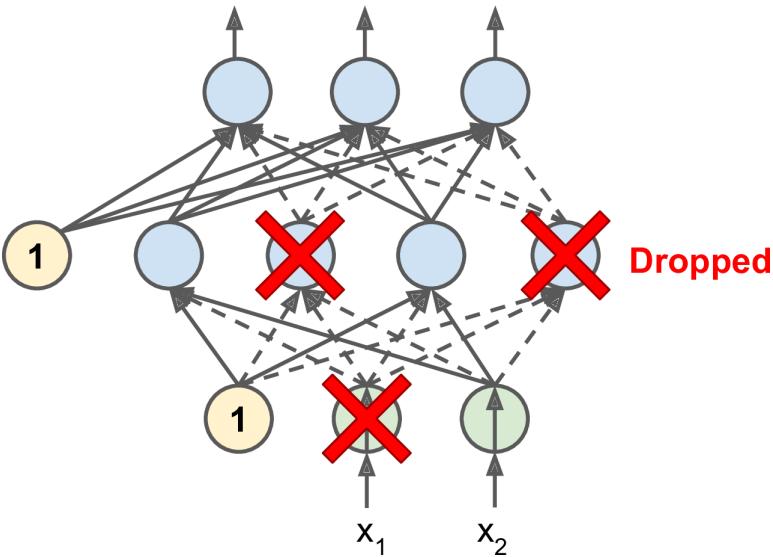


Figure 4.118

It's surprising at first that this destructive technique works at all! Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Once you have run many training steps, you have essentially trained many different neural networks (each with just one training instance). These neural networks are obviously not independent because 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.

It's surprising at first that this destructive technique works at all! There is one small but important technical detail. Suppose $p = 50\%$, in which case during testing a neuron would be connected to twice as many input neurons as it would be (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 will be 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 Keras, you can use the `keras.layers.Dropout` layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all; it just passes the inputs to the next layer.

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate 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 (better in generalizing). So, it is generally well worth the extra time and effort.

4.10.4.2 Monte Carlo dropout

In 2016, a paper by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout. First, the paper established a profound connection between dropout networks

(i.e., neural networks containing a **Dropout** layer before every weight layer) and approximate Bayesian inference, giving dropout a solid mathematical justification. Second, the authors introduced a powerful technique called **MC dropout**, which can boost the performance of any trained dropout model without having to retrain it or even modify it at all, provides a much better measure of the model’s uncertainty, and is also amazingly simple to implement.

Notice that for normal dropout, at test time (i.e. once trained) the prediction is *deterministic*. Without other source of randomness, given one test data point, the model will always predict the same label or value. For Monte Carlo dropout, instead, the dropout is applied at *both* training and test time. Therefore, in the test phase the prediction is no longer deterministic, but depends on which nodes/links you randomly choose to keep. This means that, given a same datapoint, your model could predict different values each time you pass it through the network. So the primary goal of Monte Carlo dropout is to generate *random predictions and interpret them as samples from a probabilistic distribution*. In the authors’ words, they call it *Bayesian interpretation*. Averaging over multiple predictions with dropout on gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout off. And the best part: it does not require any changes in the model’s architecture. We can even use this trick on a model that has already been trained!

4.10.5 Example: Higgs dataset

The most popular deep learning library, after Keras and TensorFlow, is Facebook’s PyTorch library. The good news is that its API is quite similar to Keras’s (in part because both APIs were inspired by Scikit-Learn and Chainer), so once you know Keras, it is not difficult to switch to PyTorch, if you ever want to. To install it, refer to the very clear information at the site <https://pytorch.org/get-started/locally/>. Below is proposed an example of a deep neural network implementation, analyzed just through PyTorch → (credit: F. Di Clemente). The attached .rar archive contains three files: a folder with two .csv datasets of different level of accuracy, and two Jupyter notebooks, one implemented with normal gradient descent and the other with mini-batch gradient descent.

4.11 Convolutional neural networks

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958 and 1959, giving crucial insights into the structure of the visual cortex, the area of the cerebral cortex that processes visual information. In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see Figure 4.119, in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field.

Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in Figure 4.119, notice that each neuron is connected only to a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

These studies of the visual cortex inspired the neocognitron, introduced in 1980, which

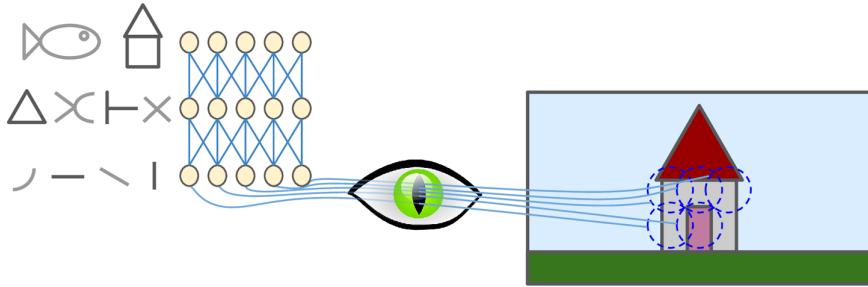


Figure 4.119

gradually evolved into what we now call **convolutional neural networks** (CNNs). This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: convolutional layers and pooling layers. Let's look at them now.

4.11.1 Convolutional layers

The most important building block of a CNN is the *convolutional layer*: neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous sections), but only to pixels in their receptive fields (see Figure 4.120). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

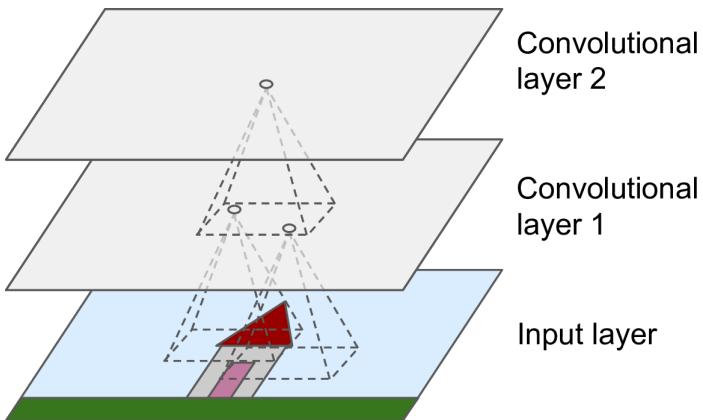


Figure 4.120: CNN layers with rectangular local receptive fields.

All the multilayer neural networks we've looked at so far had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

A neuron located in row i , column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see Figure 4.121). In order for each neuron of a convolutional layer to have the local receptive field with same height

and width of the other of the same layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding* (in other words, it accounts for “missing” neurons in the previous layer).

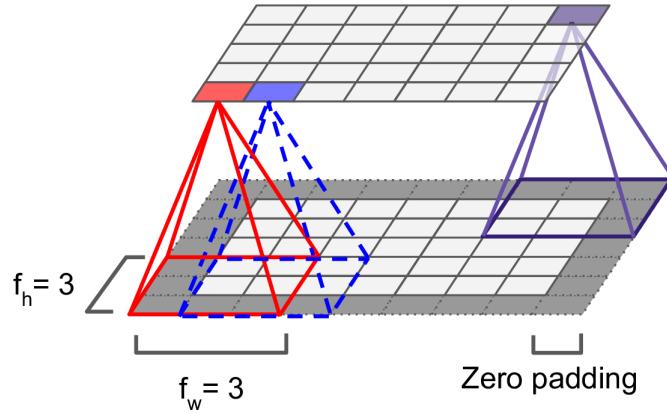


Figure 4.121

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 4.122. This dramatically reduces the model’s computational complexity. The shift from one receptive field to the next is called the *stride*. In the diagram, a 5×7 input layer (plus zero padding) is connected to a 3×4 layer using 3×3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i , column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \cdot s_h$ to $i \cdot s_h + f_h - 1$, columns $j \cdot s_w$ to $j \cdot s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

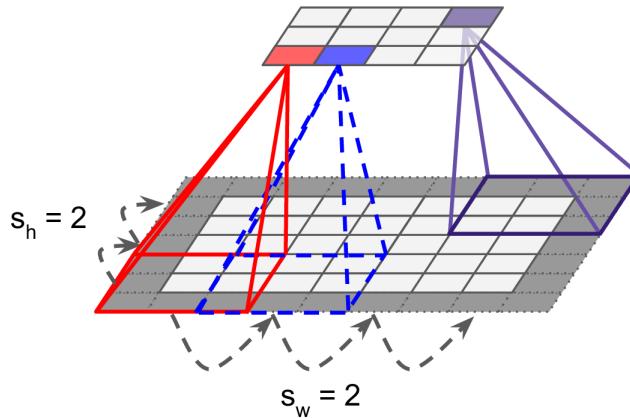


Figure 4.122: Reducing dimensionality using a stride of 2.

4.11.1.1 Filters

A neuron’s weights w_{ij} can be represented as a small image the size of the receptive field. For example, Figure 4.123 shows two possible sets of weights, called *filters*. The first one is represented as a black square with a vertical white line in the middle (it is a 7×7 matrix full of 0s except for the central column, which is full of 1s); neurons in a convolutional layer using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the

4 Machine learning techniques

central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

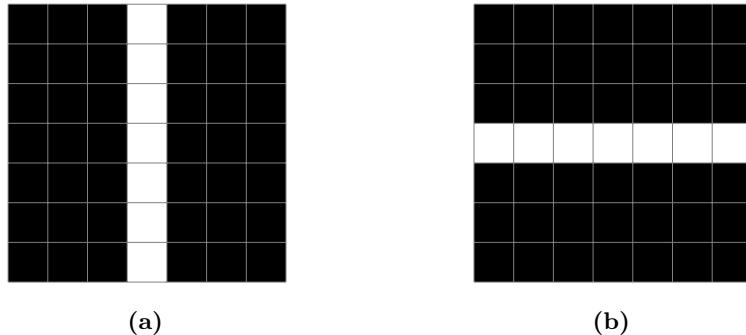


Figure 4.123

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 4.124 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most. Of course, you do not have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

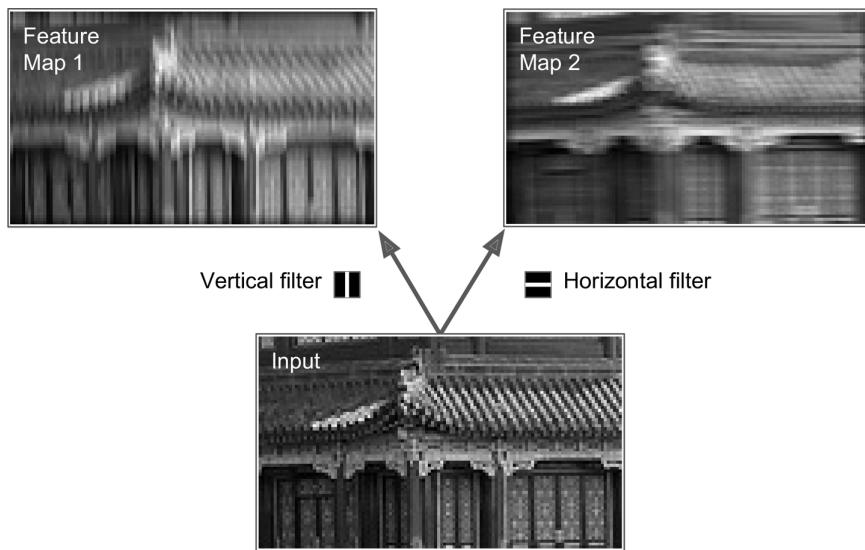


Figure 4.124

4.11.1.2 Stacking multiple feature maps

Up to now, for simplicity, we have represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter, so it is more accurately represented in 3D (see

Figure 4.125). It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same weights and bias term). The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model, because once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. Instead, neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

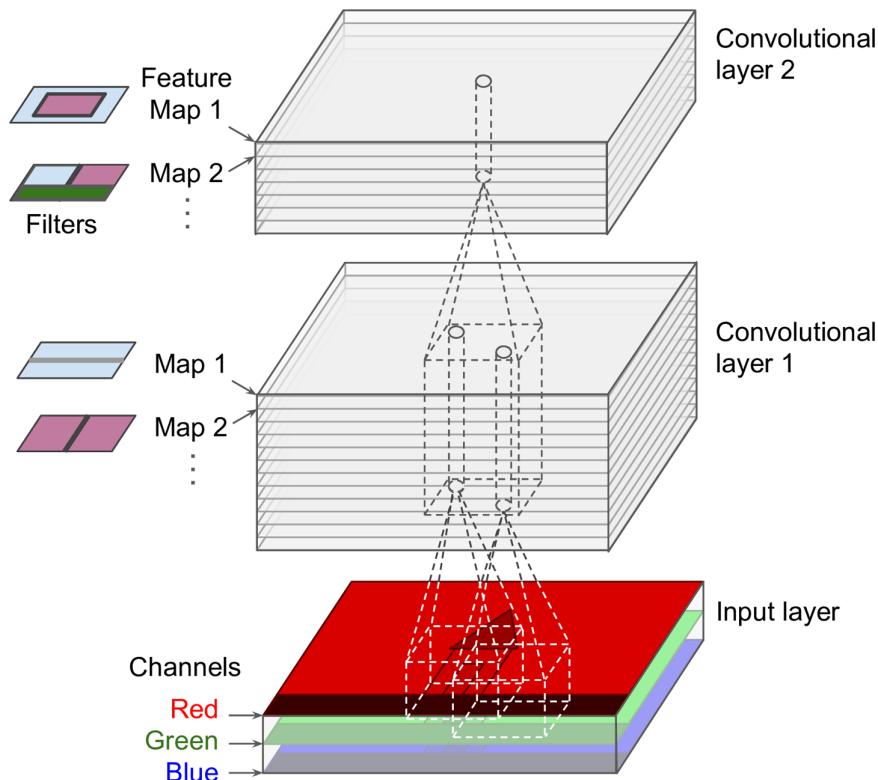


Figure 4.125

Input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just one channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

Specifically, a neuron located in row i , column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \cdot s_h$ to $i \cdot s_h + f_h - 1$ and columns $j \cdot s_w$ to $j \cdot s_w + f_w - 1$, across all feature maps (in layer $l - 1$). Note that all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

All the preceding explanations can be summarized in one big mathematical equation which shows how to compute the output of a given neuron in a convolutional layer:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} w_{u,v,k',k} \quad (4.11.1)$$

with

$$\begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \quad (4.11.2)$$

It is a bit ugly due to all the different indices, but all it does is calculate the weighted sum of all the inputs, plus the bias term. In this equation:

- $z_{i,j,k}$ is the output of the neuron located in row i , column j in feature map k of the convolutional layer (layer l).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k .
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i' , column j' , feature map k' (or channel k' if the previous layer is the input layer).
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u , column v (relative to the neuron's receptive field), and feature map k' .

Notice that both the weights and the biases do not depend on i and j , i.e. on the coordinates of the neuron whose output we are computing.

In summary, it is clear that by introducing convolutional neural networks we have drastically reduced the number of parameters, since every neuron takes the output of only a limited number of neurons of the previous layer. However, this decrease is reflected in the introduction of many more hyperparameters than “standard” perceptrons (e.g., width and length of the receptive fields, stride, filters). Therefore we must be more careful to avoid the risk of overfitting.

4.11.1.3 Memory requirements

Another problem with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of back-propagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100 , with stride 1 and eventual padding. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15\,200$ (the $+1$ corresponds to the bias terms). In fact, all the 150×100 neurons within a specific feature map share the same parameters defined by the filter; accordingly each feature map has just 5×5 different weights for every input layer, in our case 3. Thus the result is 15 200, which is fairly small compared to a fully connected layer. However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that's a total of 225 million float multiplications. Surely not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (12 MB) of RAM. And that's just for one instance—if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

Of course if training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats.

4.11.2 Pooling layers

Once you understand how convolutional layers work, the **pooling layers** are quite easy to grasp. Their goal is to subsample (i.e., shrink) the input image in order to reduce the computational load, the memory usage and the number of parameters (thereby limiting the risk of overfitting).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean.

Figure 4.126 shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2×2 *pooling kernel*, with a stride of 2 and no padding. Only the maximum input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field the input values are 1, 5, 3, 2, so only the maximum value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

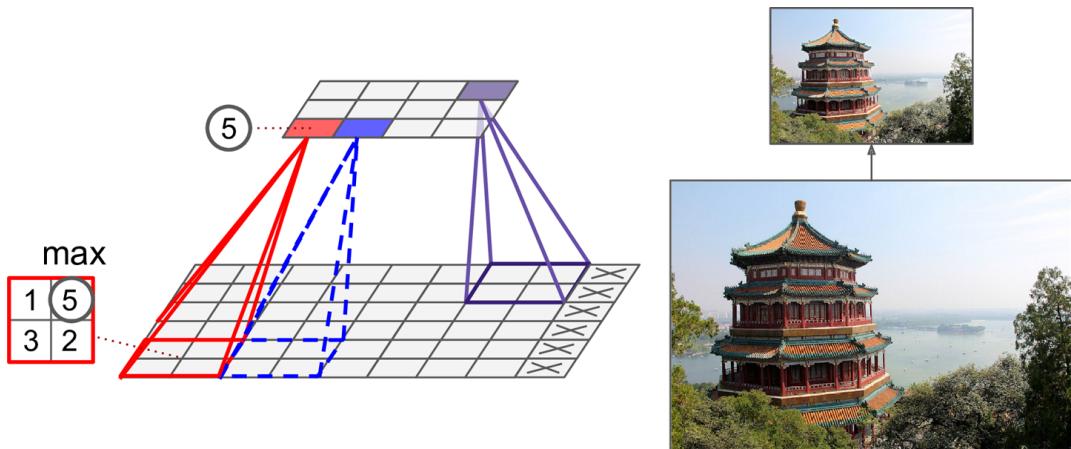


Figure 4.126

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of *invariance to small translations*, as shown in Figure 4.127. Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2×2 kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right, respectively. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the

output is different: it is shifted one pixel to the right (but there is still 75 % invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale. Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

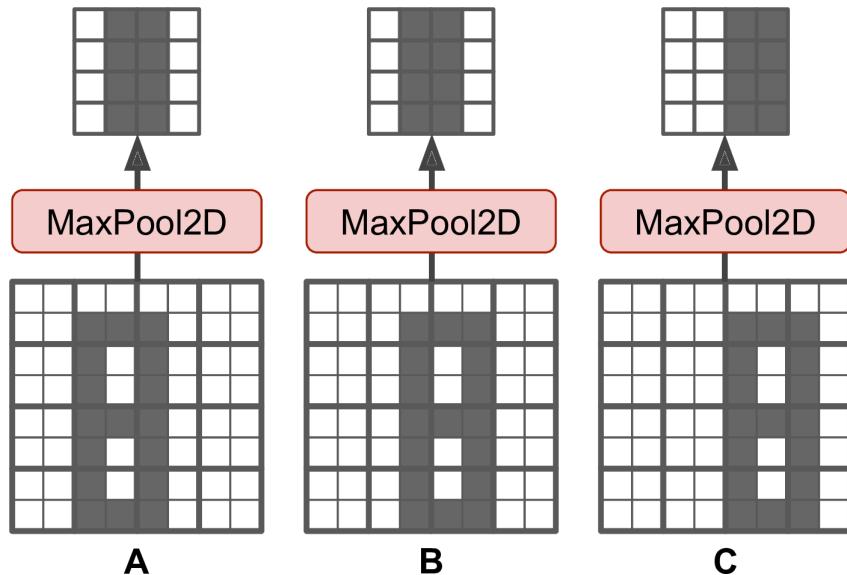


Figure 4.127

However, max pooling has some downsides too. Firstly, it is obviously very destructive: even with a tiny 2×2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75 % of the input values. And in some applications, invariance is not desirable. Take semantic segmentation, the task of classifying each pixel in an image according to the object that pixel belongs to: obviously, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is equivariance, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

4.11.3 Data augmentation

So far we have seen that convolutional networks (but in general all artificial neural networks) need to perform a huge number of operations during training. Furthermore, the essential element to allow an efficient training is to have a dataset as large as possible, so that the machine learns from as many instances as possible. It is obvious, however, that finding a sufficiently large number of examples is not always easy or possible (for example, taking a million photos of flowers to teach the network to distinguish different species is certainly not practical).

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance. This reduces overfitting, making this a regularization technique. The generated instances should be as realistic as possible: ideally, given an image from the augmented training set, a human should not be able to tell whether it was augmented or not. Simply adding white noise will not help; the modifications should be learnable (white noise is not).

For example, 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 4.128). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. For a model that's more tolerant of different lighting conditions, you can similarly generate many images with various contrasts. In general, you can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, you can greatly increase the size of your training set.

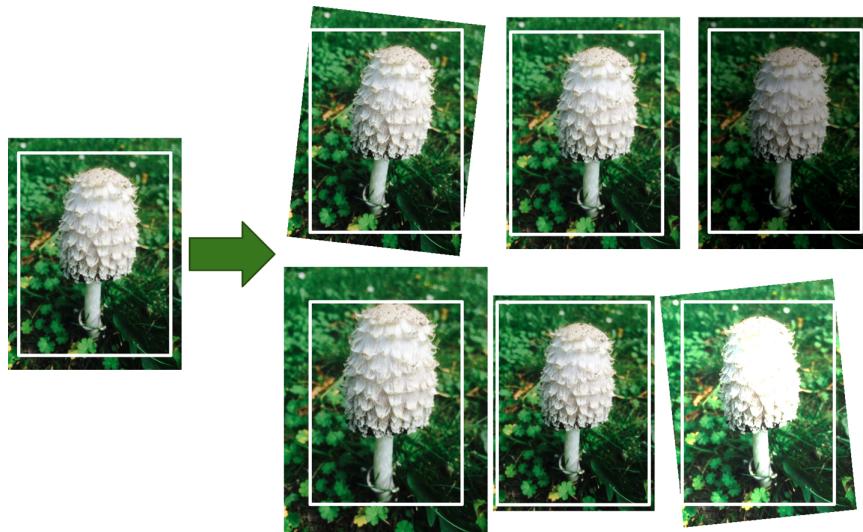


Figure 4.128

CHAPTER 5

Adversarial search

Contents

5.1	Go	239
5.1.1	Basic rules	239
5.1.1.1	Liberties and capture	240
5.1.1.2	Ko rule	240
5.1.1.3	Suicide	241
5.1.1.4	Life and death	242
5.1.1.5	Scoring rules	243
5.1.2	Tactics	243
5.1.3	Ranks and ratings	243
5.2	Game programming	244
5.2.1	History of chess programming	244
5.2.2	Complexity of chess and Go	245
5.2.3	Modeling the three phases of Go	248
5.2.3.1	Programming a Go game	250
5.2.3.2	Coloring the graph	251
5.3	Adversarial search	252
5.3.1	Optimal decision: minimax	254
5.3.1.1	Multiplayer games	255
5.3.2	Alpha–beta pruning	255
5.3.3	Evaluation function	256
5.3.3.1	Horizon effect	258
5.3.4	Monte Carlo tree search	259
5.4	Reinforcement learning	262
5.4.1	Policy search	262
5.4.2	Introduction to OpenAI gym	263
5.4.3	The credit assignment problem	264
5.4.4	Policy gradients	266
5.4.5	Markov decision processes	267
5.5	Alpha Go, AlphaGo Zero & Alpha Zero	269
5.5.1	The algorithm	270
5.5.1.1	The pillars	270
5.5.1.2	The training	272
5.5.1.3	Remarks	274
5.5.2	Performances	276

5.1 Go

Go is a two-player strategic board game that originated in China, where it has been played for at least 2500 years, but is also very popular in East Asia; only in recent years it has spread to the rest of the world as well. It is a strategically very complex game despite its simple rules; a Korean proverb says that “no game of go has ever been played twice”, which is likely if you consider that there are approximately 2.08×10^{170} different possible positions.

The word Go is a short form of the Japanese word *igo* (囲碁), literally “encirclement board game” or “board game of surrounding”. In fact the Go is played by two players who alternately place black and white pawns, called *stones*, on the vacant intersections of a board formed by a 19×19 grid (Fig. 5.1), called *goban* (碁盤). The usual board size is a 19×19 grid but for beginners, or for playing quick games, the smaller board sizes of 13×13 and 9×9 are also popular. The aim of the game is to control an area of the game greater than that controlled by the opponent; for this purpose the players try to arrange their stones so that they cannot be captured, at the same time carving out territories that the opponent cannot invade without being captured. See https://en.wikipedia.org/wiki/Rules_of_Go for the complete list of rules.



(a) The Go board (generally referred to by its Japanese name *goban*, 碁盤).

(b) The first 150 moves of a Go game animated.

Figure 5.1

5.1.1 Basic rules

The two players, Black and White, take turns placing stones of their color on the intersections of the board, one stone at a time. The board is empty to begin with. Black plays first, unless Black is given a handicap of two stones or more (in which case, White plays first). The players may choose any unoccupied intersection to play on, except for those forbidden by the ko and suicide rules (see below). Once played, a stone can never be moved and can be taken off the board only if it is captured. A player may also pass, declining to place a stone, though this is usually only done at the end of the game when both players believe nothing more can be accomplished with further play. When both players pass consecutively,

5 Adversarial search

the game ends. The stones that are still on the board but unable to avoid capture, called *dead stones*, are removed, and is then scored.

5.1.1.1 Liberties and capture

Vertically and horizontally adjacent stones of the same color form a *chain* (also called a *string* or *group*), forming a discrete unit that cannot then be divided. Only stones connected to one another by the lines on the board create a chain; stones that are diagonally adjacent are not connected (Fig. 5.2). Chains may be expanded by placing additional stones on adjacent intersections, and can be connected together by placing a stone on an intersection that is adjacent to two or more chains of the same color.

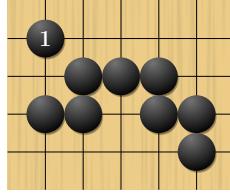


Figure 5.2: *Strings*. The eight black stones (excluding the stone at 1) form a string.

A vacant point adjacent to a stone, along one of the grid lines of the board, is called a *liberty* for that stone. Stones in a chain share their liberties (Figs. 5.3 and 5.4). A chain of stones must have at least one liberty to remain on the board. When a chain is surrounded by opposing stones so that it has no liberties, it is *captured* and removed from the board (Fig. 5.5 and 5.6).

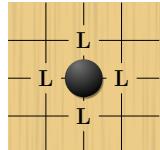


Figure 5.3: *Stone liberties*. The points marked “L” are liberties of the black stone.

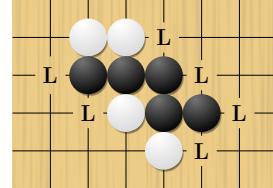


Figure 5.4: *String liberties*. The points marked “L” are liberties of the black string.

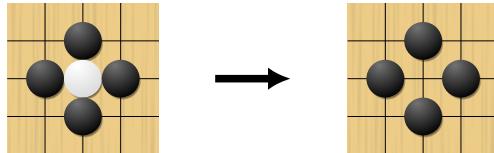


Figure 5.5: *Stone capture*. The white stone has no liberties and is thus captured and removed from the board.

5.1.1.2 Ko rule

Players are not allowed to make a move that returns the game to the previous position. This rule, called the *ko rule*, prevents unending repetition. As shown in the example pictured (Fig. 5.7): Black has just played the stone marked 1, capturing a white stone at the intersection marked with the red circle. If White were allowed to play on the marked intersection, that move would capture the black stone marked 1 and recreate the situation

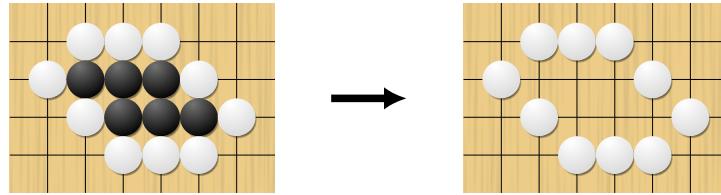


Figure 5.6: *String capture.* The black string has no liberties and is thus captured and removed from the board.

before Black made the move marked 1. Allowing this could result in an unending cycle of captures by both players. The ko rule therefore prohibits White from playing at the marked intersection immediately. Instead White must play elsewhere, or pass; Black can then end the ko by filling at the marked intersection, creating a five-stone black chain. If White wants to continue the ko (that specific repeating position), White tries to find a play elsewhere on the board that Black must answer; if Black answers, then White can retake the ko. A repetition of such exchanges is called a *ko fight*.

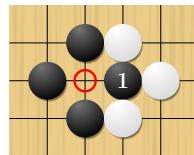


Figure 5.7: An example of a situation in which the ko rule applies.

While the various rule-sets agree on the ko rule prohibiting returning the board to an immediately previous position, they deal in different ways with the relatively uncommon situation in which a player might recreate a past position that is further removed.

5.1.1.3 Suicide

One of the fundamental rules of Go is that every stone remaining on the board must have at least one open point (a liberty) directly orthogonally adjacent (up, down, left, or right), or must be part of a connected group that has at least one such open point next to it. From this general statement, it immediately follows that a player cannot *suicide*, i.e. it may not place a stone such that it or its group immediately has no liberties (Fig. 5.8), unless doing so immediately deprives an enemy group of its final liberty. In the latter case, the enemy group is captured, leaving the new stone with at least one liberty (e.g. in Fig 5.11). This rule is responsible for the all-important difference between one and two eyes: if a group with only one eye is fully surrounded on the outside, it can be killed with a stone placed in its single eye.

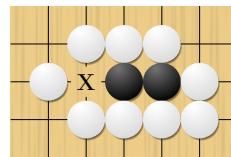


Figure 5.8: *Suicide.* Black can't play at "X" since the whole black string would be captured.

5.1.1.4 Life and death

When a group of stones is mostly surrounded and has no options to connect with friendly stones elsewhere, the status of the group is either *alive*, *dead* or *unsettled*. A group of stones is said to be alive if it cannot be captured, even if the opponent is allowed to move first. Conversely, a group of stones is said to be dead if it cannot avoid capture, even if the owner of the group is allowed the first move. Otherwise, the group is said to be unsettled: the defending player can make it alive or the opponent can *kill* it, depending on who gets to play first.

An *eye* is an empty point or group of points surrounded by one player's stones (Fig. 5.9). If the eye is surrounded by White stones, Black cannot play there unless such a play would take White's last liberty and capture the White stones (Fig. 5.11); such a move is forbidden according to the suicide rule in most rule sets, but even if not forbidden, such a move would be a useless suicide of a Black stone.

If a white group has two eyes (Fig. 5.10), Black can never capture it because Black cannot remove both liberties simultaneously. If White has only one eye, Black can capture the white group by playing in the single eye, removing White's last liberty. Such a move is not suicide because the white stones are removed first.

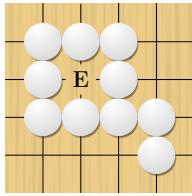


Figure 5.9: *Eye*. The point marked “E” is an eye.

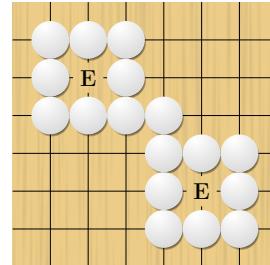


Figure 5.10: *Two eyes*. The points marked “E” are eyes. The string cannot be captured.

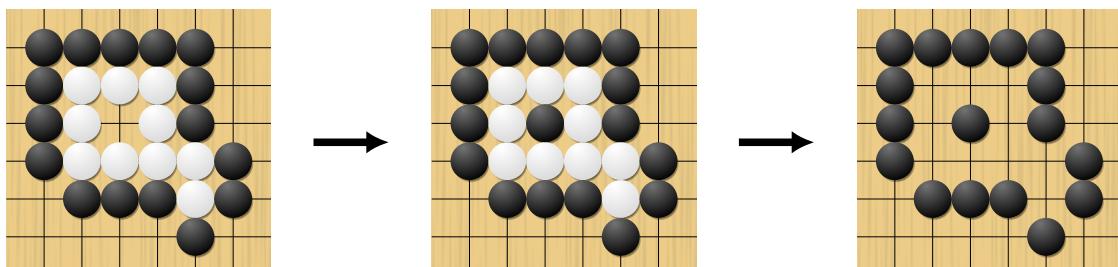


Figure 5.11: *Capturing a string with one eye*. The string is first surrounded and then the eye is filled. The white stones are removed because they have no liberties, creating liberties for the black stone which filled the eye. Thus, playing in the eye does not violate the suicide rule.

In the example of Figure 5.12, all the circled points are eyes. The two white groups in the upper corners are alive, as both have at least two eyes. The groups in the lower corners are dead, as both have only one eye. The group in the lower left may seem to have two eyes, but the surrounded empty point marked “a” is not actually an eye. In other words, in principle Black would be denied the move in “a” because the stone it would put wouldn't have liberties, but in doing so it deprives the white stone (those above it) of its liberty, hence capturing it. Therefore, Black can play there and take a white stone. Such a point is often called a *false eye*.

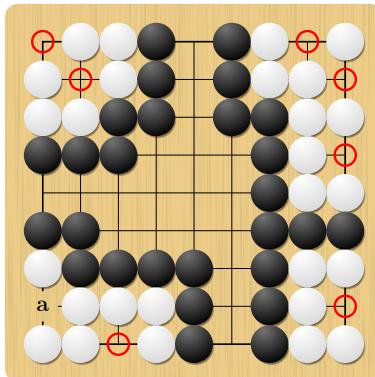


Figure 5.12: Examples of eyes (marked). The white groups at the top of the board are alive, as they have at least two eyes. The white groups at the bottom are dead as they only have one eye. The point marked “a” is a false eye.

5.1.1.5 Scoring rules

Two general types of scoring system are used, and players determine which to use before play. Both systems almost always give the same result.

- *Area scoring* (including Chinese): A player’s score is the number of stones that the player has on the board, plus the number of empty intersections surrounded by that player’s stones.
- *Territory scoring* (including Japanese and Korean): In the course of the game, each player retains the stones they capture, termed *prisoners*. Any dead stones removed at the end of the game become prisoners. The score is the number of empty points enclosed by a player’s stones, plus the number of prisoners captured by that player.

5.1.2 Tactics

There are several tactical constructs aimed at capturing stones. These are among the first things a player learns after understanding the rules. Recognizing the possibility that stones can be captured using these techniques is an important step forward.

The most basic technique is the *ladder*. To capture stones in a ladder, a player uses a constant series of capture threats (*atari*, 当たり) to force the opponent into a zigzag pattern as shown in Figure 5.13. Unless the pattern runs into friendly stones along the way, the stones in the ladder cannot avoid capture. Experienced players recognize the futility of continuing the pattern and play elsewhere.

5.1.3 Ranks and ratings

In Go, the rank indicates a player’s skill in the game. Traditionally, ranks are measured using *kyu* and *dan* grades, a system also adopted by many martial arts. More recently, mathematical rating systems similar to the Elo rating system have been introduced (see below). Such rating systems often provide a mechanism for converting a rating to a kyu or dan grade. Kyu grades (abbreviated “k”) are considered student grades and decrease as playing level increases, meaning 30th kyu is the lowest available kyu grade, while 1st kyu is the strongest. Dan grades (abbreviated “d”) are considered master grades, and increase from 1st dan to 7th dan. First dan equals a black belt in eastern martial arts using this system. The difference among each amateur rank is one handicap stone. For example, if a

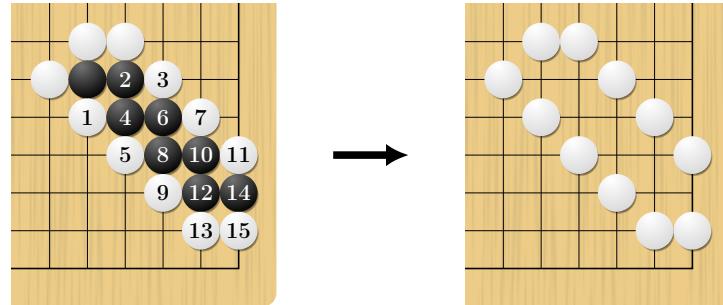


Figure 5.13: Ladder. A ladder results from Black trying to save a dead stone. White wins the ladder when the edge of the board is encountered. The black stones are then removed from the board.

5k plays a game with a 1k, the 5k would need a handicap of four stones to even the odds. Top-level amateur players sometimes defeat professionals in tournament play. Professional players have professional dan ranks (abbreviated “p”), from 1 to 9. These ranks are separate from amateur ranks.

5.2 Game programming

5.2.1 History of chess programming

The idea of studying algorithms and artificial codes for playing games is very old. The reason for this interest on a global scale is first due to the fact that there is a close interplay between artificial intelligence and the real functioning of the brain: building a simpler artificial model emulating the behavior of the brain on a smaller scale allows a better understanding of real aspects that would otherwise be difficult to study (great interdisciplinarity). Secondly, there is also the “competitive” aspect of the matter, that is, trying to understand how to best approach with someone competing against you. Traditionally, chess, Go, backgammon, etc. are considered limited but difficult problems (of course with their differences). Therefore they constitute very good tests for the development of artificial intelligence techniques and machine learning.

The problem of “building a machine able to play well at chess” was formulated for the first time as a bet in 1769 by the baron von Kempelen, technician of the Empress Maria Theresa of Austria. Thus, the baron invented a chess-playing machine, called the Turk, which allowed him to win most of the games played during its demonstrations around Europe and the Americas for nearly 84 years, playing and defeating many challengers, including statesmen. However, despite its success, the machine revealed to be an elaborate hoax; in fact, the Turk was in reality a mechanical illusion that allowed a skilled human chess master hiding inside to operate the machine. Lastly, the machine ended burned in a fire at Philadelphia, thus leaving to posterity nothing but a picturesque story.

The real history of machines playing difficult games like chess began only after World War II, which certainly gave a tremendous boost to computer research. In fact, from those years on scientists started using computers systematically in their works. The main places where these technologies first developed were United States and Russia (at the time Soviet Union), also thanks to a heavy research on game theory and artificial players financed by their powerful military apparatuses. In particular, the Soviet chess team was led by the famous world champion M. Botvinnik, which played a major role in the organization of chess, becoming a leading member of the coaching system that enabled the Soviet Union to

dominate top-class chess during that time. Furthermore, he was an engineer and computer scientist; so he also helped the development of computer programs as a technician.

Despite the powerful impetus provided by the Soviet Union, the very first working program was created at Los Alamos (New Mexico, USA) in 1956¹, but with a 6×6 chessboard and without bishops. In fact, at the very beginning seemed very difficult to play chess on a computer, and hence they used simplified versions of the game by reducing the dimensions of the board or by eliminating some pieces.

A further step in the process of perfecting computers took place in 1967, when for the first time a machine, the MIT's Mac Hack, participated at a human tournament. Even though the machine wasn't actually up to par with the other players, achieving very poor results, it had the significant advantage (with respect to its predecessors) of being relatively small, and therefore of being able to be transported with some ease.

However, starting from the 80s, the situation began to change drastically in favor of the machines. In 1983 the computer Belle (Bell Labs) was the first machine to achieve Master-level play, while in 1988 Deep Thought defeated the Grandmaster B. Larsen. In the following years some commercial programs (Fritz3, Genius) also defeat the World Champion G. Kasparov in brief matches.

But the final turning point came in 1997 when Deep Blue (IBM) managed to win, although marginally, against the world champion in a tournament-match with long games. After the defeat, Kasparov said that he sometimes saw deep intelligence and creativity in the machine's moves, suggesting that during the second game, human chess players had intervened on behalf of the machine, which would be a violation of the rules. IBM denied that it cheated, arguing that the only human intervention occurred between games. Kasparov demanded a rematch, but IBM had dismantled Deep Blue after its victory and refused the rematch. Kasparov also requested printouts of the machine's log files, but IBM refused once again, although the company later published the logs on the Internet. Anyway, from that moment it was clear that computers had surpassed humans in terms of performances (at least for chess). Computer scientists believed that playing chess was a good measurement for the effectiveness of artificial intelligence, and by beating a world champion chess player, IBM showed that they had made significant progresses.

5.2.2 Complexity of chess and Go

The Elo² rating system is a method for calculating the relative skill levels of players in zero-sum games such as chess. Without going too far on how it actually works, the difference in the Elo ratings between two players serves as a predictor of the outcome of a match. Two players with equal ratings who play against each other are expected to score an equal number of wins. A player whose rating is 100 points greater than their opponent's is expected to score 64%; if the difference is 200 points, then the expected score for the strongest player is 76%. A player's Elo rating is represented by a number which may change depending on the outcome of rated games played. After every game, the winning player takes points from the losing one. The difference between the ratings of the winner and loser determines the total number of points gained or lost after a game. The Elo thesis is: *If a player performs as expected, it gains nothing. If it performs better than expected, it is rewarded, while if it performs poorer than expected, it is penalized.*

Typically, with a rating between 0 and 2000 you are considered a beginner/hobbist, with different levels of ability, while scores above 2000 are reserved for Masters, Grandmasters

¹The Fermi–Pasta–Ulam (FPU) problem on the theory of chaotic behavior marked the beginning of both a new field, nonlinear physics, and the age of computer simulations on scientific problems.

²After its creator Arpad Elo, a Hungarian-American physics professor.

5 Adversarial search

and World Champions. The highest score ever registered was 2882, attributable to the Norwegian champion M. Carlsen currently in charge since 2013.

In Figure 5.14 is shown the estimated Elo ratings for various chess computer programs over the past years. As we can see, year by year there has been a systematic growth in computer skills, even surpassing man. This is somehow frustrating from the point of view of humans, since in chess there is a direct relationship that has made “trivial” the development of ever stronger programs (Tab. 5.1). The dominant component was the velocity of the machine: for instance, Deep Blue was able to record about 1 million positions per second, while a human player only 1 (as order of magnitude).

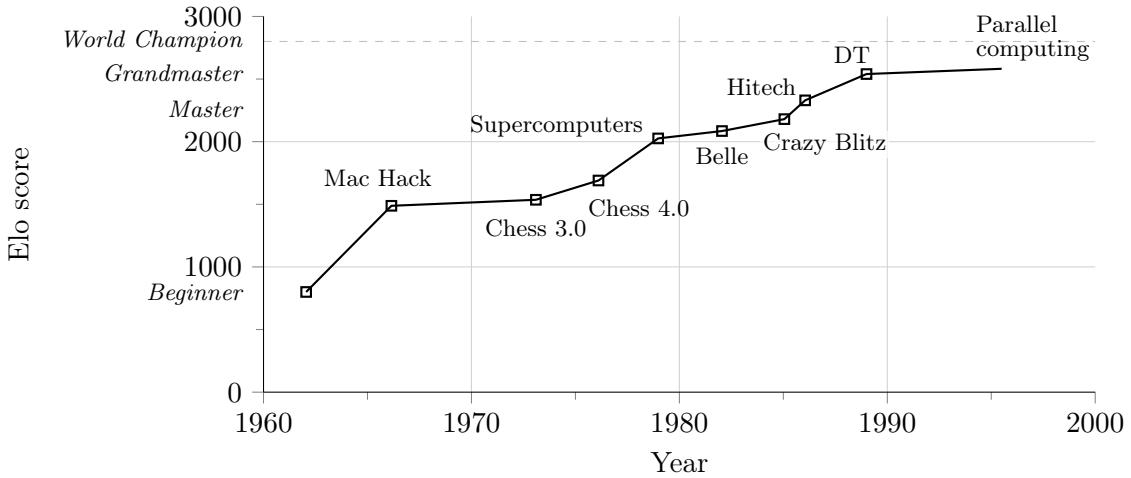


Figure 5.14

MIPS	0.25	0.5	1	1.5	2	3	4	6
ΔElo	-180	-87	0	47	80	124	154	195

8	12	16	24	32	48	64
223	261	287	323	347	379	402

Table 5.1: MIPS (million instructions per second) vs. increase in the Elo score.

However, the same thing cannot be said for the game of Go, due to the much larger number of possibilities allowed by the game. Therefore, in the last decades we have seen a strong growth in the capabilities of computers, but not a parallel strong increase in the understanding of the game. As said by Feng-hsiung Hsu, the father of Deep Thought and Deep Blue, in 2002: “Go is too hard for a computer at the moment”. In fact, at the time Go programs were typically settled around 10 kyu, which is essentially a beginner level (Fig. 5.15). Finally, only in more recent times (2016) DeepMind’s program Alpha Go has made headlines by beating a human professional Go player, the world champion Lee Sedol.

Certainly the size of the board can be an additional source of complexity, since it increases the number of possible moves (a 8×8 board for chess, whereas a 19×19 board for Go). But as shown in Table 5.2, even a 9×9 Go board does not improve the situation of Go programs. So *the size of the board is not the main reason* under the computational “inaccessibility” of Go; It must be searched elsewhere.

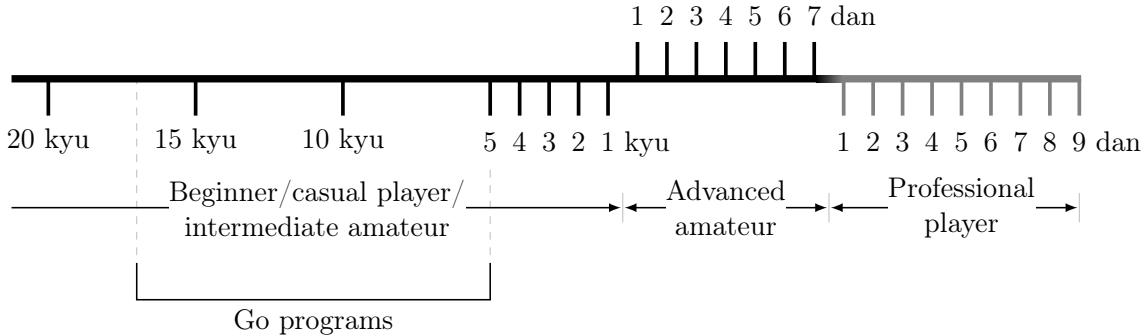


Figure 5.15

Game	$\log_{10}(E)$	$\log_{10}(A)$	Computer-human results
Checkers	17	32	Chinook > H
Othello	30	58	Logistello > H
9 × 9 Go	40	85	Strongest Go program ≪ H
Chess	50	123	Deep Blue ≥ H
15 × 15 Go-moku	100	80	The game is solved
19 × 19 Go	160	400	Strongest Go program ≪ H

Table 5.2: E is the number of legal game positions reachable from the initial position of the game (state-space complexity), while A is the number of leaf nodes in the smallest full-width decision tree that establishes the value of the initial position; a full-width tree includes all nodes at each depth (game-tree complexity). This is an estimate of the number of positions one would have to evaluate in a minimax search to determine the value of the initial position.

In Table 5.3 are presented the most important difference between chess and Go.

1. As said before, the chessboard is made of 8×8 squares, while the traditional Go board is a 19×19 grid.
2. On average a game of chess is concluded after 80 moves, while a Go game is concluded after 300 moves.
3. As we have seen in Chapter 1, the branching factor is the maximum number of successors of any node in the tree. Naturally, every game is different because it can propose a different configuration, with the possibility of having other ramifications. For example, in a chess game, if a “node” is considered to be a legal position, this means that on average a player has about 35 legal moves at their disposal at each turn (naturally, at the beginning we have more possibilities because we start with an empty board, and they reduce during the development).
4. In chess a game ends when a player manages to checkmate the opponent’s king. This definition of “end of game” is clear and concise, since at every moment it is quite easy to control if this condition is met. The same thing cannot be said for the game of Go, because a game ends when both players pass and the winner is determined once all territories have been counted. So the win condition is harder to identify.
5. Another very important difference between chess and Go are the long range effects. In chess many pieces (e.g. queens, rooks, bishops) can move long distances, while in

5 Adversarial search

Go the stone cannot be moved once placed on the board! In the latter the long range effects are due to large scale structures, i.e. to particular patterns and positions of the stones which can affect the opponent's next moves (e.g. ladder, life and death, etc.). These "retarded" effects are very difficult to model, and indeed this difference is presumably the *key feature* behind the computational complexity of Go with respect to chess.

6. A point in favor of Go is the state of the board. In chess the state of the board can change very rapidly because of the high mobility of the pieces, while in Go it mostly changes incrementally, in a smoother way (except in captures).
7. Typically, computer programs for table games use a prototype of *evaluation function*, i.e. a function that, once fed with the positions of the pieces on the board, is able to evaluate the best move for it in order to take advantage of the opponent. For chess is quite easy to find a good correlation with the number and quality of pieces on the board. Instead, for Go we have very poor correlations with the number of stones or the territory surrounded; so the evaluation for Go is very difficult to quantify.
8. The simplicity of chess allows the user to program the computer basing on tree searches, with good evaluation criteria. For Go, instead, we have to many branches for brute-force search, and pruning (reduction of the number of paths to explore) is difficult for the lack of good evaluation measures.
9. A chess player is able to foresee for few sequences up to 10 moves (professional players). A Go player, even a beginner, is able to read up to 60 moves (deep, but narrow searches). In Go there sequences that are essentially forced, like ladders, and so we can read up more positions.
10. We will talk about this later.
11. This point refers to the way the two games are classified and studied.
12. This point refers to how more or less accessible it can be to play with a disadvantage (e.g. using less pieces in chess, or giving some moves of advantage in Go). In theory, chess does not provide a handicap system. In the game of Go this exists and is balanced.

5.2.3 Modeling the three phases of Go

In the opening of the game, players usually play and gain territory in the corners of the board first, as the presence of two edges makes it easier for them to surround territory and establish their stones. From a secure position in a corner, it is possible to lay claim to more territory by extending along the side of the board. The opening is the most theoretically difficult part of the game and takes a large proportion of professional players' thinking time. In the opening, players often play established sequences called *joseki*³ (定石), which are locally balanced exchanges for both black and white sides; however, the *joseki* chosen should also produce a satisfactory result on a global scale. It is generally advisable to keep a balance between territory and influence. The artificial intelligence tool to model this first phase are *neural networks for pattern recognition*.

³There is a go proverb that states that "learning joseki loses two stones in strength", meaning that rote learning of sequences is not advantageous. Rather learning *from* joseki should be a player's goal. Hence the study of joseki is regarded as a double-edged sword and useful only if learned not by rote but rather by understanding the principles behind each move.

	Feature	Chess	Go
1	Board size	8×8 squares	19×19 grid
2	Moves per game	Small (~ 80)	Large (~ 300)
3	Branching factor	~ 35	~ 200
4	End of game and scoring	Checkmate (simple definition – quick to identify)	Counting territory (consensus by players – hard to identify)
5	Long range effects	Pieces can move long distances (e.g., queens, rooks, bishops)	Stones do not move, patterns of stones have long range effects (e.g., ladders; life & death)
6	State of board	Changes rapidly as pieces move	Mostly changes incrementally (except for captures)
7	Evaluation of board positions	Good correlation with number and quality of pieces on board	Poor correlation with number of stones on board or territory surrounded
8	Programming approaches used	Amenable to tree searches with good evaluation criteria	Too many branches for brute-force search, pruning is difficult due to lack of good evaluation measures
9	Human lookahead	Typically up to 10 moves	Even beginners read up to 60 moves (deep but narrow searches, e.g. ladders)
10	Horizon effect	Grandmaster level	Beginner level (e.g., ladders)
11	Human grouping processes	Hierarchical grouping (Chase & Simon, 1973)	Stones belong to many groups simultaneously (Reitman, 1976)
12	handicap system	None	Good handicap system

Table 5.3: Comparison between chess and Go.

The middle phase of the game is the most combative, since we assist to an attempt to consolidate the territory, and usually lasts for more than 100 moves. During the middlegame, the players invade each other's territories, and attack formations that lack the necessary two eyes for viability. Such groups may be saved or sacrificed for something more significant on the board. Also very important are the use of *cuts* (Fig. 5.17) to counter the opponent's game and the creation of eyes (especially double ones) to protect their territories. This phase is modeled through *Monte Carlo methods to evaluate the probability of success of an action*.

The end of the middlegame and transition to the endgame is marked by a few features. Near the end of a game, play becomes divided into localized fights that do not affect each other, with the exception of ko fights, where before the central area of the board related to all parts of it. No large weak groups are still in serious danger. Moves can reasonably

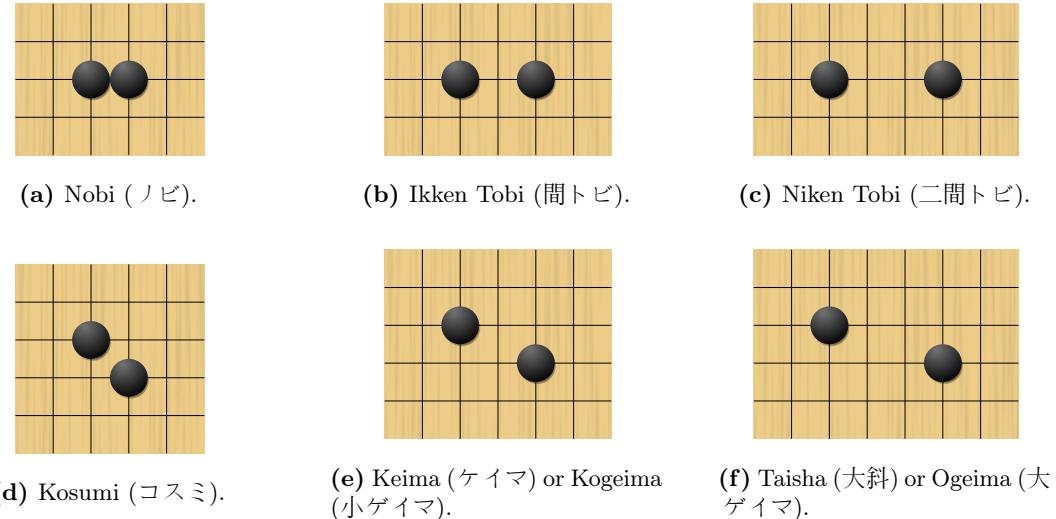


Figure 5.16: Elementary links. To be precise, these are just link structures; a real joseki is a standard sequence of moves also in *response* to the opponent (in other words, we should add also the White in order to represent joseki).

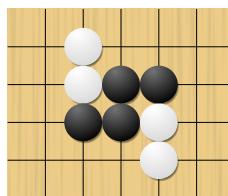


Figure 5.17: *Cutting strings.* The two white strings have been cut by the black string.

be attributed some definite value, such as 20 points or fewer, rather than simply being necessary to compete. Both players set limited objectives in their plans, in making or destroying territory, capturing or saving stones. In other words, in this phase the structures are completed in order to reach the maximum obtainable territory. For this reason, *tactical modules for the search of forced sequences* are typically exploited. In fact, when the game is coming to the very end, the number of possible moves is largely restricted and hence they can be studied analytically, so we can choose the one that maximizes the yield. But don't think that this research is a walk in the park; for example, in chess the possible combinations with 7 pieces left are completely analyzed, i.e. the outcomes are known, but these studies are stored in enormous packages (terabytes!).

5.2.3.1 Programming a Go game

Especially in the early stages of the game, the use of neural networks turns out to be very efficient. The idea of deep learning, namely the use of many hidden layers and hierarchical structures, was already in the air in 1997, but at that time some of the very efficient techniques that we studied in the previous chapters were not yet available.

Neural networks can be used in a variety of different ways: for instance, we can imagine to build a network with 19×19 input neurons and 19×19 output neurons, one for the state (occupied or not) of each point of the grid, in order to predict the fate of every point on the board, rather than just the overall score.

In doing so we can simplify the problem by exploiting the symmetries of the board and of the pattern: color reversal, reflection and rotation of the board. In certain cases, we can

also approximate translation invariance; for example, when we discuss the center of the board we can imagine the grid to have an infinite extension.

Some of the more advanced programs (e.g. Honte, by Dahl, 1999, from the Norwegian Defense Research Establishment) use neural nets together with more conventional AI-methods, like alpha-beta search and stochastic techniques. They also study the influence of the stones over one region through a random walk, i.e. by evaluating the probability to reach a stone of the same (or opposite) color.

5.2.3.2 Coloring the graph

The study of the game of Go is also strictly related to the problem of *coloration of a graph*. A graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense “related”. The objects correspond to mathematical abstractions called *vertices* (also called nodes or points) and each of the related pairs of vertices is called an *edge* (also called link or line). Typically, a graph is depicted in diagrammatic form as a set of dots or circles for the vertices, joined by lines or curves for the edge. In the simplest form, the problem consist in coloring the vertices of a graph such that no two adjacent vertices are of the same color.

Obviously, the answer to this problem depends on the number of colors we have available and on the type of graph we have to examine. For instance, if we have only one color at disposal, we can solve the problem only if there are no edges connecting the vertices. By increasing the number of colors also increases the complexity of the problem. Figure 5.18a shows an example of graph which can be colored with just two colors, whereas Figure 5.18b shows an example of impossible coloring for three colors.

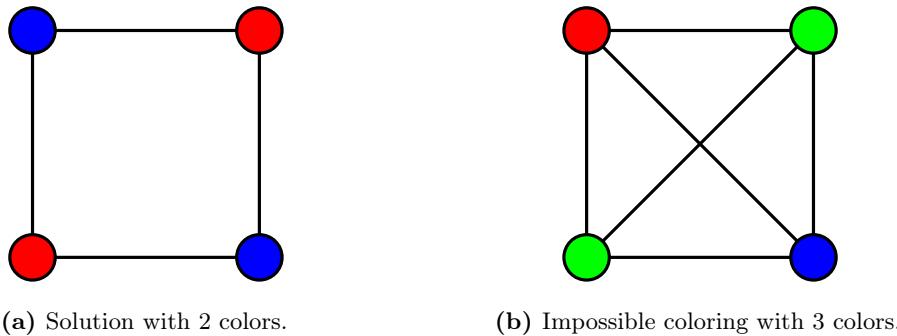


Figure 5.18

Each different graph has its own answer to the problem. Suppose we can randomly generate many graphs with a fixed number of points (n_p) but varying the number of connections (n_l), and we ask if it is possible to color them with a specific number of k colors. Then we can average the single answers for each of the graphs sharing the same ratio n_l/n_p to obtain a probability of positive answer. If we do this and we plot this probability as a function of the ratio between the number of lines and the number of points, n_l/n_p , we obtain something similar to Figure 5.19 (gray curve). Clearly, for $n_l \ll n_p$ the curve approaches 1 (i.e. we are able to color with different colors) because the graph is essentially disconnected. On the other hand, when $n_l \gg n_p$ the probability goes to 0 because we have so many connections that we can't differentiate the colors. In between these two limits we have all the intermediate possibilities, smoothly changing between 0 and 1. The interesting thing is that in the limit of $n_p \rightarrow \infty$ (thermodynamic limit) the system presents a *first-order phase transition*, that is at a critical value $n_l/n_p|_{\text{crit}}$ the probability drops abruptly, leaving

a discontinuity at that position (black curve).

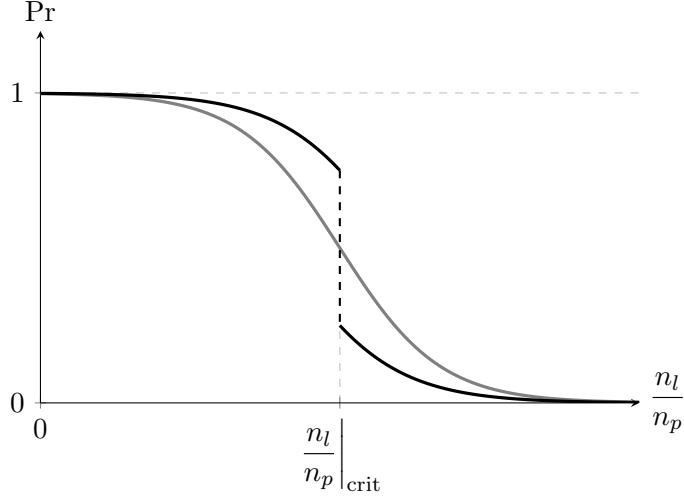


Figure 5.19: Schematic plot of the probability to solve the problem. The gray curve refers to real cases, while the black one refers to the thermodynamic limit ($n_p \rightarrow \infty$).

The game of Go is essentially like a graph where we try to connect stone and understand if the structure is robust (in which case it can survive) or no. So, in some sense the game of Go shares the same complexity of the graph coloring problem. In particular, the goal here is to find the critical value of the ratio n_l/n_p for which the phase transition occurs. This is a very difficult problem, i.e. it is a NP-complete; but since any problem belonging to this class can be recast into any other member of the class (recall Sec. 1.2.1.2), then if we are able to find a solution for the graph coloring problem we are in principle also able to solve the game of Go.

5.3 Adversarial search

In Chapter 1 we talked about agents and the different ways they interact with the environment. However there exists also *multiagent environments*, in which each agent needs to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce contingencies into the agent's problem-solving process. So in this section we cover competitive environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems, often known simply as **games**.

Mathematical game theory, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is “significant”, regardless of whether the agents are cooperative or competitive. In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, *zero-sum games* of perfect information (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. In other words, an advantage that is won by one of two sides is lost by the other. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

Games, unlike most of the toy problems studied in Chapter 1, are interesting because they are too hard to solve. For example, the search tree of chess has about 10^{154} nodes, although the search graph has “only” about 10^{40} distinct nodes. This means the game

tree is best thought of as a theoretical construct that we cannot realize in the physical world. Games therefore require the ability to make some decision even when calculating the *optimal* decision is infeasible. Games also penalize inefficiency severely. Whereas an implementation of A* search that is half as efficient will simply take twice as long to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal. Game-playing research has therefore spawned a number of interesting ideas on how to make the *best possible use of time*.

We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. The function defining the final numeric value for a game that ends in the terminal state is called **utility function** (also called an objective function or payoff function). A game can be formally defined as a kind of search problem along the game tree—a tree where the nodes are game states and the edges are moves.

Figure 5.20 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names). So the score is +1 if MAX wins, -1 if MAX loses and 0 if the game is a draw; notice that all the utility values sum up to 0 (zero-sum game).

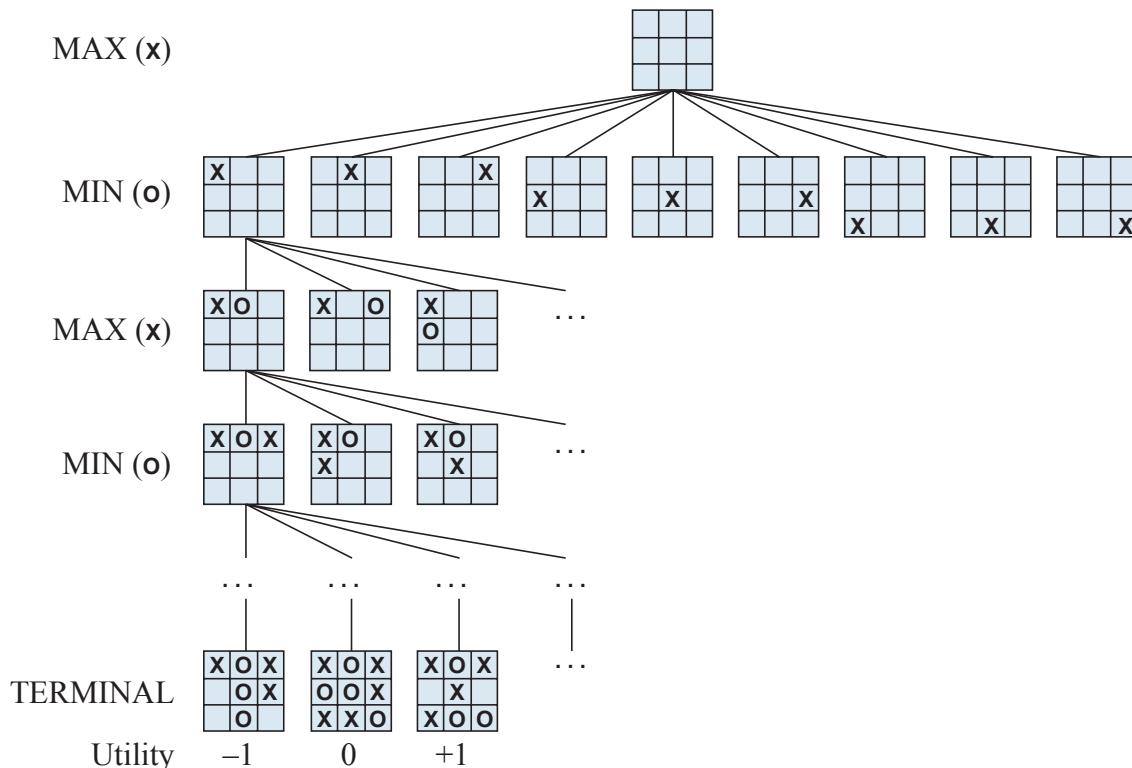


Figure 5.20

5.3.1 Optimal decision: minimax

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something to say about it. MAX therefore must find a contingent *strategy*.

Consider for instance the trivial game in Figure 5.21. The possible moves for MAX at the root node are labeled a_1, a_2, a_3 . The possible replies to a_1 for MIN are b_1, b_2, b_3 , and so on. The Δ nodes are “MAX nodes”, in which it is MAX’s turn to move, and the ∇ nodes are “MIN nodes”. This particular game ends after one move each by MAX and MIN. The utilities of the terminal states in this game range from 2 to 14. Given a game tree, the optimal strategy can be determined from the **minimax value** of each node. The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. This means that MAX tries to maximize its yield, while MIN tries to minimize MAX’s result⁴. Obviously, the minimax value of a terminal state is just its utility.

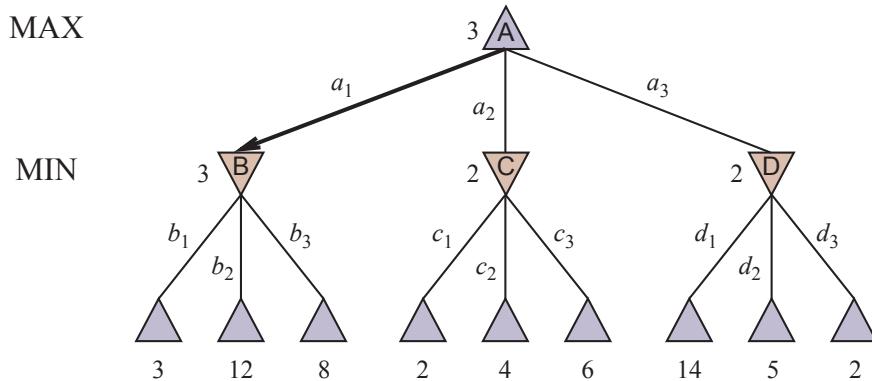


Figure 5.21

In the particular case of Fig. 5.21, the first MIN node, labeled B, has three successor states with utility values 3, 12 and 8, so its minimax value is 3. In fact, if MAX reaches the node B and MIN plays at its best, then MAX will necessarily finish at the node with utility function 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2 and 2, so it has a minimax value of 3 (MAX’s nodes always have the minimax value of the successor node with highest minimax value). We can also identify the *minimax decision* at the root: action a_1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are *backed up* through the tree as the recursion unwinds. For example, in Figure 5.21, the algorithm first recurses down to the three bottom-left nodes and uses the utility function on them to discover that their values are 3, 12 and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D.

⁴Put simply, each player tries to increase their yield, but sometimes it can be more useful to minimize that of the opponent (the two possibilities are not always interchangeable).

5.3.1.1 Multiplayer games

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a vector of values. For example, in a three-player game with players A, B and C, a vector (n_A, n_B, n_C) is associated with each node (Fig. 5.22). For terminal states, this vector gives the utility of the state from each player's viewpoint (in two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite).

Now we have to consider non-terminal states. Consider the node marked X in the game tree shown in Figure 5.22. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $(1, 2, 6)$ and $(4, 2, 3)$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $(1, 2, 6)$. Hence, the backed-up value of X is this vector. The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n .

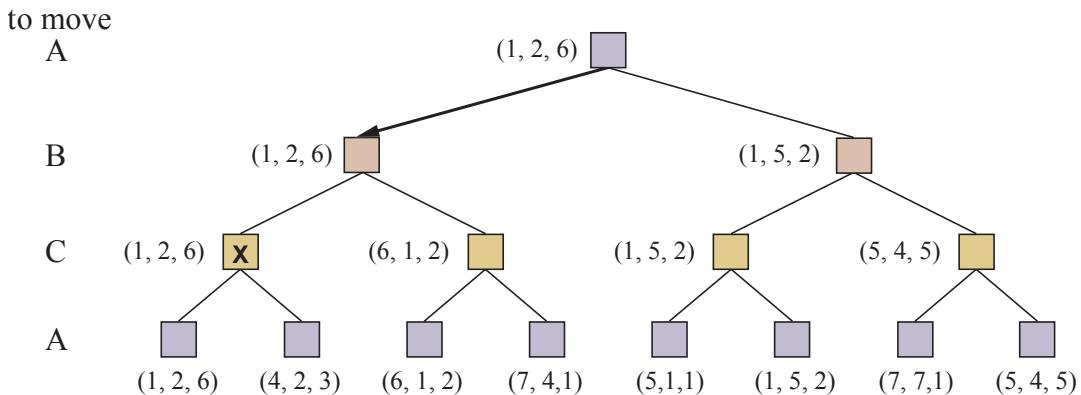


Figure 5.22

Anyone who plays multiplayer games quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve *alliances*, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. Alliances are still a natural consequence of optimal strategies for each player in a multiplayer game. For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement. So players must balance the immediate advantage of joining forces against the long-term advantage that other players can benefit.

5.3.2 Alpha–beta pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of *pruning* to eliminate large parts of the tree from consideration. The particular technique we examine is called **alpha–beta pruning**. When applied to

a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree, such that the player at root has a choice of moving to that node. If that layer has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

In practice let's consider the example of Figure 5.23, showing the stages in the calculation of the optimal decision. At each point are shown the range of possible minimax values for each node. (a) We start by examining the deepest nodes of the tree; the first leaf below B has the utility value 3. Hence B, which is a MIN node, has a minimax value of *at most* 3 (restricted to the interval $(-\infty, 3]$). (b) The second leaf below B has a utility value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8, i.e. again larger than the other; we have seen all B's successor states, so we can conclude that the minimax value of B is exactly 3. Now, we can also infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) Let's now analyze the other branches. The first leaf below C has the value 2. Hence C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

The effectiveness of alpha–beta pruning is highly dependent on the *order* in which the states are examined. For example, in (e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of D had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best. To do this, however, we need an *a priori* knowledge of the tree.

5.3.3 Evaluation function

The minimax algorithm generates the entire game search space, whereas the alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. E.g., in chess is practically impossible to reach analytically the terminal state.

Claude Shannon's paper "Programming a Computer for Playing Chess" (1950) proposed instead that programs should cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning non-terminal nodes into terminal leaves. An evaluation function returns an *estimate* of the expected utility of the game from a given position, just as the heuristic functions of Chapter 1 return an estimate of the distance to the goal. How exactly do we design good evaluation functions?

- First, the evaluation function should order the terminal states in the same way as the true utility function: states that are wins must evaluate better than draws, which in

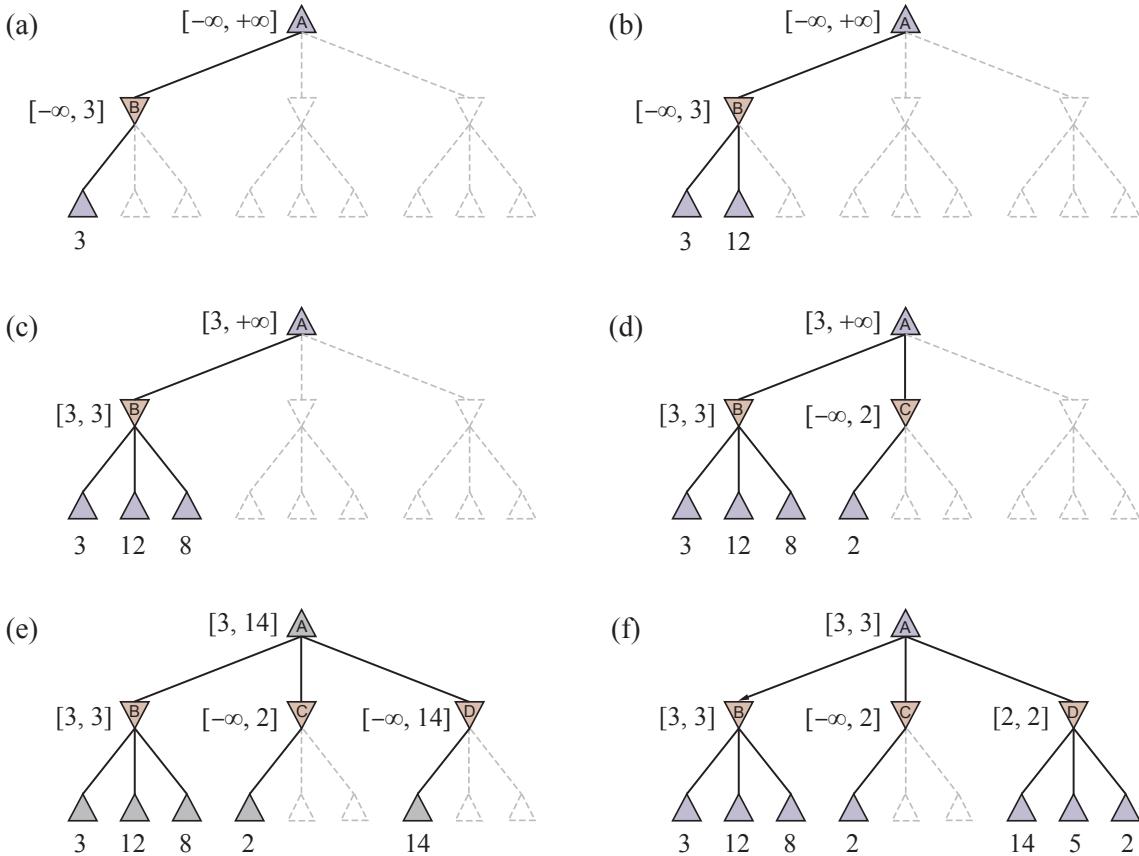


Figure 5.23

turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game.

- Second, the computation must not take too long (the whole point is to search faster!).
- Third, for non-terminal states, the evaluation function should be strongly correlated with the actual chances of winning.

One might well wonder about the phrase “chances of winning”. After all, chess is not a game of chance, since we know the current state with certainty. But if the search must be cut off at non-terminal states, then the algorithm will necessarily be uncertain about the final outcomes of those states. This type of uncertainty is induced by computational, rather than informational, limitations. Given the limited amount of computation that the evaluation function is allowed to do for a given state, the best it can do is make a guess about the final outcome.

Most evaluation functions work by calculating various *features* of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. The features, taken together, define various *categories* or equivalence classes of states: the states in each category have the same values for all the features. For example, one category contains all two-pawn vs. one-pawn endgames. Any given category, generally speaking, will contain some states (position of those pieces) that lead to wins, some that lead to draws and some that lead to losses. The evaluation function cannot know which states are which, but it can return a single value that reflects the proportion of states with each outcome. For example, suppose our experience suggests

that 72 % of the states encountered in the two-pawns vs. one-pawn category lead to a win (utility +1), 20 % to a loss (-1) and 2 % to a draw (0). Then a reasonable evaluation for states in the category is the weighted average of the values (also called expected value). In principle, the expected value can be determined for each category, resulting in an evaluation function that works for any state.

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning. Instead, most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value. For example, introductory chess books give an approximate “material value” for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5 and the queen 9. Other features such as “good pawn structure” and “king safety” might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position. Mathematically, this kind of evaluation function is essentially a weighted linear combination, because it can be expressed as

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s) \quad (5.3.1)$$

where each w_i is a weight and each f_i is a feature of the position. For chess, the f_i could be the numbers of each kind of piece on the board, and the w_i could be the values of the pieces (1 for pawn, 3 for bishop, etc.). The weights aren’t necessarily constant; they are application coefficients that can vary with stage of game (opening, middle game, endgame), pieces on the board (e.g. presence or absence of queens), other characteristics of the position, or high level strategy or plans.

The astute reader will have noticed that the features and weights are not part of the rules of chess! They come from centuries of human chess-playing experience. It should be clear that *the performance of a game-playing program depends strongly on the quality of its evaluation function*. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.

5.3.3.1 Horizon effect

The idea of an evaluation function is very powerful in reducing the search tree, but in some particular cases, the fact of having cut the search tree can cause problems. An intuitive example is so-called **horizon effect**, that arises when the program is facing an opponent’s move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics.

Consider for instance the chess state depicted in Figure 5.24, in which is the turn of Black to move. By moving in a1 the black rook can actually check the white king. However, in response the king can move one position diagonally (in g2) to get out of check. The rook can then repeat the check by moving in a2, but in turn the king can once again delay the defeat by moving diagonally (in f1). One may therefore wonder which of the two sides is advantaged. This configuration is in whose favor?

It is clear that the line made up of white pawns is trapping the king under the influence of the rook. However, a human player would understand that the block is actually temporary, since after some zigzagging the king can in reality escape the block and move with freedom across all the board, with the possibility to exploit also the other pieces. In particular, the white pawn in d7 may be easily promoted to a queen, giving a huge advantage to White and securing the king. Thus, a human player would claim that White is currently in the lead.

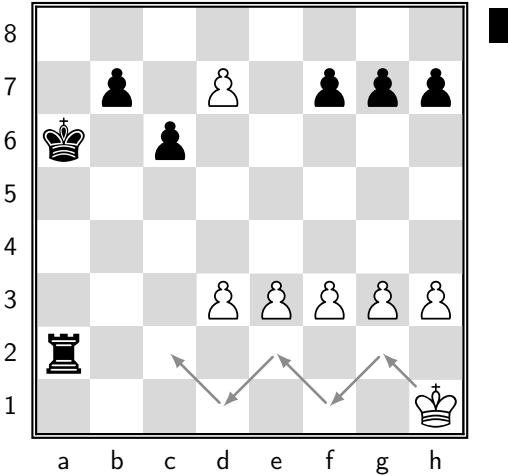


Figure 5.24: Horizon effect. It's up to Black to move.

Instead, the evaluation of a computer might be different, depending on the depth of the search. For instance, if the algorithm stops analyzing the search tree only after 3 moves from the state of Figure 5.24, then it will say that Black is in the lead because it has a rook and the opponent only pawns. On the other hand, if the algorithm is implemented sufficiently in depth to explore many more moves, i.e. to see over the horizon, then it will notice that the pawn might become a queen, giving an important jump in the evaluation function for White. In other words, a series of checks by the black rooks forces the inevitable queening move by White “over the horizon” and makes this position look like a win for Black, when it is really a win for White.

This horizon effect is inherent in adopting a truncated search tree and cannot be eliminated, but it can still be mitigated (e.g., by introducing exceptions in the algorithm that allow specific solving moves for specific configurations leading to the horizon effect). This effect is even more emphasized in the game of Go because of the presence of larger structures, where the algorithm would be forced to look more than 10 moves forward to see right.

5.3.4 Monte Carlo tree search

Another very powerful technique to study board games is **Monte Carlo tree search** (MCTS), which in some sense computes an evaluation function in an automatic way. Its purpose is to analyze the most promising moves by expanding the search tree based on random sampling of the search space. The application of Monte Carlo tree search in games is based on many *playouts*, also called *roll-outs*. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

The algorithm is organized in four steps:

- **Selection:** The algorithm searches the portion of the tree that has already been represented in the memory. Starting from the root node R of the search tree, we go down the tree by repeatedly selecting legal moves according to a specific policy until we reach a leaf node L . Recall that the root is the current game state (from which the machine is asked take the optimal decision) and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated. So if one,

5 Adversarial search

several or all of the legal moves in a node does not have a corresponding node in the search tree, we stop selection. We will return on how to efficiently choose the path later on (see below).

- **Expansion:** Unless selection reached a terminal state, i.e. unless L ends the game decisively for either player (e.g. win/loss/draw), expansion adds a new child node to the tree represented in memory. In practice we create its possible child nodes, which correspond to any available action from the game position defined by L (so we expand L), and we choose to explore node C from one of them. The new node corresponds to a state that is reached by performing the last action in the selection phase. When expansion reaches a terminal state, which is extremely rare, then the current iteration skips directly to backpropagation.
- **Simulation:** Starting from the newly-created node in the expansion phase, let's perform a complete random simulation (a playout, in our notation) of the game/problem by selecting each move at random till the end. As said before, a playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost or drawn). No new nodes are created in this phase. This is the "Monte Carlo" part of the algorithm.
- **Backpropagation:** Once a winner has emerged from the simulation phase, use the result of the playout to propagate the information back to all nodes along the path from the last visited node in the tree (C) to the root (R). In this phase the statistics are updated: each visited node has its simulation count (the denominator) incremented, while its win count may also be incremented, depending on which player has won the simulation.

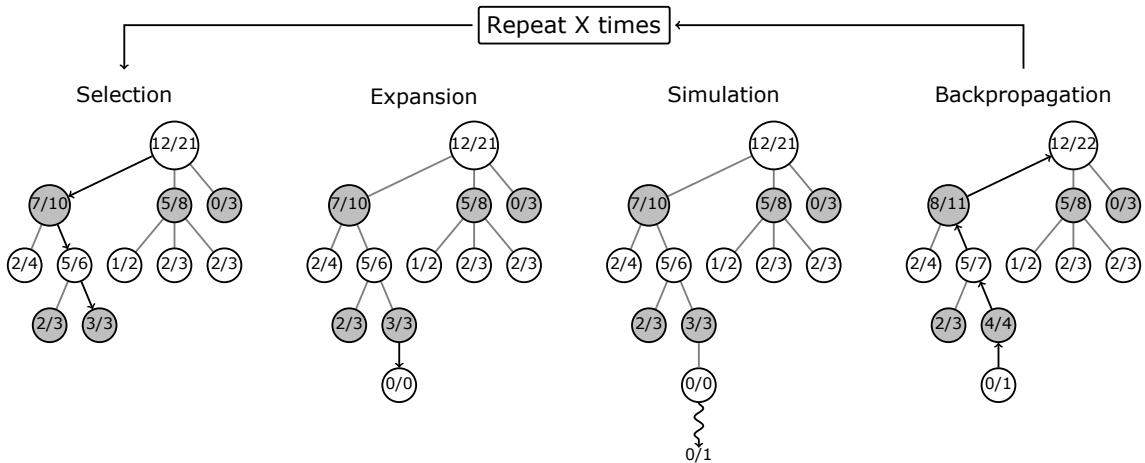


Figure 5.25

Refer for instance to Figure 5.25. This graph shows the steps involved in one decision (i.e. one move in the real world), with each node presenting the ratio of wins to total playouts from that point in the game tree for the player that the node represents. For example, the root node says there are 12 wins out of 21 playouts for White from this position so far. Once chosen a path till a leaf node (shown in bold), we choose one of its possible child nodes; we initialize it with the ratio 0/0. At this point we start a new random game from that specific node and depending on the outcome we consequently update the numbers of the nodes along the chosen path. For instance, if White loses the simulation,

all nodes along the selection incremented their simulation count (the denominator), but among them only the black nodes were credited with wins (the numerator). If instead White wins, all nodes along the selection would still increment their simulation count, but among them only the white nodes would be credited with wins. In games where draws are possible, a draw causes the numerator for both black and white to be incremented by 0.5 and the denominator by 1.

As we have previously anticipated, the choice of the path in the first step can severely affect the efficiency of the algorithm, and therefore the proposition of an optimal move. Our path selection should achieve two goals: we should *explore* new paths to gain information on as many moves as possible, and we should also use existing information to *exploit* paths known to be good (i.e. giving a high win rate). In order to help us achieve these two goals, we need to select child nodes using a *selection function that balances exploration and exploitation*. One (bad) way to do this is to select paths randomly. Random selection certainly does explore well, but it does not exploit at all. Another (equally bad) way is to use the average win rate of each node. This achieves good exploitation, but it scores poorly on exploration; it risks discarding a move which can be good, but that has not been sufficiently explored to be able to confirm this. Luckily, some very smart people have figured out a good selection function that balances exploration with exploitation well, called UCB1 (Upper Confidence Bound 1). When applied to MCTS, the combined algorithm is named UCT (Upper Confidence bound 1 applied to Trees). So, MCTS + UCB1 = UCT. The UCB1 selection function is given by

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln s_p}{n_i}} \quad (5.3.2)$$

where

- w_i is the number of simulations that result in a win for the considered node after the i -th move;
- n_i is the number of simulations for the considered node after the i -th move;
- s_p is the total number of simulations after the i -th move run by the parent node of the one considered;
- c is the exploration parameter, in practice usually chosen empirically.

The left term is the *exploitation* term. It is simply the average win rate, going larger the better a node has historically performed. The right term is the *exploration* term. It goes larger the less frequently a node is selected for simulation. The exploration parameter c is just a number we can choose that allows us to control how much the equation favors exploration over exploitation. Note that the numbers inside the nodes in the tree diagram of Figure 5.25 are the statistics for that node, corresponding to number of wins and total number of simulations (w_i and n_i). Each time we need to select a path across multiple nodes, we use the UCB1 selection function to get a value for each child node, and we select the child node with the *maximum* value. This also means that the selected path may not be the same after each round, i.e. after a sequence of these four steps, because the update taking place in the “Backpropagation” can modify the hierarchy of UCB1 values in the tree. So, MCTS does not need an explicit evaluation function; simply implementing the game’s mechanics is sufficient to explore the search space.

In summary, during one round of search the path is selected according to the right balance between win rate and exploration frequency. Then, rounds of search are repeated

as long as the time allotted to a move in the real world is respected. The efficiency of this method often increases with time as more playouts are assigned to the moves that have frequently resulted in the current player’s victory according to previous playouts. Finally, only the *move with the most simulations made* (i.e. the highest denominator) is chosen as the final answer. That is in the “training” we evaluate the possible moves according to UCB1, but the ultimate choice falls on the move that has seen the greatest number of simulations.

5.4 Reinforcement learning

Reinforcement learning (RL) is one of the most exciting fields of machine learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years, particularly in games and in machine control, but seldom making the headline news. But a revolution took place in 2013, when researchers from a British startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch, eventually outperforming humans in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games. This was the first of a series of amazing feats, culminating in March 2016 with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go, and in May 2017 against Ke Jie, the world champion.

In reinforcement learning, a software agent makes observations and takes actions within an environment, and in return it receives *rewards*. Its objective is to learn to act in a way that will maximize its expected rewards *over time*. You can think of positive rewards as bonuses, and negative rewards as maluses or penalties (the term “reward” is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its bonuses and minimize its penalties.

5.4.1 Policy search

The algorithm a software agent uses to determine its actions is called its *policy*. The policy could be a neural network taking observations as inputs and outputting the action to take, for instance. The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment!

For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 - p$ (the two options are exclusive). The rotation angle would be a random angle between $-r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two policy parameters you can tweak: the probability p and the angle range $2r$. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 5.26). This is an example of **policy search**, in this case using a brute-force approach. When the policy space is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then “kill” the 80 worst policies and make the 20 survivors produce 4 offspring each. An offspring is a copy of

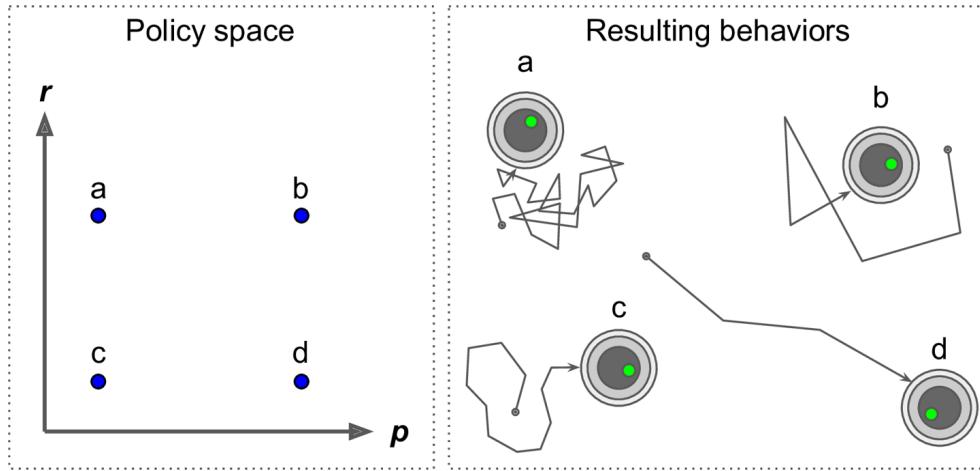


Figure 5.26: Four points in policy space (left) and the agent’s corresponding behavior (right).

its parent plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy.

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards. We will discuss this approach, called **policy gradients** (PG), in more detail later in this section. Going back to the vacuum cleaner robot, you could slightly increase p and evaluate whether doing so increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase p some more, or else reduce p .

5.4.2 Introduction to OpenAI gym

One of the challenges of reinforcement learning is that in order to train an agent, you first need to have a working environment. The problem is that training is hard and slow in the real world (we can’t speed up time or undo certain moves), so you generally need a *simulated environment* at least for bootstrap training. OpenAI Gym is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

Let’s consider for instance the cart-pole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 5.27). Here the parameters that enter our analysis are the angle ϑ of the pole with respect to the vertical (positive means clockwise), its angular velocity $\dot{\vartheta}$, the cart’s horizontal position x , and its linear velocity \dot{x} . This means that one observation of the environment corresponds to a 1D array containing just these four floats.

Let’s create a policy by using neural networks. Just like with the policy we hardcoded earlier, this neural network will take an observation as input, and it will output the action to be executed. More precisely, it will estimate a probability for each action, and then we will select an action randomly, according to the estimated probabilities. In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be $1 - p$. For example, if it outputs 0.7, then we will pick action 0 with 70 % probability, or action 1 with 30 % probability.

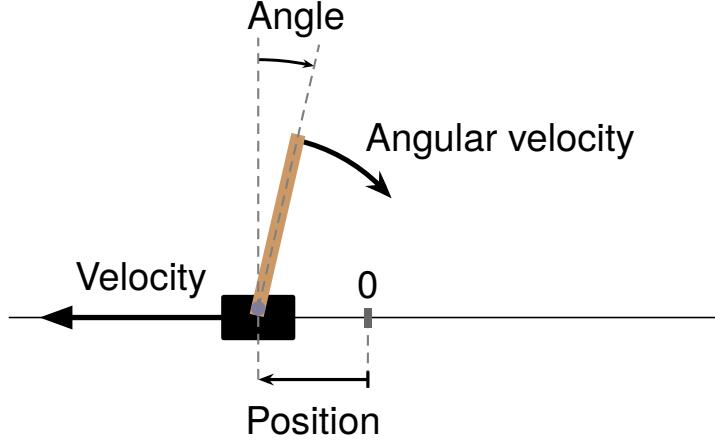


Figure 5.27

You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state.

In summary, the structure of neural network is the following: we have exactly 4 input neurons, since in the case of CartPole the size of the observation space is 4, and we have just five hidden units because it's a simple problem. The hidden units are implemented with the exponential linear unit (ELU) activation function

$$f_\alpha(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha(e^z - 1), & \text{if } z < 0 \end{cases} \quad (5.4.1)$$

which is linear for positive values and negative and slowly flattening for $z \rightarrow \infty$. The ELU activation function looks a lot like the ReLU function, with a few major differences: it takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem; it has a nonzero gradient for $z < 0$, which avoids the dead neurons problem; if α is equal to 1 then the function is smooth everywhere, including around $z = 0$, which helps speed up gradient descent since it does not bounce as much to the left and right of $z = 0$.

Finally, we want to output a single probability (the probability of going left), so we have a single output neuron using the sigmoid activation function. If there were more than two possible actions, there would be one output neuron per action, and we would use the softmax activation function instead. Naturally, the network is fully connected (see Figure 5.28).

5.4.3 The credit assignment problem

Ok, we now have a neural network policy that will take observations and output action probabilities. But how do we train it? If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in reinforcement learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example,

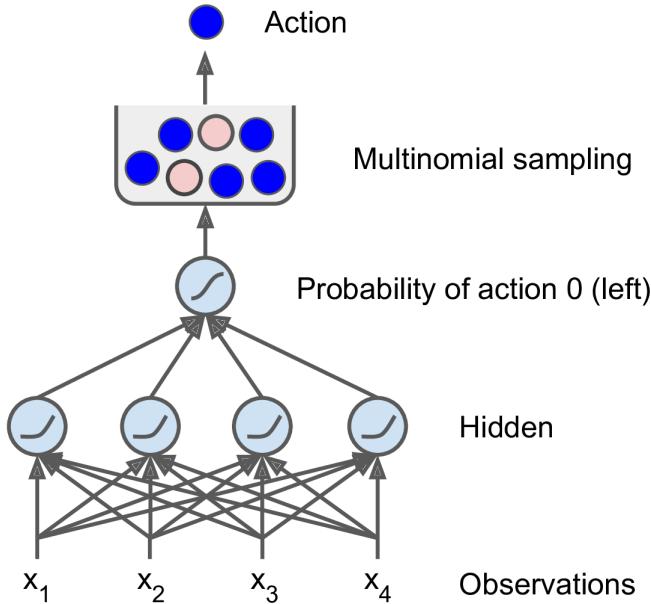


Figure 5.28

if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the **credit assignment problem**: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it.

In other words, just a single observation is not enough to estimate the action that should presumably give the highest reward over time. For example, imagine the case in which the pole is tilted toward right by a small amount and its angular velocity is high and negative (i.e. directed counterclockwise). Since the pole is tilted, one may be pushed to give the cart a small acceleration in the right direction, in order to balance the angular momentum and bring it back closer to the vertical. But in doing so, because of the high angular velocity, we are likely to end up in a new state with still small angle, now negative (this seems good), but with a very high angular velocity (this is bad). This means that probably whatever further move may not be sufficient to prevent the pole from falling. Therefore, even if the first move left the pole standing, thus obtaining a positive reward, it led to a certain fall in the following instants. So evaluating just one snapshot can give bad long-terms results.

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come *after* it, usually applying a *discount factor* γ at each step. This sum of discounted rewards is called the *action's return*. Consider the example in Figure 5.29. If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally -50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \cdot 0 + \gamma^2 \cdot (-50) = -22$. If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount factors vary from 0.9 to 0.99.

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return. However, if we play the game enough times, on average initial good actions will get a higher return than bad ones. Our goal is to estimate how much better or worse an action is, compared to the other possible

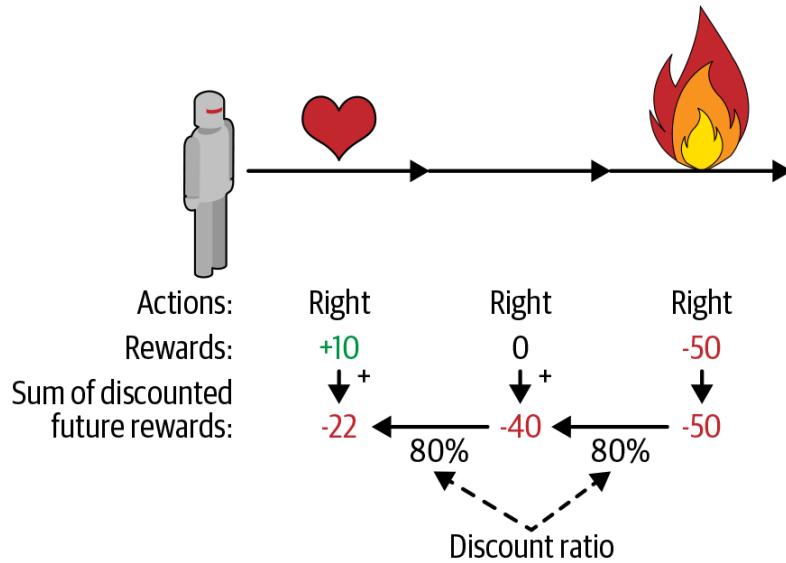


Figure 5.29

actions, on average. This is called the *action advantage*. For this, we must run many episodes and normalize all the action returns (by subtracting the mean and dividing by the standard deviation); the fact of running many episodes naturally introduces some sort of randomization (the outcomes typically differ from each other), which is very similar to Monte Carlo tree search. After that, we can reasonably assume that actions with a negative advantage were bad while actions with a positive advantage were good. Perfect—now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients.

5.4.4 Policy gradients

As discussed earlier, policy gradient algorithms optimize the parameters of a policy by following the gradients toward higher rewards (they are indeed called “gradient ascent” methods). One popular class of policy gradient algorithms, called *REINFORCE algorithms*, was introduced back in 1992. Here is one common variant:

1. First, let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don’t apply these gradients yet.
2. Once you have run several episodes, compute each action’s advantage (using the method described in the previous section).
3. If an action’s advantage is positive, it means that the action was probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the action’s advantage is negative, it means the action was probably bad, and you want to apply the opposite gradients to make this action slightly less likely in the future. The solution is simply to multiply each gradient vector by the corresponding action’s advantage.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a gradient descent step.

5.4.5 Markov decision processes

We will now look at another popular family of algorithms, which are less direct than policy gradient algorithms. To understand these algorithms, we must first introduce the concept of Markov process. In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s') , not on past states; this is why we say that the system has no memory. Notice that we have already met Markov chains, although indirectly, in some previous paragraphs, for example when we talked about the Metropolis algorithm or the Monte Carlo tree search.

For instance, Figure 5.30 shows a basic example of Markov chain. Suppose that the process starts in state s_0 , and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back because no other state points back to s_0 . If it goes to state s_1 , it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever (this is a terminal state). Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

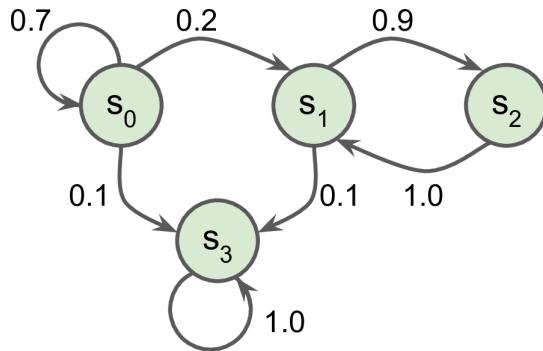


Figure 5.30

The natural extension of Markov chains are the so-called **Markov decision processes** (MDP), which were first described in the 1950s by Richard Bellman. They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

For example, the Markov decision process represented in Figure 5.31 has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds). If it starts in state s_0 , the agent can choose between actions a_0 , a_1 or a_2 . If it chooses action a_1 , it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0 , it has a 70% probability of gaining a reward of +10 and remaining in state s_0 . It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1 . In state s_1 it has only two possible actions: a_0 or a_2 . It can choose to stay put by repeatedly choosing action a_0 , or it can choose to move on to state s_2 and get a negative reward of -50. In state s_2 it has no other choice than to take action a_1 , which will most likely lead it back to state s_0 , gaining a huge reward of +40 on the way. By looking at this

5 Adversarial search

MDP, it is immediate to guess which strategy will gain the most reward over time: in state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1 , but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

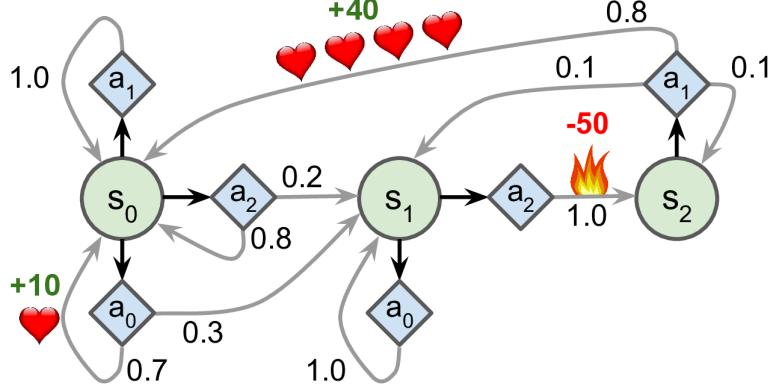


Figure 5.31

Bellman found a way to estimate the *optimal state value* of any state s , noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state s , assuming it acts optimally. He showed that if the agent acts optimally, then the **Bellman optimality equation** applies

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad (5.4.2)$$

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s' , given that the agent chose action a . For example, in Figure 5.31 $T(s_2, a_1, s_0) = 0.8$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s' , given that the agent chose action a . For example, in Figure 5.31, $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

So this recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

In summary, this equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero (a sort of 0-th order approximation), and then you iteratively update them using the previous value:

$$V_{k+1}(s) \rightarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \quad (5.4.3)$$

In this equation, $V_k(s)$ is the estimated value of state s at the k -th iteration of the algorithm. A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar

algorithm to estimate the optimal *state-action values*, generally called **Q-Values** (Quality Values). The optimal Q-Value of the state-action pair (s, a) , noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a , but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the Q-Value iteration algorithm

$$Q_{k+1}(s, a) \rightarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right] \quad (5.4.4)$$

Note that T in this and previous equations takes the role of kernel of an integral equation (even if now it is discrete). Anyway, once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s , it should choose the action with the highest Q-Value for that state:

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a) \quad (5.4.5)$$

5.5 Alpha Go, AlphaGo Zero & Alpha Zero

AlphaGo is a computer program that plays the board game Go. It was originally developed by DeepMind Technologies, which was later acquired by Google. Subsequent versions of AlphaGo became increasingly powerful, including a version that competed under the name Master. After retiring from competitive play, AlphaGo Master was succeeded by an even more powerful version known as **AlphaGo Zero**, which was completely self-taught without learning from human games (except for the rules). AlphaGo Zero was then generalized into a program known as **AlphaZero**, which played additional games, including chess and shogi⁵. AlphaZero has in turn been succeeded by a program known as *MuZero* which learns without being taught the rules.

In October 2015, in a match against the strong European player Fan Hui, the original AlphaGo became the first computer Go program to beat a human professional Go player without handicap on a full-sized 19×19 board. This was a big surprise, because for the first time artificial intelligence turned out to be able to generate machines capable of competing with humans even at high levels. This event soon made headlines, and with this result the creators of AlphaGo decided to contact the South Korean ex World Champion Lee Sedol in order to organize a match. When they contacted him, they knew the machine was still not ready, but by checking the machine efficiency they were confident of its increase in the following times. In fact, in March 2016 the meeting took place and AlphaGo beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicap. Although it lost to Lee Sedol in the fourth game, Lee resigned in the final game, giving a final score of 4 games to 1 in favor of AlphaGo. After the match between AlphaGo Master and Ke Jie, the number one ranked player in the world at the time (2017), DeepMind retired AlphaGo, while continuing AI research in other areas.

From that day on it was clear that future programs that play Go could only be compared with other machines, because they have now reached qualities far superior to those of humans (Elo score well above 3000). This is exactly the same thing that happened with chess, only about 20 years late.

⁵Shogi (shōgi, 将棋) is a Japanese variant of chess. Literally, shōgi means general's (shō, 将) board game (gi, 棋).

5.5.1 The algorithm

5.5.1.1 The pillars

Before the advent of AlphaGo, a typical computer chess program was based on a deterministic tree search, in the sense of non-stochastic. For example, the powerful Stockfish implements an advanced alpha–beta search, that seeks to decrease the number of nodes that are evaluated by the minimax algorithm. As said in the previous sections, alpha–beta pruning stops evaluating a move when at least one previously examined possibility has been proved to be better, thus cutting away branches that cannot possibly influence the final decision. But the search time is still limited (the analysis of the whole tree would take a long time); therefore it usually stops at a certain depth and evaluates the goodness of a position by introducing the concept of *evaluation function*.

As said in [21]:

Each position s is described by a sparse vector of handcrafted features $\Phi(s)$, including midgame/endgame-specific material point values, material imbalance tables, piece-square tables, mobility and trapped pieces, pawn structure, king safety, outposts, bishop pair, and other miscellaneous evaluation patterns. Each feature ϕ_i is assigned, by a combination of manual and automatic tuning, a corresponding weight w_i and the position is evaluated by a linear combination $v(s, \mathbf{w}) = \Phi^T \mathbf{w}$.

The mathematical formulation behind the evaluation function heavily relies on human knowledge, and its quality is what distinguishes a good chess program to a bad one, because just a little error in the evaluation can generate serious problems in choosing the optimal move. Also, we need to take care of phenomena originating from having pruned the tree, like the horizon effect.

On the other hand, AlphaZero does not require any human expertise as input. This means that the underlying methodology of AlphaGo Zero can be applied to any game with perfect information (the game state is fully known to both players at all times) because no prior expertise is required beyond the rules of the game. As said by the authors of [15]:

AlphaZero replaces the handcrafted knowledge and domain-specific [i.e. by having in mind a specific game with specific rules] augmentations used in traditional game-playing programs with deep neural networks, a general-purpose reinforcement learning algorithm, and a general-purpose tree search algorithm.

Going on from [14], instead of a handcrafted evaluation function and move-ordering heuristics:

Our new method uses a deep neural network \mathbf{f}_θ with parameters θ . This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\mathbf{p}, v) = \mathbf{f}_\theta(s)$. The vector of move probabilities \mathbf{p} represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s .

and in [15]:

Instead of an alpha–beta search with domain-specific enhancements, AlphaZero uses a general-purpose Monte Carlo tree search (MCTS) algorithm. Each search consists of a series of simulated games of self-play that traverse a tree from root state s_{root} until a leaf state is reached. Each simulation proceeds by selecting in each state s a move a with low visit count (not previously frequently explored), high move probability, and high [action] value (averaged over the leaf states of simulations that selected a from s) according to the current neural network \mathbf{f}_{θ} . The search returns a vector π representing a probability distribution over moves, $\pi_a = \Pr(a|s_{\text{root}})$.

Therefore, **deep neural networks** and **MCTS** are the main ingredients of the algorithm, that must be combined and iterated throughout the estimation of a move.

Remember that minimax algorithm can be problematic if implemented with a hand-crafted evaluation function, since any small inaccuracy can be a disaster for the machine game. This was exactly the problem of using neural networks to model minimax search. In fact, neural networks are examples of fitting machines, that is, they are not interpolating machines, and therefore they unavoidably introduce some error in the evaluation which can turn out to be a catastrophe for the outcome of the game. Instead, by adopting the stochastic Monte Carlo tree search we are making a sort of average of the values (win rates and explorations) obtained in the previous branches. That's why the combination neural networks plus MCTS proved to be surprisingly so effective.

For what concern the MCTS ([14]):

The MCTS uses the neural network \mathbf{f}_{θ} to guide its simulations (see Fig. 5.32). Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximize an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$, until a leaf node s' is encountered. This leaf position is expanded and evaluated only once by the network to generate both prior probabilities and evaluation, $(P(s', \cdot), V(s')) = \mathbf{f}_{\theta}(s')$. Each edge (s, a) traversed in the simulation is updated to increment its visit count $N(s, a)$, and to update its action value to the mean evaluation over these simulations, $Q(s, a) = 1/N(s, a) \sum_{s' \mid s, a \rightarrow s'} V(s')$ where $s, a \rightarrow s'$ indicates that a simulation eventually reached s' after taking move a from position s .

MCTS may be viewed as a self-play algorithm that, given neural network parameters θ and a root position s , computes a vector of search probabilities recommending moves to play, $\pi = \alpha_{\theta}(s)$, proportional to the exponentiated visit count for each move, $\pi_a \propto N(s, a)^{1/\tau}$, where τ is a temperature parameter.

So, from these lines it is clear what the creators of AlphaZero meant before by saying “low visit count [...], high move probability, and high value”. In fact, this selection criterion for the Monte Carlo tree search is essentially the equivalent of the UCB1 formula

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln s_p}{n_i}} \quad (5.5.1)$$

that we have already seen a few sections ago. Now the visit count plays the role of n_i (number of simulation for the considered node), the move (or prior) probability is essentially linked to s_p (the number of simulations run by the parent node) and the high action value is the equivalent of the win rate w_i/n_i . Therefore, the former two in the combination low/high

5 Adversarial search

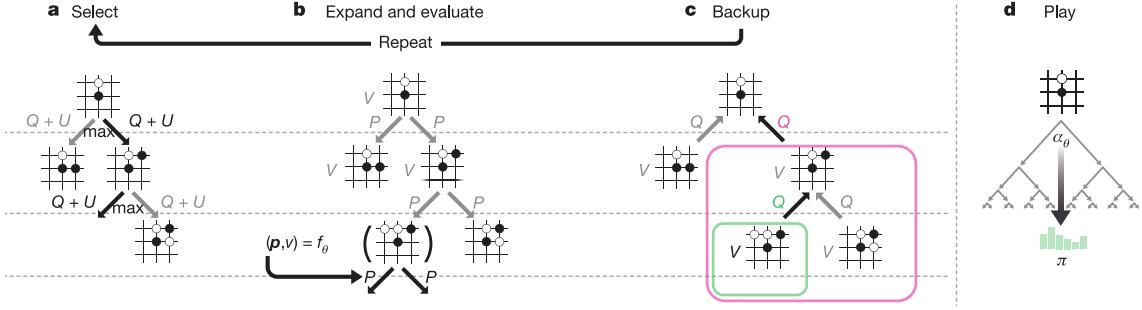


Figure 5.32

favor exploration over exploitation, while the latter instead promotes the exploitation term. The only differences that we have in this algorithm are just: (1) a slight variation in the functional form of the upper confidence bound; (2) we don't perform any random playout from the expanded node because these numbers are entirely computed by the neural network.

5.5.1.2 The training

As said before, the main ingredients behind AlphaZero are the DNNs and MCTS, and once combined together they are able to train the machine at a superhuman level. As usual the training procedure works by iteration and is implemented as follows ([14]):

The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move. First, the neural network is initialized to random weights θ_0 . At each subsequent iteration $i \geq 1$, games of self-play are generated (Fig. 5.33a). At each time-step t , an MCTS search $\pi_t = \alpha_{\theta_{i-1}}(s_t)$ is executed using the previous iteration of neural network $f_{\theta_{i-1}}$ and a move is played by sampling the search probabilities π_t . A game terminates at step T when both players pass [(this is referred to AlphaGo Zero)], when the search value drops below a resignation threshold or when the game exceeds a maximum length; the game is then scored to give a final reward of $r_T \in \{-1, 1\} [\dots]$. The data for each time-step t is stored as (s_t, π_t, z_t) , where $z_t = \pm r_T$ is the game winner from the perspective of the current player at step t . In parallel (Fig. 5.33b), new network parameters θ_i are trained from data (s, π, z) sampled uniformly among all time-steps of the last iteration(s) of self-play. The neural network $(p, v) = f_{\theta_i}(s)$ is adjusted to minimize the error between the predicted value v and the self-play winner z , and to maximize the similarity of the neural network move probabilities p to the search probabilities π . Specifically, the parameters θ are adjusted by gradient descent on a loss function l that sums over the mean-squared error and cross-entropy losses, respectively:

$$l = (z - v)^2 - \pi^T \log p + c \|\theta\|^2 \quad (5.5.2)$$

where c is a parameter controlling the level of ℓ_2 weight regularization (to prevent overfitting).

From how it is written it seems a lot convoluted, but in reality it is nothing so prohibitive for our knowledge; simply every word must be carefully understood. Basically, we begin with a board position s_1 and we complete a Monte Carlo tree search as depicted in Fig. 5.32

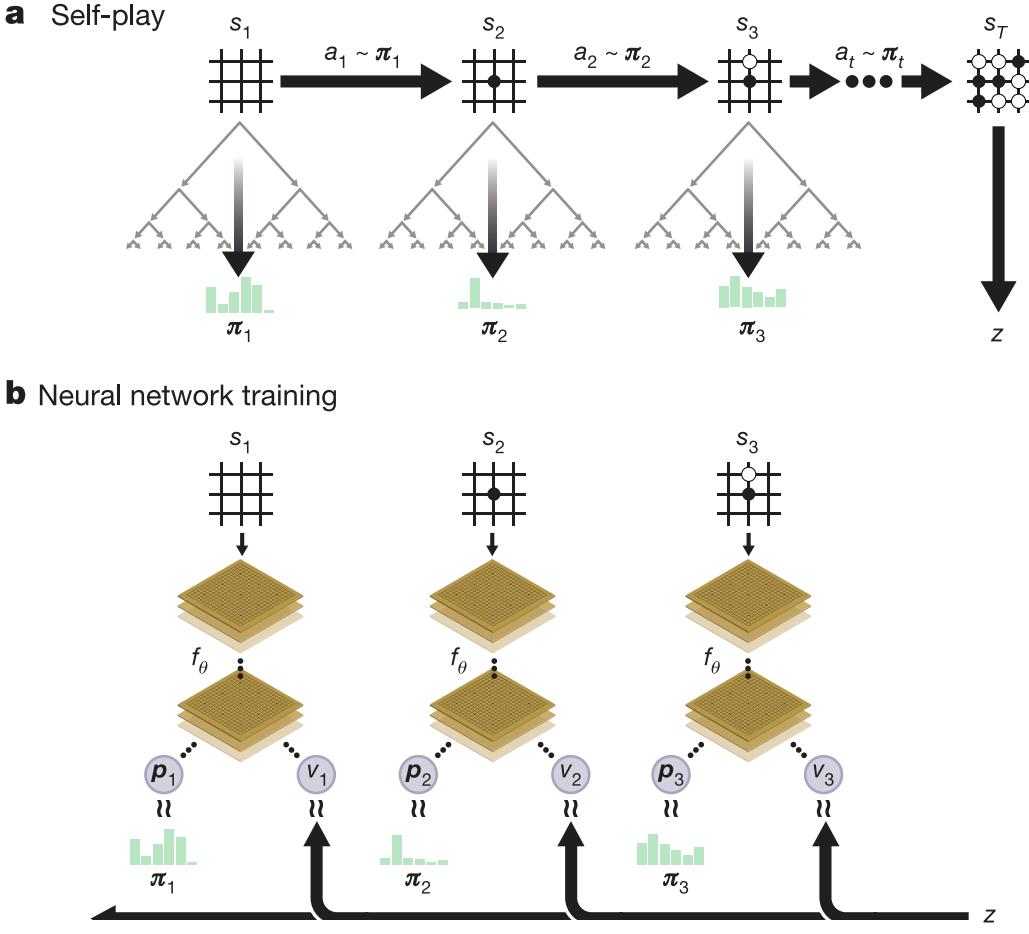


Figure 5.33

using the latest neural network \mathbf{f}_{θ_i} (initially random). The ultimate result of this search will be a vector π_1 of probabilities recommending moves to play. Then, from the root node s_1 we select a move a_1 according to the search probabilities computed before, that is we sample the move from π_1 , $a_1 \sim \pi_1$, and we perform it. In this way we definitely change our original location into a new one, namely s_2 ; therefore the previous one is momentarily forgotten, while s_2 will constitute the new root node for the next search. At this point, from this new root node we execute a new MCTS, still using the same neural network \mathbf{f}_{θ_i} . As before, the result will be a vector π_2 of probabilities for the possible actions. So we sample again a possible move from it, $a_2 \sim \pi_2$, and we end up by landing on the state s_3 . As you will have understood, this state will be involved into another MCTS, which in turn will provide new actions to produce a new root, and so on and so forth. Thus, step after step we are progressively increasing the index t by 1, keeping i fixed (indeed we have always used the same network \mathbf{f}_{θ_i}). This means that in this first phase the program has essentially *played a game against itself*, since it has performed a sequence of moves that have allowed it to explore one position at a time, s_1, \dots, s_T . In other words, during each turn t of a game, we perform a fixed number of MCTS simulations starting from the current $s_{\text{root}} = s_t$ state and we pick a move a_t by sampling from the improved policy π_t ($a_t \sim \pi_t$). Naturally we cannot leave the game continuing till the very end, both because it is a waste of resources and because it can take a very long time (e.g., a game of Go lasts on average 200 moves). Therefore, we decide to *truncate* the current game at step T and we just score the terminal position s_T according to the rules of the game to compute the winner z : -1 for a loss, 0

for a draw, and +1 for a win.

This preliminary phase is the so-called “self-play phase”; once completed you enter more explicitly in the actual “training phase”. In this phase the tuples (s_t, π_t, z_t) obtained before, where z_t is the final perspective of the current player at step t ($z_t = \pm z_T$), are sent to neural network in order to be processed, i.e. to compute the corresponding values (\mathbf{p}_t, v_t) . Therefore, by following a standard back-propagation, the neural network parameters Θ are updated to maximize the similarity of the policy vector \mathbf{p}_t to the search probabilities π_t , and to minimize the error between the predicted winner v_t and the game winner z_t . So this is done during gradient descent by minimizing the following loss function:

$$L = \sum_t \left[(z_t - v_t)^2 - \pi_t^\top \log \mathbf{p}_t + c \|\Theta\|^2 \right] \quad (5.5.3)$$

where the first term is the mean squared error (comparing scalar values), the second is the cross entropy loss (comparing probability distributions) and the third is a regularization term included in order to prevent overfitting. Finally, the new parameters are used in the next iteration of self-play ($i \rightarrow i + 1$) as in Fig. 5.33a.

The parameters Θ of the deep neural network in AlphaZero are trained by reinforcement learning from self-play games. These search probabilities π usually select much stronger moves than the raw move probabilities \mathbf{p} of the neural network $\mathbf{f}_\Theta(s)$; MCTS may therefore be viewed as a powerful policy improvement operator.

In summary, AlphaZero training is split into two independent parts: *network training* and *self-play* data generation. These two parts only communicate by transferring the latest network checkpoint from the training to the self-play, and the finished games from the self-play to the training. Each self-play job is independent of all others; it takes the latest network snapshot, produces a game and makes it available to the training job by writing it to a shared replay buffer.

As an estimate of the orders of magnitude we report what has been said in [15]:

We trained separate instances of AlphaZero for chess, shogi, and Go. Training proceeded for 700 000 steps (in mini-batches of 4096 training positions) starting from randomly initialized parameters. [...] Training lasted for approximately 9 hours in chess, 12 hours in shogi, and 13 days in Go.

Figure 5.34 shows the performance of AlphaZero during self-play reinforcement learning, as a function of training steps, on an Elo scale. In chess, AlphaZero first outperformed Stockfish after just 4 hours (300 000 steps); in shogi, AlphaZero first outperformed Elmo after 2 hours (110 000 steps); and in Go, AlphaZero first outperformed AlphaGo Lee after 30 hours (74 000 steps).

5.5.1.3 Remarks

The AlphaZero algorithm described in the Science paper [15] differs from the original AlphaGo Zero algorithm [14] in several respects. One of these is the following.

The rules of Go are invariant to rotation and reflection. This fact was exploited in AlphaGo and AlphaGo Zero in two ways. First, training data were augmented by generating eight symmetries for each position. Second, during MCTS, board positions were transformed by using a randomly selected rotation or reflection before

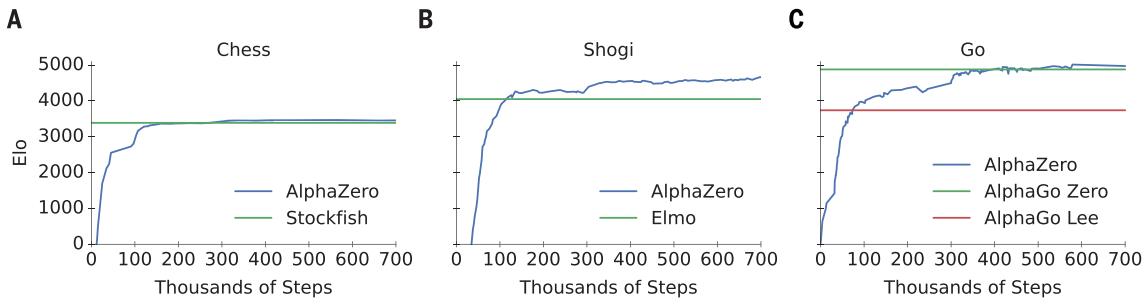


Figure 5.34

being evaluated by the neural network, so that the Monte Carlo evaluation was averaged over different biases. To accommodate a broader class of games, AlphaZero does not assume symmetry; the rules of chess and shogi are asymmetric (e.g., pawns only move forward, and castling is different on kingside and queenside). AlphaZero does not augment the training data and does not transform the board position during MCTS.

In short, this means that in AlphaGo and AlphaGo Zero we can exploit these two discrete symmetries (parity and rotation) of the Go game to augment the number of data, thus strengthening the information that a board position can appear in different ways, and to increase the accuracy of the neural network, by averaging over more positions. Clearly this extra information cannot be implemented in AlphaZero because chess and shogi are not invariant.

AlphaGo Zero uses a *convolutional neural network* architecture that is particularly well-suited to Go: the rules of the game are translationally invariant, the board is large, and the game is simple (we just need to recognize connections between stones). So the input of the network does not need to be highly connected with the other layers, because eventually the most important initial correlations will re-emerge on a bigger scale. Here we can have a very basic structure that can be collected at the very simple level and processed at higher level. Therefore, the hierarchical structure of convolutional neural networks is exactly what we need. At low level it's sufficient to understand that two linked stones are safe, for instance, then the second layer could study more sophisticated patterns, and so on. In this way the network is particularly simplified because we are reducing the total number of weights. On the other hand, chess and shogi does not share this invariance and they are radically different from Go. Nonetheless, AlphaZero uses the same convolutional network architecture as AlphaGo Zero.

Finally:

The hyperparameters of AlphaGo Zero were tuned by Bayesian optimization. In AlphaZero, we reuse the same hyperparameters, algorithm settings, and network architecture for all games without game-specific tuning. The only exceptions are the exploration noise and the learning rate schedule.

Remember that the hyperparameters are not the parameters that we want to tune in the back-propagation, to be clear, but they are intrinsic of the learning algorithm we are using. They are saying that the same hyperparameters are fine for all games. And this is interesting, because it indicates that this type of machine they developed is rather universal.

5.5.2 Performances

We conclude this chapter by comparing the performances of AlphaZero with respect to other famous and specialized programs, like Stockfish for chess, Elmo for shogi and the previous version AlphaGo Zero for Go. Everything is summed up in Figure 5.35.

As we can see from box A, the superiority of AlphaZero with respect to Stockfish is overwhelming in the game of Go. If AlphaZero is playing white it wins 29.0 % of the times and draws 70.6 % of the times; only the remaining 0.4 % of the times it loses, probably because AlphaZero is implemented with a stochastic component, while Stockfish is implemented with deterministic tree search. It is interesting to notice the significant difference in playing white (which makes the first move) or black⁶. In fact, if AlphaZero plays black superiority is strongly smoothed, since almost all the games finish with a draw, but still it wins twice more frequently than Stockfish. The results also attracted perplexity from a methodological point of view, due to the experimental configuration and the different type of hardware used. For example, one of the original developers of Stockfish pointed out how the playing conditions were penalizing for Stockfish, which is not optimized for fixed-time play per move, instead taking advantage of having a fixed time per game and distributing the time to different degrees in the course of the moves as needed. Anyway, under standard conditions AlphaZero greatly outperforms Stockfish.

For shogi there is no competition at all and color doesn't seem to play such an important role. Instead, against AlphaGo Zero, the previous version of the program optimized for the game of Go, the outcome is more balanced, but AlphaZero is still slightly better. This is due to the fact that AG0 still relies on a bit amount of human knowledge strictly applied to the game.

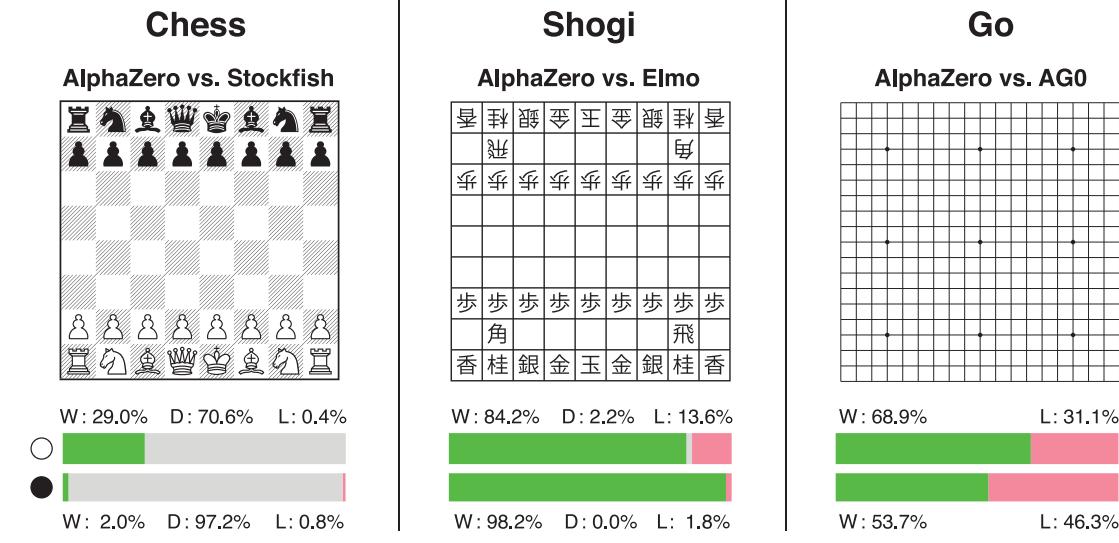
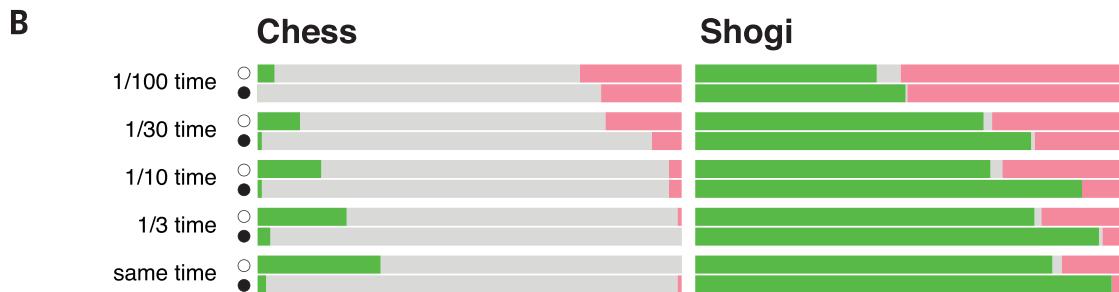
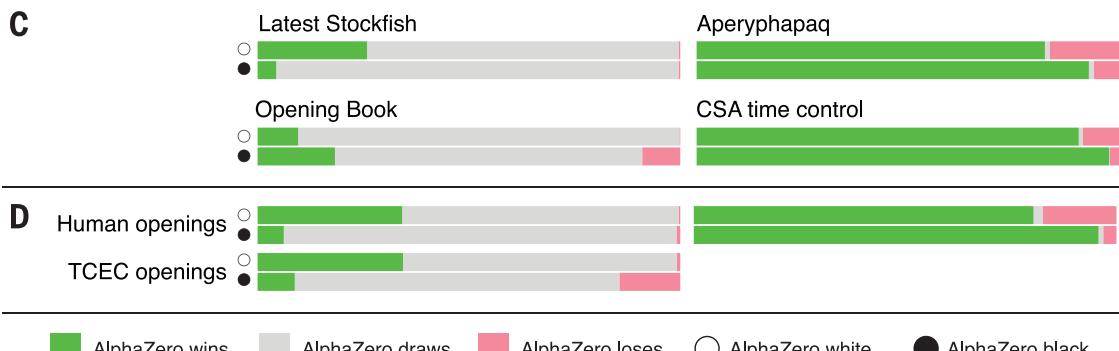
The big difference of AlphaZero with respect to traditional computer programs is evident if we consider the number of positions studied by the algorithm per second.

AlphaZero searches just 60 000 positions per second in chess and shogi, compared with 60 million for Stockfish and 25 million for Elmo. [...] AlphaZero may compensate for the lower number of evaluations by using its deep neural network to focus much more selectively on the most promising variations.

That's why, according to the authors of the paper, AlphaZero has a more humanlike approach to searching than Stockfish (humans can evaluate only a limited number of moves). Also, a traditional computer program is able to spot in the search tree even a very narrow path that can lead to the best move; if we want to go much deeper we just have to increase the velocity of the computer. For instance, a typical endgame position in which there are rooks or queens on the board is very difficult to evaluate for a human being (without specific books), since these pieces generate long-range interactions and have many degrees of freedom. But an algorithm like Stockfish is actually able to find the best move to perform. Instead, for AlphaZero is rather uncommon to follow forced lines: Monte Carlo tree search computes an average of many outcomes, and hence it can be quite struggling for him to spot the optimal move. In the end it manages to win the same, but it could take many more moves than a human with a good book of endgame techniques.

In box B is shown the scalability of AlphaZero with thinking time, that is how good it performs if the time at disposal for choosing a move is scaled down to a certain amount

⁶In chess, there is a general consensus among players and theorists that the player who makes the first move (White) has an inherent advantage. Since 1851, compiled statistics support this view; White consistently wins slightly more often than Black, usually scoring between 52–56 %.

A

B

C


■ AlphaZero wins ■ AlphaZero draws ■ AlphaZero loses ○ AlphaZero white ● AlphaZero black

Figure 5.35: Comparison with specialized programs. (A) Tournament evaluation of AlphaZero in chess, shogi, and Go in matches against, respectively, Stockfish, Elmo, and the previously published version of AlphaGo Zero (AG0) that was trained for 3 days. In the top bar, AlphaZero plays white; in the bottom bar, AlphaZero plays black. Each bar shows the results from AlphaZero's perspective: win (W; green), draw (D; gray), or loss (L; red). (B) Scalability of AlphaZero with thinking time compared with Stockfish and Elmo. Stockfish and Elmo always receive full time (3 hours per game plus 15 s per move); time for AlphaZero is scaled down as indicated. (C) Extra evaluations of AlphaZero in chess against the most recent version of Stockfish at the time of writing and against Stockfish with a strong opening book. Extra evaluations of AlphaZero in shogi were carried out against another strong shogi program, Aperyqhapaq, at full time controls and against Elmo under 2017 CSA world championship time controls (10 min per game and 10 s per move). (D) Average result of chess matches starting from different opening positions, either common human positions (see also Fig. 5.36) or the 2016 TCEC world championship opening positions, and average result of shogi matches starting from common human positions (see also Fig. 5.36). CSA world championship games start from the initial board position.

5 Adversarial search

with respect to opponent's. Clearly, by reducing the thinking time, the performances of AlphaZero begin to weaken because we decrease the number of simulations onto which the algorithm is based. Stockfish finally overwhelms AlphaZero only if the latter is allowed just 1/100 of the time.

To verify the robustness of AlphaZero, the authors also played additional matches that started from common human openings (Fig. 5.36). AlphaZero defeated Stockfish in each opening, suggesting that AlphaZero has mastered a wide spectrum of chess play. The frequency plots in Fig. 5.36 and the time line show that common human openings were independently discovered and played frequently by AlphaZero during self-play training. Just by knowing the rule of the game and nothing else, AlphaZero discovered what humans had assimilated in centuries of history.

For example:

- The first case is called “English Opening”. It is a flank opening, which is among the most popular and successful of White’s twenty possible first moves. AlphaZero discovered it quite soon and plays it roughly 10 % of times, without ever forgetting it.
- The first move in the second column is called “Queen’s Gambit”. It is one of the oldest openings and is still commonly played today. It is traditionally described as a gambit because White appears to sacrifice the c-pawn; however, this could be considered a misnomer as Black cannot retain the pawn without incurring a disadvantage. AlphaZero discovers it immediately and plays it more and more often over time.
- The third move in the second column is an alternative of the “French Defense”. It has a reputation for solidity and resilience, although some lines can lead to sharp complications. Black’s position is often somewhat cramped in the early game. This has been played for centuries also at the highest level, but AlphaZero at a certain moment stop playing it, probably discovering it is not so good.
- The fifth move in the second column is called “Spanish Defense” or “Ruy Lopez”. Apparently, AlphaZero seems to like playing it.
- The last move in the first column is the “Caro–Kann Defense”. It is strongly played by AlphaZero, but after some time it stop considering it.



Figure 5.36: AlphaZero plays against (A) Stockfish in chess and (B) Elmo in shogi. In the left bar, AlphaZero plays white, starting from the given position; in the right bar, AlphaZero plays black. Each bar shows the results from AlphaZero’s perspective: win (green), draw (gray), or loss (red). The percentage frequency of self-play training games in which this opening was selected by AlphaZero is plotted against the duration of training, in hours.

Appendix

Programs and computer codes

In this conclusive part of the text we propose some codes taken from the book “Neural Networks: An Introduction” by B. Müller, J. Reinhardt, M. T. Strickland [6]. As said by the authors: “The programs in this book were developed on IBM AT and PS/2 personal computers under the DOS operating system. They should run without modifications on any personal computer compatible with this standard. [...] The code is written in the programming language C, which we chose for its versatility, portability, and widespread availability on personal computers”.

The purpose of these programs is to provide practical demonstrations of some of the main classes of neural networks. The reader is encouraged to ‘play’ with these programs: by varying the input data and the model parameters one can gain a familiarity with the models which is hard to acquire from theoretical ruminations alone. Of course, running prefabricated software is only of limited educational value. We therefore provide both the executable versions of the programs and the source codes and urge readers to their implement own modifications of the codes.

In the following chapters we will discuss the programs one by one. For ease of reference each chapter gives a summary of the corresponding network model as it was introduced in earlier parts of the book (where necessary new theoretical concepts are also discussed). This is followed by a brief description of the program and its input and output. For full details the program source code should be consulted.

These little programs are inserted in the attachments under form of `.rar` files. To open them double-click on the name of the program in the corresponding title section or open the attachment panel to the left into Adobe Acrobat Reader. We also put there the free and open-source DOS emulator called DOSBox  for Windows; however, you can find the most updated version at the following link: <https://www.dosbox.com/download.php?main=1>. At the same webpage you can also consult a brief tutorial on how this emulator works. Briefly, for our case we have to launch the emulator and then:

- Type ‘`MOUNT C C:\Codes_Folder`’ to set the local directory `C:\Codes_Folder` as the new drive (called `C`, for simplicity) for DOSBox. Within this folder there must be our little programs.
- After you’ve done this, you will be prompted with a `Z:>`. Now, just write what you wanted to call your new DOSBox drive, which as I said above, we called `C`. To navigate to that newly mounted drive just type in ‘`C:`’.
- Finally, change the directory to the specific one containing all the files for a certain program (e.g. ‘`CD ASSO_Folder`’) and run the executable of the program (e.g. ‘`ASSO.EXE`’).

APPENDIX A

ASSO: associative memory

This program implements an associative-memory network of the type advanced by Hopfield and Little. In accordance with the discussion of Sec. 3.2 on, the network consists of a set of N units s_i (bits) which can take on the values ± 1 . A set of p patterns σ^μ each containing N bits of information is to be memorized. This storage is distributed over the set of $N \times N$ real-valued synaptic coefficients w_{ij} mutually connecting the neurons. Memory recall proceeds by initially clamping the neurons to a starting pattern $s_i(0)$. Then the state of the network develops according to a relaxation dynamics in discrete time steps $s_i(0) \rightarrow s_i(1) \rightarrow \dots$. Whether a neuron flips its state depends on the strength of the “local field”

$$h_i(t) = \sum_{j=1}^N w_{ij} s_j(t) \quad (\text{A.1})$$

Several varieties of updating rules can be introduced. In a deterministic network the new state of the neuron is uniquely controlled by the local field derived from the previous time step

$$s_i(t+1) = \text{sgn} [h_i(t) - \vartheta_i] \quad (\text{A.2})$$

ϑ is a global threshold which is normally set equal to zero for balanced patterns (equal numbers of “on” and “off” neurons). In contrast to this rule, in a stochastic network only the probability for the new state is known, namely

$$\Pr[s_i(t+1) = \pm 1] = f[\pm(h_i(t) - \vartheta_i)] \quad (\text{A.3})$$

where $f(-x) = 1 - f(x)$. The probability function $f(x)$ usually is taken to be

$$f(x) = \frac{1}{1 + e^{-2\beta x}} \quad (\text{A.4})$$

where the parameter $\beta = 1/T$ has the interpretation of an inverse temperature.

Given the updating rule (A.2) or (A.3) one still has the choice of timing. The updating steps can be taken as follows.

- In *parallel*, i.e. all values s_i are updated “simultaneously” depending on the state of the other neurons $s_j(t)$ before the common updating step.
- *Sequentially* in the sense that the updating proceeds in a given order (e.g. with increasing value of the label i). The local field is calculated as a function of $s_j(t)$, $j < i$, and $s_j(t-1)$, $j \geq i$.

A ASSO: associative memory

- *Randomly*, so that the update proceeds without a fixed order and a given neuron is updated once per unit time interval only on the average

More interesting than the relaxation dynamics are the learning rules which determine the synaptic coefficients w_{ij} from the patterns σ_i^μ which are to be stored. The following alternatives are implemented in the program ASSO.

- (1) The original Hebb's rule

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^\mu \sigma_j^\mu \quad (\text{A.5})$$

which was discussed extensively in Sec. 3.2 and followings of this book.

- (2) The iterative learning rule of Diederich and Opper in the form

$$w_{ij} \rightarrow w'_{ij} = w_{ij} + \sum_{\mu=1}^p \delta w_{ij}^\mu \quad (\text{A.6})$$

with the synaptic modifications

$$\delta w_{ij}^\mu = \frac{1}{N} \vartheta(1 - \tilde{\gamma}_i^\mu) \sigma_i^\mu \sigma_j^\mu \quad (\text{A.7})$$

and the stability coefficients

$$\tilde{\gamma}_i^\mu = \sigma_i^\mu \sum_{j=1}^N w_{ij} \sigma_j^\mu \quad (\text{A.8})$$

The aim of this and the following more involved learning rules is to ensure the stability of the sample patterns by iteratively increasing the stability coefficients.

- (3) The learning rule of Gardner and Forrest, which is slightly different from (A.7) in that the threshold for synaptic modification is determined locally for each neuron:

$$\delta w_{ij}^\mu = \frac{1}{N} \vartheta(\kappa - \gamma_i^\mu) \sigma_i^\mu \sigma_j^\mu \quad (\text{A.9})$$

where

$$\gamma_i^\mu = \frac{\tilde{\gamma}_i^\mu}{\|w_i\|} \quad (\text{A.10})$$

are the normalized stability coefficients. Here $\|w_i\|$ is the Euclidean norm

$$\|w_i\| = \sqrt{\sum_{k=1}^N w_{ik}^2} \quad (\text{A.11})$$

On the other hand, κ is a positive real constant which determines the ‘depth of imbedding’ of the patterns.

- (4) The prescription of Abbott and Kepler results from (A.7) by scaling of the synaptic increment with a smoothly varying coefficient f_i

$$f_i = \frac{1}{N} \|w_i\| \left(\kappa - \gamma_i^\mu + \delta + \sqrt{(\kappa - \gamma_i^\mu + \delta)^2 - \delta^2} \right) \quad (\text{A.12})$$

δ is a small parameter of the order $\delta \approx 0.01$.

- (5) The learning rule of Krauth and Mézard. The algorithm repeatedly visits all neurons and selectively enhances the pattern $\mu_0(i)$ which locally has the lowest stability. The updating rule is

$$\delta w_{ij} = \frac{1}{N} \sigma_i^{\mu_0(i)} \sigma_j^{\mu_0(i)} \quad (\text{A.13})$$

where $\mu_0(i)$ is that pattern for which the stability coefficient takes its minimum value

$$\gamma_i^{\mu_0(i)} = \min_{\nu=1,\dots,p} \gamma_i^\nu \equiv c_i \quad (\text{A.14})$$

The iteration terminates when all coefficients c_i are larger than a predetermined positive real constant C . This learning rule in principle is able to exhaust the maximum storage capacity of a Hopfield network, i.e. $\alpha = 2$.

- (6) Heteroassociation of patterns ($1 \rightarrow 2 \rightarrow 3 \rightarrow \dots$), which can be learned using the Diederich–Opper rule. Since this will be the subject of Appendix B we can skip the discussion here. We only mention that in the pattern-retrieval phase parallel updating has to be chosen if heteroassociation is to work.

A.1 Program description

Since the human brain is not particularly adept at analyzing long bit-strings the state of the network is graphically displayed as a two-dimensional rectangular field with dimensions $\text{nx} \times \text{ny}$ filled with dark or light squares. Keep in mind, however, that this representation of the data just serves to make a nice display. The Hopfield network knows nothing about geometrical properties or pattern shapes. The maximum dimensions are $\text{nx} = 13$ and $\text{ny} = 12$, which leads to a network of $N = 156$ neurons (or bits). The network can be trained both with ordered and with random patterns. The former are read from a data file **ASSO.PAT**. We have provided a data file which contains renditions of the 26 capital letters of the English alphabet, from A to Z.

The user is asked to specify the total number of patterns **patts** and the number of predefined patterns **plett** among them, which may be set to zero (Fig. A.1). In the case **patts** > **plett** the patterns read in from the file are complemented by the necessary number of random patterns. These have no “mean magnetization”, i.e. the probabilities for $\sigma_i^\mu = \pm 1$ are equal. Up to **patts** = 312 patterns are allowed, which is useful when studying the saturation properties of the iterative learning rules. However, for large values of **patts**, learning times tend to become intolerably large. Next the parameter **wsign** can be specified. It determines, whether the synaptic couplings can take on any real value or are forced to have a specified sign (forget about this for the moment). Finally the variable **rule** is read in and this selects the learning rule for the synaptic coefficients.

After these parameters have been specified the learning procedure is started (by pressing **<Escape>**). The program displays in rapid succession the patterns to be learned (Fig. A.2). Some of the learning models ask for the entry of an additional parameter which determines the strength of the local embedding of the patterns. During the learning phase a marker indicates at which of the neurons the synapses are modified at a given instant. After each iteration step the number of modified synapses change is displayed in an extra window to the right of the screen (in the case of the Krauth learning rule the lowest stability coefficient, is shown instead). The learning terminates if one of three conditions is met: (1) Convergence is reached, i.e. **change** = 0; (2) A timeout is reached after 1000 iteration steps; (3) The user runs out of patience and stops the iteration by pressing any key.

A ASSO: associative memory

The learning process can also be temporarily interrupted by pressing the key ‘a’ to analyze the synaptic matrix w_{ij} . The minimum, maximum, average, mean square values and other information of the coupling elements are given (Fig. A.3). Subsequently the user is given the choice of inspecting a histogram of the current distribution of coupling coefficients w_{ij} (Fig. A.4).

In addition the stability coefficients can be analyzed (Fig. A.5). If this option is chosen, after a slight delay a table shows the distribution of values of the normalized local stability coefficients γ_i^μ defined in (A.10). The distribution of the coefficients γ_i^μ contains valuable information on the performance of the associative memory. In particular, the negative coefficients (if any) correspond to those locations where a bit in one of the patterns is stored incorrectly. The distribution of stability coefficients is also displayed graphically as a histogram (Fig. A.6). The stability analysis can be performed repeatedly during the learning phase, if desired.

Having studied this information one proceeds to a further input panel asking for the following variables (Fig. A.7):

- **update:** The rule of updating the network, as discussed above. Random, sequential, or parallel dynamics can be chosen.
- **sstep:** Single-step mode.
- **temp:** The temperature T of the network.
- **theta:** The neural threshold parameter ϑ in (A.2). The same value is used for all neurons. For normal operation set $\vartheta = 0$.
- **main:** This determines how the starting pattern is obtained. A correct pattern will be distorted either by random flipping of bits or by masking, i.e. blanking out, one or several lines at the bottom of the pattern.

After leaving this panel the network is ready to operate as an associative memory. The following control parameters can be chosen:

- **s:** Search for a pattern.
- **p:** Change the operating parameters as described above.
- **m:** Modify the synaptic coefficients.
- **e:** Exit the program.

If one presses the key ‘s’ , the program asks for the starting pattern, which for convenience is designated by a letter (even if you have defined your own type of pattern or if random patterns are learned). Thus at most the first 25 patterns can be chosen for searching, even if **patts** is larger. The last required input depends on the previous choice of **main**. Either the number of blanked-out lines **blank** or the fraction of randomly flipped bits noise has to be entered. Note the definition of **noise**: it is not the fraction of bits which are inverted. Rather, the algorithm visits all neurons and selects each of them with the probability **noise**. The state of a selected neuron is then determined by ‘tossing a coin’. In this way even the maximum value **noise** = 1 will lead to a pattern in which about half of the bits are correct but there is no correlation between the original and the distorted pattern. Using noise directly to flip the neurons would lead to anticorrelated patterns. Since the Hopfield network is completely symmetric under spin flip $s_i \rightarrow -s_i$ no new information could be gained by allowing such starting configurations.

Now the pattern-retrieval phase can begin (Fig. A.8). Three frames are displayed which contain (1) the distorted starting pattern $s_i(0)$, (2) the correct pattern σ_i^μ which is to be recognized, (3) the actual state of the network. By sweeping through the network the values $s_i(t)$ are rapidly updated according to the rule specified before. Alternatively, in the single-step mode (`sstep = 1`) after each updating sweep the program pauses until the return key is pressed. The updating cycles are stopped when the correct pattern has been recognized, i.e. $s_i(t)$ agrees with σ_i^μ . If the network fails to converge to the correct pattern the iteration can be interrupted manually.

To facilitate an analysis of the network performance two columns of numbers are displayed. These are the overlaps of the initial ($t = 0$) and of the actual state of the network with the set of stored patterns $m_\mu(t) = \frac{1}{N} \sum_{i=1}^N \sigma_i^\mu s_i(t)$. This overlap is related to the number of incorrect bits (the Hamming distance $H_\mu(t) = \frac{N}{2}(1 - m_\mu(t))$). The magnitude of this overlap will decide whether a configuration lies in the basin of attraction of a pattern μ .

Also displayed is the energy (or rather the Lyapunov function) $E(t) = -\frac{1}{2} \frac{1}{N} \sum_{i,j} w_{ij} s_i(t) s_j(t)$. In the case of the Hebbian learning rule, which leads to symmetric couplings w_{ij} , this quantity does not increase with time, i.e. the dynamics is characterized by a descent into a local energy minimum. For the other learning rules, however, the function $E(t)$ can move in both directions and has no special significance.

Having watched the iteration process you may again type `s` to confront the network with a new distorted starting pattern, type `p` to go back to the parameter entry panel, type `m` to modify the synapses or type `e` to quit. For more information on the ASSO program consult directly the book.

A.2 Snapshots

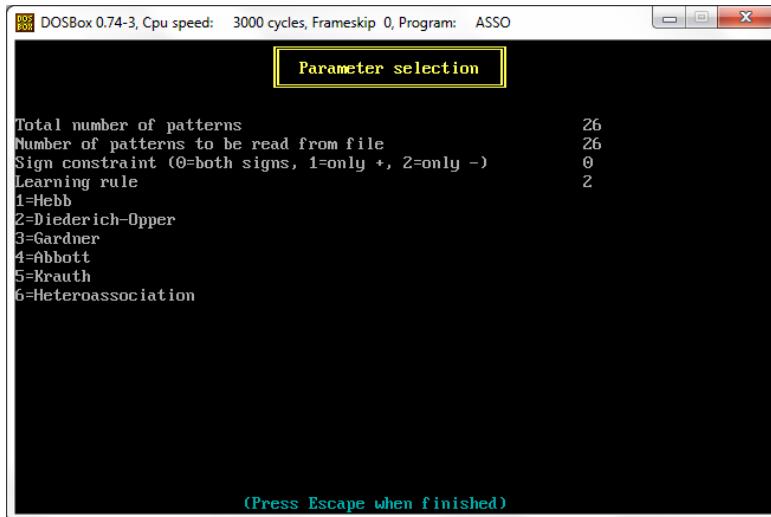


Figure A.1: Choice of the stored pattern and learning rule.

A ASSO: associative memory

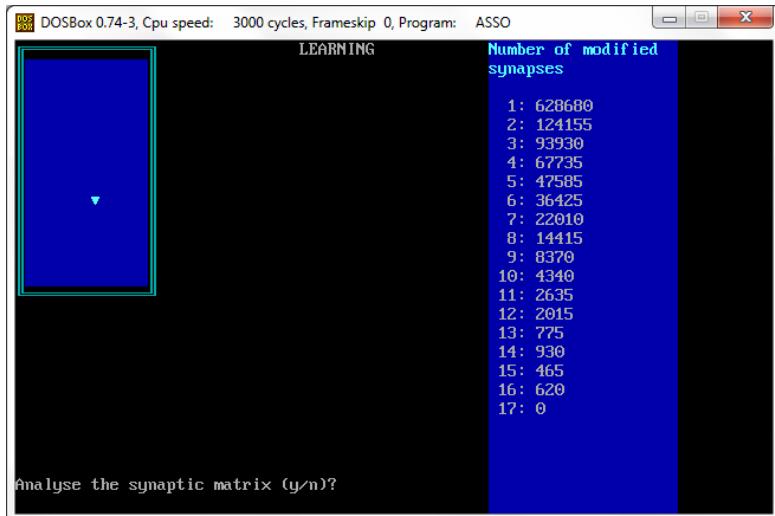


Figure A.2: Learning iterations.

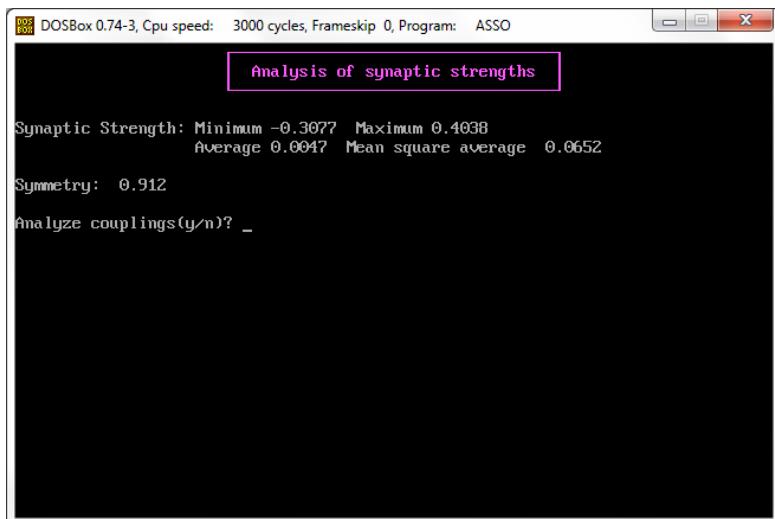


Figure A.3: Information on the synaptic strengths.

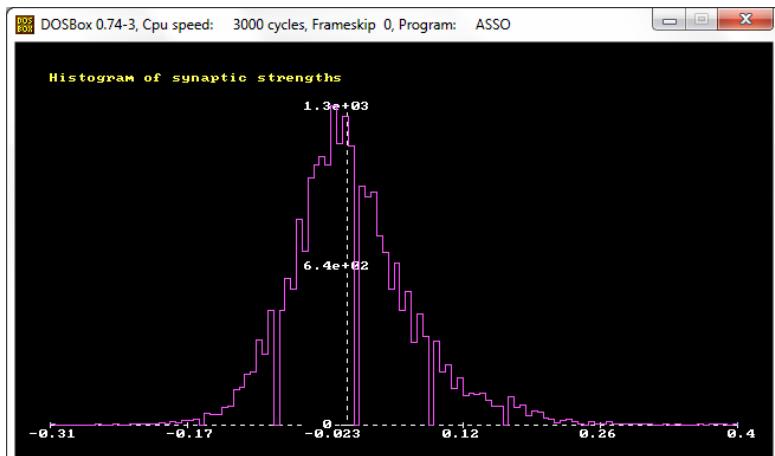


Figure A.4: Histogram of the synaptic strengths.

A.2 Snapshots

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: ASSO

Analyze stabilities(y/n)? y											
-10.00	0	-6.00	0	-2.00	0	2.00	293	6.00	79	10.00	0
-9.80	0	-5.80	0	-1.80	0	2.20	217	6.20	50	10.20	0
-9.60	0	-5.60	0	-1.60	0	2.40	210	6.40	81	10.40	0
-9.40	0	-5.40	0	-1.40	0	2.60	173	6.60	49	10.60	0
-9.20	0	-5.20	0	-1.20	0	2.80	169	6.80	39	10.80	0
-9.00	0	-5.00	0	-1.00	0	3.00	128	7.00	37	11.00	0
-8.80	0	-4.80	0	-0.80	0	3.20	143	7.20	43	11.20	0
-8.60	0	-4.60	0	-0.60	0	3.40	102	7.40	26	11.40	0
-8.40	0	-4.40	0	-0.40	0	3.60	94	7.60	36	11.60	0
-8.20	0	-4.20	0	-0.20	0	3.80	111	7.80	27	11.80	0
-8.00	0	-4.00	0	0.00	0	4.00	151	8.00	13	12.00	0
-7.80	0	-3.80	0	0.20	0	4.20	119	8.20	6	12.20	0
-7.60	0	-3.60	0	0.40	0	4.40	88	8.40	15	12.40	0
-7.40	0	-3.40	0	0.60	0	4.60	112	8.60	9	12.60	0
-7.20	0	-3.20	0	0.80	3	4.80	99	8.80	7	12.80	0
-7.00	0	-3.00	0	1.00	24	5.00	123	9.00	2	13.00	0
-6.80	0	-2.80	0	1.20	135	5.20	72	9.20	10	13.20	0
-6.60	0	-2.60	0	1.40	249	5.40	84	9.40	1	13.40	0
-6.40	0	-2.40	0	1.60	262	5.60	77	9.60	0	13.60	0
-6.20	0	-2.20	0	1.80	219	5.80	77	9.80	0	13.80	0
Stability coefficient gamma: Min. 0.939 Max. 9.569 Aver. 3.608											
Press any key!											

Figure A.5: Stability coefficients.

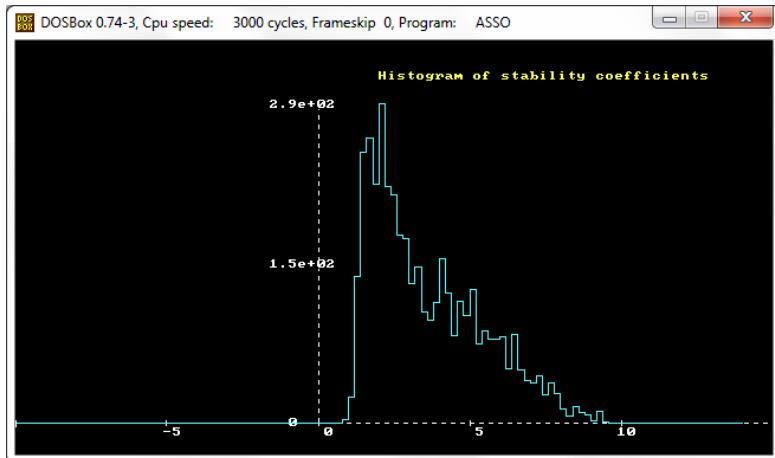


Figure A.6: Histogram of the stability coefficients.

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: ASSO

Parameter selection	
How to iterate? 0:random 1:sequential 2: parallel	1
Sweep network in single-step mode (0=no,1=yes)	0
Temperature	0.000000
Threshold potential	0.000000
Start with noisy pattern (1) or incomplete pattern (2)	1
(Press Escape when finished)	

Figure A.7: Testing the neural network: choice of the parameters.

A ASSO: associative memory

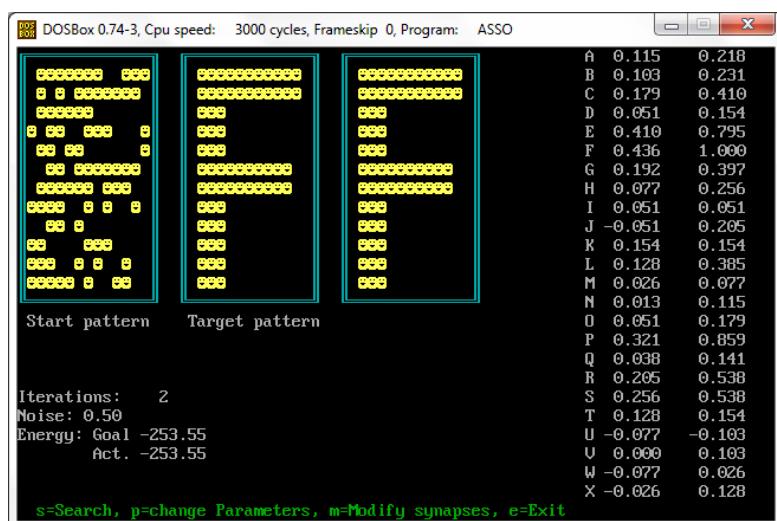


Figure A.8: Results of the execution.

APPENDIX B

ASSCOUNT: associative memory for time sequences

An obvious way to teach a Hopfield-type network to reproduce time sequences instead of stationary patterns is to introduce synaptic coefficients with an intrinsic time dependence as discussed in Sec. 3.2.5. Thus instead of the synaptic matrix w_{ij} , which acts instantaneously to generate the local field defined in (A.1), one may introduce a collection of synapses w_{ij}^τ acting with a distribution of characteristic time delays of magnitude τ . The local field, which according to (A.2) determines the probability for the updated neuron state $s_i(t+1)$, can be replaced by a generalized convolution in time.

$$h_i(t) = \sum_{\tau=0}^{\tau_{\max}} \sum_{j=1}^N \lambda^\tau w_{ij}^\tau s_j(t-\tau) \quad (\text{B.1})$$

Here the coefficients λ^τ determine the relative strength of the various types of synaptic couplings. The ordinary Hopfield model corresponds to the case $\lambda^0 = 1$, $\lambda^\tau = 0$ for $\tau \geq 1$. The couplings in (B.1) can give rise to a complex time evolution of the network. In the simplest nontrivial case just two types of synapse are present. We study a network connected through prompt synapses w_{ij} and slow synapses w'_{ij} of relative strength λ with the fixed time delay τ . The local field reads (cf. Sec. 3.2.5)

$$h_i(t) = \sum_{j=1}^N w_{ij} s_j(t) + \lambda \sum_{j=1}^N w'_{ij} s_j(t-\tau) \quad (\text{B.2})$$

Such a network can be made to go through an ordered sequence of patterns $1 \rightarrow 2 \rightarrow 3 \rightarrow \dots$. To this end the couplings w'_{ij} have to provoke the transition between successive patterns in the series. This is achieved by a very simple modification of the learning rules. For example, the Hebb rule (A.5) for heteroassociation of the pattern $\sigma^{\mu+1}$ given the pattern σ^μ as an input reads

$$w'_{ij} = \frac{1}{N} \sum_{\mu=1}^p \sigma_i^{\mu+1} \sigma_j^\mu \quad (\text{B.3})$$

The improved learning rules can be modified in the same way by replacing σ_i^μ by $\sigma_i^{\mu+1}$ in the updating rule for δw_{ij}^μ . The instantaneous synapses w_{ij} are used to stabilize the patterns; thus they are trained in the ordinary way for the task of autoassociation. If the coefficient λ is large enough, the network will act as a freely running counter. The state of

the network will switch from one pattern to the next after τ clock ticks when the delayed coupling starts to ‘pull in the direction’ of the new pattern.

A second interesting mode of operation of this network arises if the parameter λ is a little too small to induce transitions by itself. Then it takes an external disturbance to destabilize the old pattern. Once the network is driven out of the basin of attraction of the last pattern into some unstable region the delayed coupling w'_{ij} (which does not yet feel the disturbance) provides a driving force toward the associated next pattern. Thus the network performs the task of counting the number of externally entered signal pulses. Such a pulse can be implemented by adding a spike of random noise with amplitude ρ to the local field

$$h_i(t) = \sum_{j=1}^N w_{ij}s_j(t) + \lambda \sum_{j=1}^N w'_{ij}s_j(t - \tau) + \rho \xi_i \delta_{t,t_0} \quad (\text{B.4})$$

where ξ_i is a random variable taking on values ± 1 .

B.1 Program description

The program **ASSCOUNT.C** is closely related to **ASSO.C** so that much of the discussion of the last section applies here also. We assume that the reader is already familiar with **ASSO.C**. Again the total number of patterns **patts** and the number of predefined patterns **plett** to be learned must be specified. For a difference, the ordered patterns take the shape of the ten numerals 0, ..., 9 which are provided on the input file **ASSCOUNT.PAT**. The data format is the same as in the file **ASSO.PAT** so that by copying this file to the new name it is possible also to use the sequence of 26 letters instead of the numerals as patterns for counting.

Since our interest now is no longer focused on the learning process, only two choices are provided: Hebb’s rule and the iterative scheme of Diederich and Opper. Learning takes about twice as much time since the synaptic matrices w_{ij} and w'_{ij} have to be learned separately. The learned association of patterns is cyclical, i.e. the last pattern in the series, σ_i^p , is followed again by σ_i^1 .

After learning is finished you are asked to specify the following set of parameters:

- **update**: The rule of updating the network which can be random or sequential.
- **sstep**: Single-step mode.
- **temp**: The operating temperature T of the network.
- **tau**: The time delay of the couplings w' , an integer in the range [1, 10].
- **lambda**: The relative strength of the delayed compared to the prompt couplings.
- **rho**: The amplitude of the counting signal as defined in (B.4).

The iteration of the network is started by providing the starting pattern for which a pixel representation of one of the numbers 0, ..., 9 can be chosen (of course, if you have changed the input file **ASSCOUNT.PAT** the patterns may look very different). You can start with a distorted version of this pattern, specifying a value **noise** $\neq 0$. The network is updated for 10 000 sweeps unless the iteration is interrupted manually. As in **ASSO.C** the running parameters can be modified and the iteration restarted. The external signal for triggering the counter is entered by pressing **<SPACE>**.

To have the delayed network configuration $s_i(t - \tau)$ available at any instant t , all the intermediate values $s_i(t - 1), \dots, s_i(t - \tau)$ have to be stored. To avoid unnecessary data

shuffling these values are stored in a cyclical buffer $s0[t][i]$ of length τ_{au} . Two pointers t_{in} and t_{out} specify the locations where a configuration is stored and read out. These pointers are incremented after each sweep of the network using modulo τ arithmetic to account for the cyclical nature of the buffer.

B.2 Numerical experiments

You might start with the set of 10 numerals (which corresponds to a modest memory loading factor $\alpha = 0.064$), using the learning rule for correlated patterns. At temperature $T = 0$, experiment with the parameter λ . Essentially, different operating modes of the network can be observed in the three distinct regions. $\lambda > \lambda_2$: freely running counter; $\lambda_1 < \lambda < \lambda_2$: counter for input signals. $\lambda < \lambda_1$: stable associative network. However, the transition between these parameter regions is soft. For $\lambda > 2$ the network evolves like a perfect clock, i.e. each counting step has a length equal to the synaptic delay time τ , since a jump of the action of the second term in (B.4) immediately induces a transition to the next pattern. When reducing the value of λ the transition from one pattern to the next often needs several sweeps and the clock runs slow.

Figure B.1 shows an example of the spontaneously counting network with a marginal value of λ . For $\tau = 1$ the network has not enough time to settle at the attractors; it will operate with distorted and superimposed patterns. A further reduction of λ will eventually cause the network to ‘get stuck’ at one of the attractors with high stability (for our patterns the first of them happens to be the number ‘8’). At $\lambda = 0.4$ all the patterns are stable against spontaneous transitions. Now we are in the region where the network can serve as a perfect counter for external signals provided that their amplitude ρ is large enough. The theoretical expectation for the required amplitude is $\rho > 1 - \lambda$ which is very roughly confirmed by our numerical experiment. Finally, below $\lambda = 0.2$ the strength of the delayed synapses is too small to induce the correct transition even for counting pulses of arbitrary strength.

The discussion so far referred to the fully deterministic case, $T = 0$. Introducing a finite temperature has a profound influence on the network in the ‘chime counting’ mode. Since the temperature-induced noise tends to destabilize a condensed pattern it aids the action of the delayed synapses. As soon as a sufficiently large fluctuation occurs, the state of the network will make a transition to the next pattern. E.g., take the parameters $\lambda = 0.4$, $\tau = 5$, which at zero temperature lead to a stationary network. At finite temperatures the network starts to count spontaneously! This effect can even serve as a ‘thermometer’ since the cycle time is sensitive to the temperature (a rough estimate: $T = 0.5$ gives a cycle length of 65; $T = 0.15$ gives a cycle length of 120). Of course, at very high temperatures the network no longer reproduces any of the learned patterns while near $T = 0$ the counting is frozen in long-lived metastable states.

B.3 Snapshots

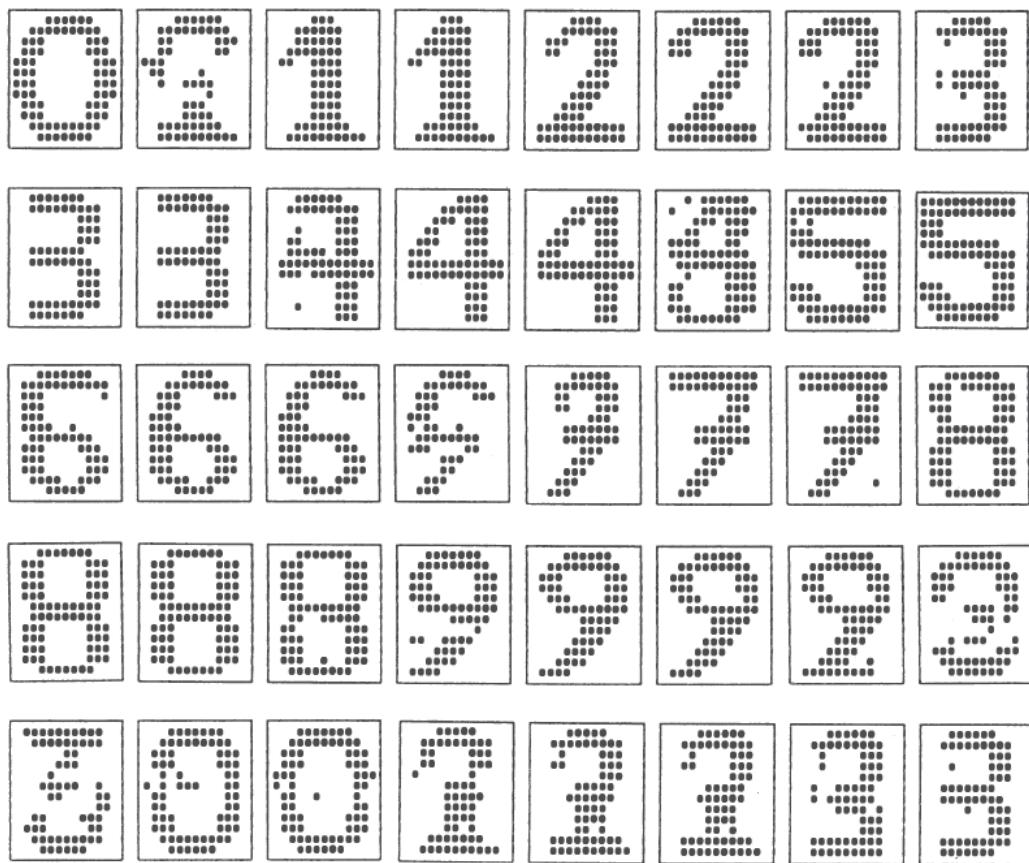


Figure B.1: The time development of a spontaneously counting network trained with the numeral patterns $0, \dots, 9$. The parameters are $\lambda = 0.7$, $\tau = 2$.

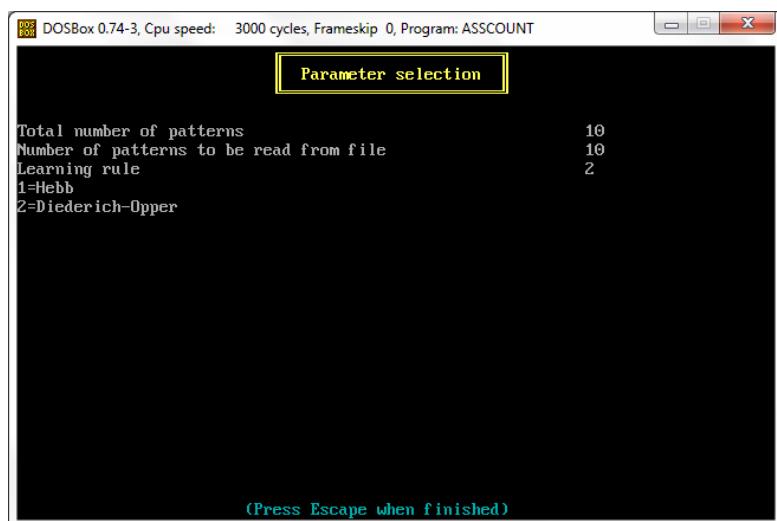


Figure B.2: Selection of patterns and learning rule.

B.3 Snapshots

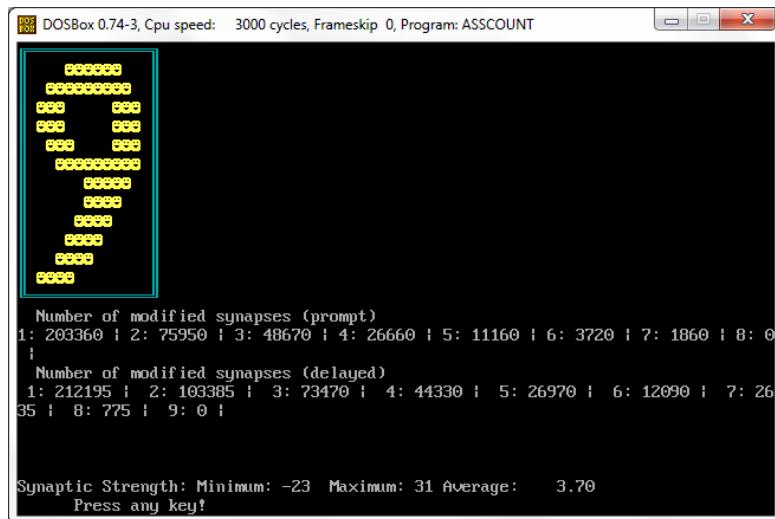


Figure B.3: Learning process.

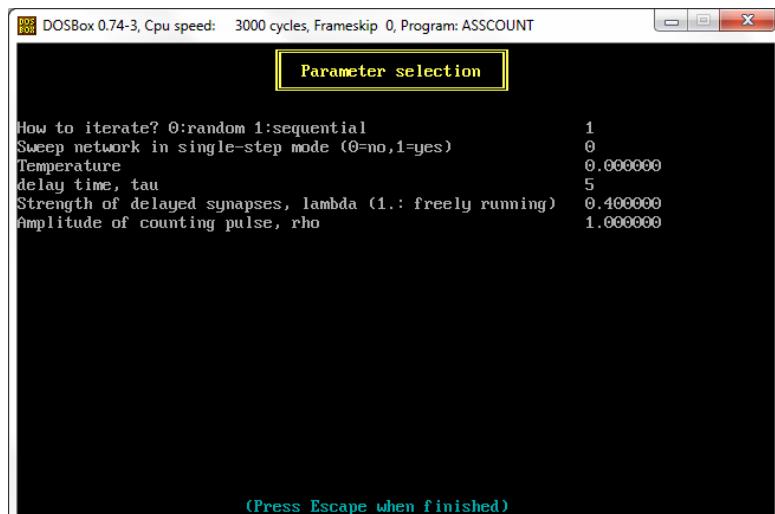


Figure B.4: Parameters selection.

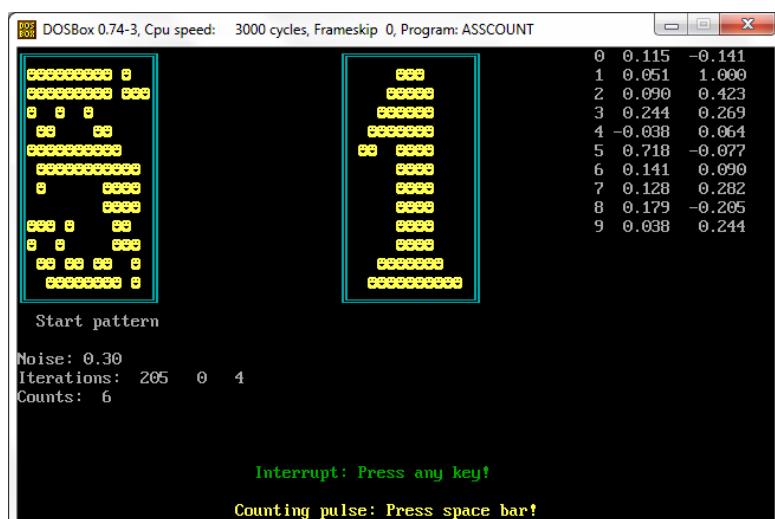


Figure B.5: Iteration of the network.

APPENDIX C

PERBOOL: learning Boolean functions with back-propagation

This program illustrates the function of a feed-forward layered neural network as described in Sections 3.3–3.4. Its task is to learn an arbitrary Boolean function of a small number of Boolean variables. This is achieved by a three-layer feed-forward network which is trained by a gradient-descent method, i.e. by the rule of error back-propagation. The network consists of an input layer σ_k , $1 \leq k \leq N_{\text{in}}$, a hidden layer s_j , $1 \leq j \leq N_{\text{hid}}$, and an output layer S_i , $1 \leq i \leq N_{\text{out}}$. The synaptic connections are \bar{w}_{jk} from the input to the hidden layer, and w_{ij} from the hidden to the output layer. In addition there are activation thresholds $\bar{\vartheta}_j$ and ϑ_i . The neurons have two activation values chosen as -1 and $+1$ (the transformation to the values 0 and 1 commonly used to represent binary numbers is trivial).

When an input pattern is fed into the network by clamping the neurons σ_k to represent an input pattern, the activation of the intermediate and output layers is determined according to the rule (3.4.11–3.4.14), namely

$$s_j = f(\bar{h}_j), \quad \bar{h}_j = \sum_{k=1}^{N_{\text{in}}} \bar{w}_{jk} \sigma_k - \bar{\vartheta}_j \quad (\text{C.1})$$

$$S_i = f(h_i), \quad h_i = \sum_{j=1}^{N_{\text{hid}}} w_{ij} s_j - \vartheta_i \quad (\text{C.2})$$

The activation function $f(x)$ for discrete two-state neurons corresponds to the step function $f(x) = \text{sgn } x$. During the learning phase, however, this is replaced by the continuous “sigmoidal” function

$$f(x) = \tanh(\beta x) \quad (\text{C.3})$$

since gradient learning is possible only if $f(x)$ is a smooth function with a finite derivative, which in our case is $f'(x) = \beta \operatorname{sech}^2(\beta x)$. At the end of the training phase the limit $\beta \rightarrow \infty$ leading to a discontinuous activation function is taken. As said in Sec. 3.3, if one wishes to work with two-state neurons throughout the training process, one can simply replace the derivative function $f'(x)$ by an averaged constant, which can be absorbed in the factor ε (see below).

The rule for error back-propagation minimizing the squared deviation of the network

output from the target values according to (3.4.15–3.4.20) leads to the synaptic corrections

$$\delta w_{ij} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu s_j^\mu, \quad \delta \vartheta_i = -\varepsilon \sum_{\mu=1}^p \Delta_i^\mu, \quad \Delta_i^\mu = [\zeta_i^\mu - f(h_i^\mu)] f'(h_i^\mu) \quad (\text{C.4})$$

$$\delta \bar{w}_{jk} = \varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu \bar{s}_k^\mu, \quad \delta \bar{\vartheta}_j = -\varepsilon \sum_{\mu=1}^p \bar{\Delta}_j^\mu, \quad \bar{\Delta}_j^\mu = \left(\sum_{i=1}^{N_{\text{out}}} \Delta_i^\mu w_{ij} \right) f'(\bar{h}_j^\mu) \quad (\text{C.5})$$

A simple modification of the learning procedure consists in the replacement of (C.4) by

$$\Delta_i^\mu = \beta [\zeta_i^\mu - f(h_i^\mu)] \quad (\text{C.6})$$

This change has the potential to speed up the convergence: the weight adjustments get increased in the “saturation region” where the value of h_i^μ is large and the factor $f'(h_i^\mu)$ approaches zero. For small values of h_i^μ (C.6) agrees with (C.4).

The learning consists of consecutive “training epochs” in which the full set of patterns σ_k^μ to be learned is presented to the network. The synaptic and threshold corrections are accumulated and applied at the end of an epoch. As a variation of this “batch learning” algorithm one can also update the network parameters after each single presentation of a pattern. This ‘online learning’ may, but is not guaranteed to, improve the learning speed.

An important modification which has the potential to improve the stability of the learning process consists in the introduction of a kind of hysteresis effect or “momentum”. In regions of the parameter space where the error surface is strongly curved the gradient terms Δ can become very large. Unless the parameter ε is chosen to be inordinately small, the synaptic corrections δw_{ij} will then tend to overshoot the true position of the minimum. This will lead to oscillations which can slow down or even foil the convergence of the algorithm. To avoid this problem the correction term can be given a memory in such a way that it will no longer be subject to abrupt changes. This is achieved by the prescription

$$\delta w_{ij}^{(n)} = \varepsilon \sum_{\mu=1}^p \Delta_i^\mu s_j^\mu + \alpha \delta w_{ij}^{(n-1)} \quad (\text{C.7})$$

where the index n denotes the number of the training epoch. In the language of numerical analysis this is called a *relaxation* procedure. If α is chosen fairly large, the search in the parameter space will be determined by the gradient accumulated over several epochs, which has a stabilizing effect. Unfortunately there is no general criterion how the gradient parameter ε and the momentum parameter α are to be chosen. The optimal values depend on the problem to be learned.

C.1 Program description

The program PERBOOL.C implements a three-layer network with dimensions $\text{nin} \leq 5$, $\text{nhid} \leq 100$, $\text{nout} \leq 5$. The desired values can be entered through an input panel which comes up immediately after starting the program.

The **nin** Boolean input variables can take on $\text{nbin} = 2^{\text{nin}}$ different values. The network is trained with a set of **patts** = nbin output patterns, i.e. the output is completely prescribed for each possible input combination¹. Even in the “scalar” case, i.e. for a single

¹To study the issue of generalization one would train the network with a subset of all possible inputs, i.e. use a smaller value of **patts**.

output variable, `nout` = 1, there is an embarrassing number of $2^{2^{\text{nin}}}$ possible Boolean functions. To define an arbitrary Boolean function we have to select a set of `nout` bit strings having the length `nin`.

These bit strings are represented in the program by the long integer variables `key[i]`, which conveniently can hold just 32 bits. Each bit specifies the desired response of the corresponding output neuron to that combination of input values which is encoded by the binary value of the position of the bit within the variable `key`. Since this may sound confusing, let us look at the familiar example of the XOR function. There are $2^2 = 4$ combinations of the two input variables and the output is defined by $00 \rightarrow 0$, $01 \rightarrow 1$, $10 \rightarrow 1$, $11 \rightarrow 0$. The input can be interpreted as a 2-bit binary number. Stepping through its value in descending order, the output is represented by the bit string 0110. Thus the XOR function is encoded by specifying the number `key` = 0110_{bin} = 6_{hex} = 6_{dec}.

The Boolean function to be learned can be defined for the program by giving the code number(s) `key[i]` for $i = 1, \dots, \text{nout}$ in hexadecimal representation, with the digits 0123456789ABCDEF. Since in general it is somewhat tedious to work out this binary code, the program allows for an alternative and more direct way of specifying the Boolean function. Having answered ‘m’ to the prompt, a menu is displayed in which the desired output bits can be directly entered by moving the cursor to the appropriate place, typing 0 or 1, and pressing <Return>. Note that for the case $\text{nout} > 1$ the complete output string ζ_i^μ , $i = 1, \dots, \text{nout}$, has to be entered: it is not possible to modify single bits. The program displays the hexadecimal value of the resulting coding variable(s) `key` which will facilitate the data entry if the same function is to be used repeatedly. The decoding of the input and output patterns takes place in the subroutine `patterns()` and produces the arrays `zotp` = ζ_i^μ and `sigp` = σ_k^μ .

Having been given the Boolean function to be learned, the program displays a second input panel which allows to specify details of the learning procedure.

- `epsilon`: The learning rate ε .
- `alpha`: The “momentum” constant α .
- `batch`: Selects the learning protocol (`online` = 0, `batch` = 1).
- `mod_cost`: Use of the modified cost function (see equation (C.6)).
- `beta`: The inverse steepness β of the activation function $f(x)$.

Subsequently the program enters the subroutine `learn()`. The synaptic coefficients and thresholds are initialized with random numbers in the range $-1 < \xi < +1$. Then one by one the input patterns are presented and a forward sweep is performed leading to output activations `sout[i]`. The difference of these values from the correct output `zout[i]` allows the calculation of modified network parameters according to the back-propagation algorithm. After each learning epoch the accumulated error $\sum_\mu \sum_i |\zeta_i^\mu - S_i^\mu|$ is shown. In addition the actual values of the network parameters are displayed on screen. Each line in the display shows the connections attached to one of the hidden neurons according to the following format:

$\bar{w}_{1,1}$	\dots	$\bar{w}_{1,\text{nin}}$	$\bar{\vartheta}_1$	$w_{1,1}$	\dots	$w_{\text{out},1}$
\vdots	\ddots	\vdots	\vdots	\vdots	\ddots	\vdots
$\bar{w}_{\text{nhid},1}$	\dots	$\bar{w}_{\text{nhid},\text{nin}}$	$\bar{\vartheta}_{\text{nhid}}$	$w_{1,\text{nhid}}$	\dots	$w_{\text{out},\text{nhid}}$
				ϑ_1	\dots	ϑ_{nout}

Table C.1: Format used for output of network configuration.

If the number of hidden neurons is large, only the first 20 lines of this table are displayed in order not to overflow the screen.

After any learning epoch the user can interrupt the learning by pressing the key ‘r’. This serves to evaluate the network performance using the current coupling coefficients. Displayed are the states of the input, intermediate, and output neurons, with a flagging of the wrong output bits. Note that this display employs the discrete activation function, so that the number of wrong bits may deviate from that contained in the error variable.

By pressing ‘p’ it is possible to view and modify the values of the parameters used by the back-propagation algorithm during the learning process.

Perhaps more illuminating—and at least more entertaining—than the numerical output is a graphical representation of the learning process. This mode is entered if the key ‘g’ is pressed during learning. Then as a function of the learning epoch a collection of $\text{patts} = 2^{\text{nin}}$ curves is shown. Each curve represents the activation of the (first) output neuron in response to one specific input pattern. Since in the learning phase the network operates with continuous functions $f(x)$, these curves move smoothly in the interval $[-1, +1]$. The abscissa extends over a range of 200 training epochs. If this number is exceeded, the display is erased and the curves reappear at the left border. By repeatedly pressing ‘g’ one can switch between graphics and text display.

If the learning converges to the correct solution, the program tries to make a smooth transition to the discrete (i.e. theta-function) operating mode by slowly increasing the value of β by a factor of 1.05 after each epoch. This sets in if the error variable falls below the value $1/\beta$ and continues until error reaches 10^{-3} .

After convergence has been reached or when the learning process is stopped by pressing the key ‘e’ the complete set of synaptic coefficients and thresholds is displayed and the final network performance is shown.

We should mention that initially the program operates in the ‘slow motion’ mode, inserting a delay of 0.5 second in each training epoch. By pressing the key **s** it is possible to switch to full speed.

C.2 Numerical experiments

The simplest example of a Boolean function with a nontrivial representation is the exclusive-OR problem (XOR) discussed in Sec. 3.3.1. This can be investigated by setting $\text{nin} = 2$, $\text{nout} = 1$, and $\text{key} = 6$. The back-propagation algorithm can solve this problem without much effort provided that the network configuration contains at least two hidden units, $\text{nhid} = 2$. Over a fairly wide range of parameters ε and α most learning runs lead to convergence to a valid solution. One typically might choose a gradient parameter $\varepsilon = 1$ and a momentum factor $\alpha = 0.9$. The steepness parameter β in principle is redundant and may be kept fixed at the value $\beta = 1$. Owing to the special form of the activation function $f(x)$, this parameter only enters as a multiplicative factor attached to the local fields h_i . Therefore a change of β can be offset by inversely rescaling all synaptic coefficients and

thresholds. If also the gradient parameter ε is rescaled (quadratically), the behavior of the network should be unchanged. In our program implementation this is not quite true since the initialization is performed with random numbers in the fixed interval $-1 < \xi < 1$. If a very small value of β is chosen, this has the same effect as an initialization with near-zero values, which adversely affects the convergence behavior.

Examining the performance of the network, one can notice that it finds a considerable variety of solutions to the XOR problem. Most solutions are essentially equivalent, differing by simultaneous sign changes of activation values s and coupling coefficients w which leave invariant the equations which describe the network. This is a reflection of the fact that with the help of negation operations the XOR function can be represented in various ways by using the de Morgan identity of Boolean algebra $\overline{A \wedge B} = \overline{A} \vee \overline{B}$. There are two distinct classes of representation of XOR where the output neuron acts either as an OR gate or as an AND gate, according to

$$A \oplus B = (A \vee B) \wedge \overline{A \wedge B} \quad (\text{C.8})$$

$$A \oplus B = (A \wedge \overline{B}) \vee (\overline{A} \wedge B) \quad (\text{C.9})$$

where \oplus is used to denote the exclusive-OR or OR function. An example of the former solution type was shown in Fig. 3.18. The following two tables show typical results of couplings and thresholds generated by the back-propagation algorithm. They are displayed in the format introduced in the last subsection

+1.657	+1.615	-1.968	+3.105
+2.536	+2.929	+3.155	-2.494
+2.306			

(a) Example of (C.8).

+2.059	-2.034	+2.368	+3.117
-2.432	+2.624	+2.518	+2.623
-2.153			

(b) Example of (C.9).

Table C.2

Starting from randomly initialized couplings, the learning algorithm often converges within about 20 iteration steps. However, it does not always find valid solutions of the XOR problem. In some instances it gets stuck at a local minimum, or the local fields are driven to large values where the derivative of the activation function $f'(h)$ is nearly zero, leading to exceedingly slow learning. Figure C.1 shows an example where the network has just managed to escape from such a situation after about 100 training epochs.

Going to problems of a higher dimension provides further insights. As the ‘benchmark’ problem we have predefined in the program PERBOOL the generalization of the XOR function to more than two variables. This function takes on the value 1 if there is an odd and 0 if there is an even number of 1s in the input. Thus it can also be called the ‘parity’ function; still another interpretation is ‘addition modulo 2’. Let us take the case `nin` = 5. Using the binary encoding of the Boolean function leads to `key` = 10010110011010010110100110010110_{bin} = 96696996_{hex}. The parity problem can be learned if the number of hidden neurons is at least equal to the input dimension, `nhid` = `nin`. However, even for `nhid` = `nin` – 1 the network can produce solutions while operating with continuous activation functions. It is not too surprising that continuous variables s_i can encode more information than discrete ones. This solution is destroyed in the final stage of the learning process when the program tries to perform the limit $\beta \rightarrow \infty$.

Other interesting examples with `nin` = 5 the reader might wish to study are the mirror symmetry function (`key` = 88224411_{hex}) and the shift register (`key`₁ = AAAAAAAA_{hex}, `key`₂ = FFFF0000_{hex}, `key`₃ = FF00FF00_{hex}, `key`₄ = F0F0F0F0_{hex}, `key`₅ = CCCCCCCC_{hex}).

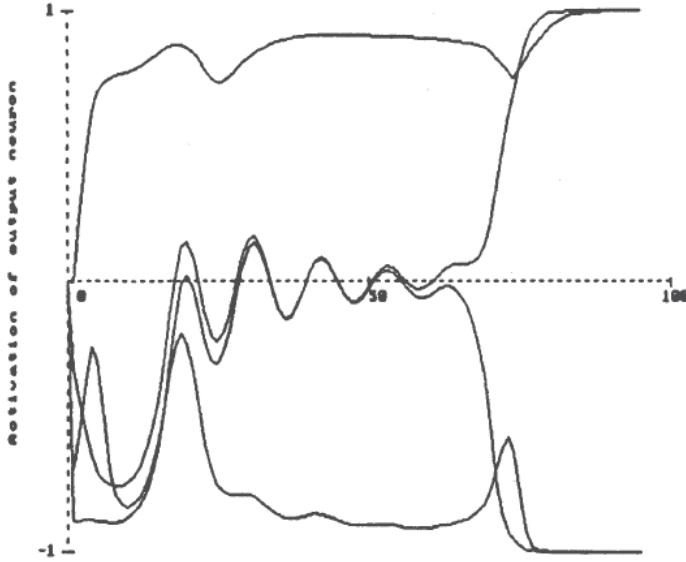


Figure C.1: A 2–2–1 perceptron learns the function XOR. The curves depict the activation of the output neuron S when the network is presented with the four possible input patterns as a function of training epochs. Parameters: $\varepsilon = 0.1$, $\alpha = 0.9$, $\beta = 1$.

The *symmetry function* signals whether the input bit pattern has mirror symmetry. As an example the following table shows the weights and thresholds of a 5–2–1 network which can recognize this property².

+2.248	-1.058	+0.029	+1.020	-2.196	-1.149	+3.113
-2.659	+1.289	+0.021	-1.284	+2.628	-1.066	+2.759
						+2.627

Table C.3

We note the antisymmetric structure of the couplings in the input layer. The mirror symmetry property of a 5-bit pattern does not depend on the bit in the center. We observe that the third input unit is indeed decoupled and has no influence on the state of the network. The output layer performs a logical conjunction of the states of the two hidden neurons. The solution found above was obtained using the parameters $\varepsilon = 0.01$, $\alpha = 0.9$, $\beta = 1$. However, very often the back-propagation algorithm does not converge to a valid solution.

C.3 Snapshots

²Incidentally, two hidden neurons are sufficient to solve the mirror-symmetry problem of arbitrary dimension!

C PERBOOL: learning Boolean functions with back-propagation

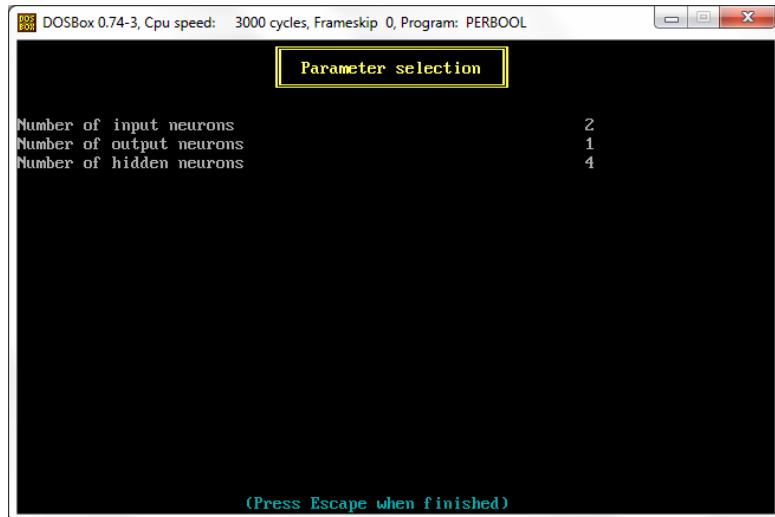


Figure C.2: Length of the input, hidden and output layers.



Figure C.3: Definition of the XOR through its truth table.

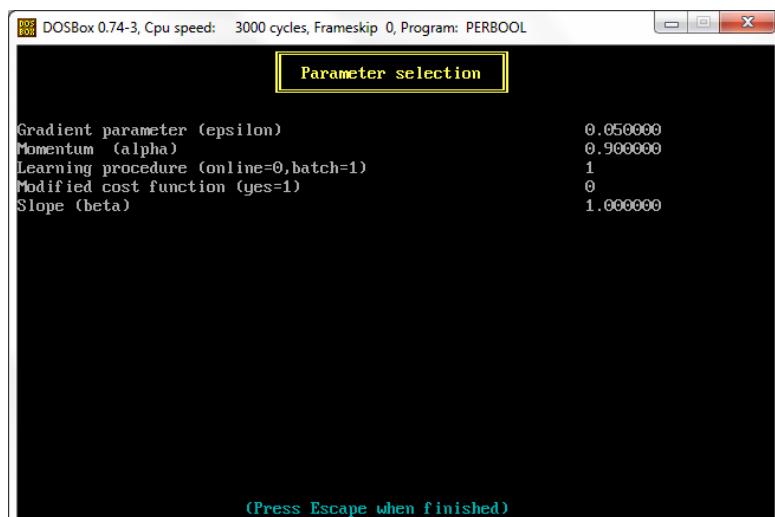


Figure C.4: Parameters of the learning protocol.

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: PERBOOL

1.373	1.398	1.777	-1.775
1.704	1.798	-0.947	2.375
-1.694	1.824	-0.956	-1.426
0.468	-1.866	-1.332	-0.993
			0.809

Iteration Deviation Beta

15	0.126	1.2
16	0.072	1.3
17	0.042	1.3
18	0.025	1.4
19	0.016	1.5
20	0.011	1.6
21	0.007	1.6
22	0.005	1.7
23	0.004	1.8
24	0.003	1.9
25	0.003	2.0
26	0.002	2.1
27	0.002	2.2
28	0.001	2.3
29	0.001	2.4
30	0.001	2.5
31	0.001	2.7

CONVERGENCE at iteration step 31

Learning using 'backpropagation'

g: graphics on/off
r: show intermediate results
p: change parameters
s: slow motion (normal/slow/pause)
e: exit

Figure C.5: Display of the training iterations.

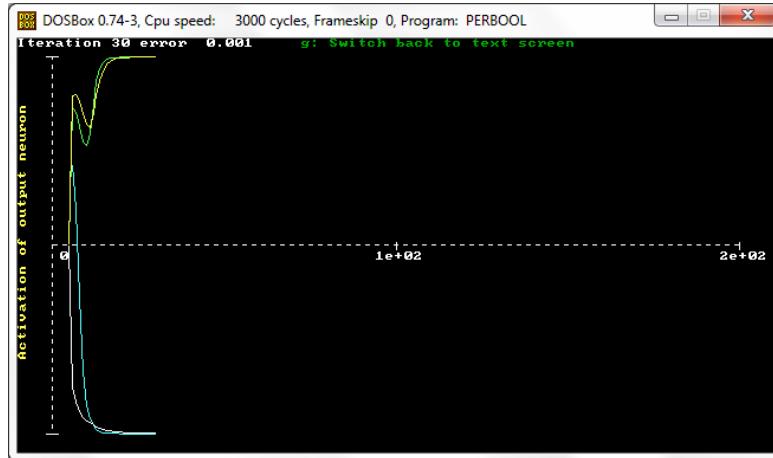


Figure C.6: Graphic representation of the learning process.

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: PERBOOL

Synaptic strength Input --> hidden layer

WIN(0,k)	k= 0:	1.373	1: 1.398
WIN(1,k)	k= 0:	1.704	1: 1.798
WIN(2,k)	k= 0:	-1.694	1: 1.824
WIN(3,k)	k= 0:	0.468	1: -1.866

Threshold values of hidden layer

1.777	-0.947	-0.956	-1.332
-------	--------	--------	--------

Synaptic strength hidden layer --> Output

WOUT(0,j)	j= 0:	-1.775	1: 2.375	2: -1.426	3: -0.993
------------	-------	--------	----------	-----------	-----------

Threshold values of Output layer

0.809

Figure C.7: Optimal synaptic coefficients and thresholds.

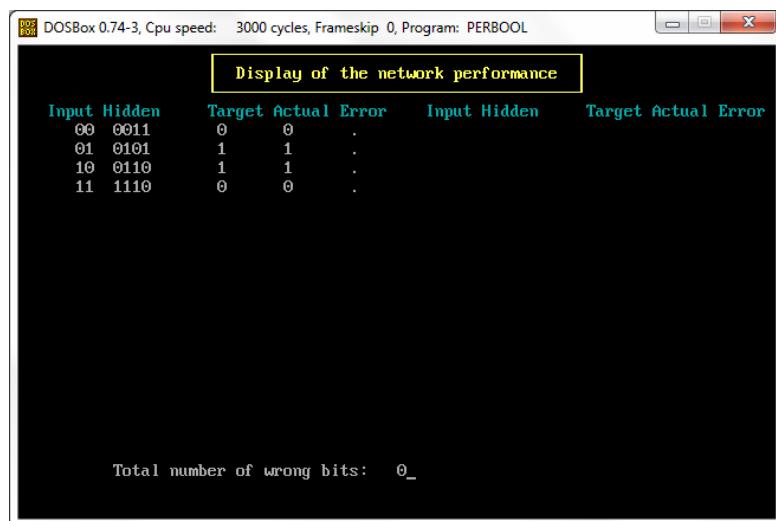


Figure C.8: Network performance.

APPENDIX D

PERFUNC: learning continuous functions with back-propagation

As discussed in Sec. 3.4.4, feed-forward networks with hidden layers can represent any smooth continuous function $\mathbb{R}^n \rightarrow \mathbb{R}^m$. As an illustration the program PERFUNC is designed to solve the following task: when presented with the values of a specimen $g(x)$ out of a class of one-dimensional functions at a set of points x_i , it predicts the function values at the output points x'_i . This is achieved by training the network with the error back-propagation algorithm by presenting various ‘patterns’ σ_i^μ where μ refers to specific values of the continuous parameters on which the function $g(x)$ depends. The training method is essentially the same as was used for Boolean-function learning, so that its description need not be repeated. As a slight variation, the couplings w_{ij} to the output layer are realized by *linear* transfer functions $\tilde{f}(x) = x$. In contrast to the bounded sigmoidal function $f(x) = \tanh(\beta x)$ used for the interior layers, this enables the output signals S_i to take on arbitrarily large values.

D.1 Program description

PERFUNC is closely related to the program PERBOOL described in Appendix C. Differences arise from the fact that *0, 1, or 2 hidden layers* are allowed and from the way training patterns are provided. Since learning can take a long time, the synaptic coefficients and thresholds can be read in from a data file to provide for a fast demonstration of the network’s capabilities. Alternatively, to operate in learning mode the following parameters can be specified via the input panel of the program.

- **nlayers:** The number of hidden layers which may take the values 0, 1, or 2.
- **nin, nout, nhid:** The number of neurons in the input, output, and hidden layer. If there are two hidden layers, these will have equal numbers of neurons.
- **ntyp:** The type of function $g(x)$ to be learned. The three choices are:
 - $g_1(x) = c_1x + c_2$
 - $g_2(x) = \sin \left[\left(c_1x + \frac{1}{2}c_2 \right) 2\pi \right]$
 - $g_3(x) = c_2x^{1+c_1}$

- **nparm**: The number of function parameters, e.g. the frequency and phase of the sine function $g_2(x)$. If **nparm** = 1, the second parameter is absent, $c_2 = 0$, or $c_2 = 1$ in the case of $g_3(x)$.
- **patts**: The number of different patterns to be used in training. A pattern consists of the input and output values σ_l , $l = 1, \dots, \text{nin}$, S_i , $i = 1, \dots, \text{nout}$ produced by a representative of the class of functions $g(x)$. It is generated by randomly selecting values for the parameters c_1 and c_2 in the range $[-1, +1]$.
- **errmax**: Target value for the mean error per output neuron. Learning is stopped when this accuracy is reached.

Subsequently the program asks for the set of input and output coordinates x_i and x'_i .

A second input panel asks for the following parameters which determine the learning phase:

- **epsilon**: The learning rate ε .
- **alpha**: The “momentum” constant α .
- **multi**: The multiplicity of pattern presentations. If **multi** > 1, the same pattern is presented repeatedly to the back-propagation algorithm before going to the next pattern (note: the momentum parameter α has no effect in this mode).
- **fixpat**: This variable determines whether a fixed set of patterns is used for training, or whether at the beginning of each training epoch new patterns are chosen.
- **beta**: The inverse steepness β of the activation function $f(x)$.

The learning process is started by initializing all couplings and thresholds to random numbers in the range $[-1, 1]$. After each learning epoch the current coupling and threshold values are displayed according to the format introduced in Sec. C.1. For **nlayers** = 2 only the parameters of the first hidden layer and the output layers are displayed; for **nlayers** = 1 the lines (columns) correspond to the input (output) units of the network. In addition a listing of the error value as a function of the training epoch number is maintained. During the training phase the user can interact with the program in various ways. After pressing the key ‘p’ the parameters which influence the learning process may be changed: **epsilon**, **alpha**, **multi**, **fixpat**, **beta**. With the key ‘r’ the current network performance may be investigated. Finally, by pressing ‘e’ the learning is stopped.

Three different choices are offered for examining the current performance:

- **n**: A numerical listing of the errors for an equidistant mesh of parameter values c_1 , c_2 in the interval $[-1, 1]$. If **nparm** = 1, the values $S_1^{\mu=1}$, ζ_1^{μ} , and the mean square error $D^{\mu} = \sqrt{\sum_{i=1}^{\text{nout}} (S_i^{\mu} - \zeta_i^{\mu})^2}$ are listed. For **nparm** = 2 a table of values of D^{μ} is displayed, where the columns correspond to the variation of the second parameter c_2 with an increment of 0.2, while the rows are calculated for different values of c_1 (increment 0.1). In addition the mean error for all patterns is printed.
- **p**: With this option the program shows a graphical plot of the deviations $S_i^{\mu} - \zeta_i^{\mu}$, $i = 1, \dots, \text{nout}$ as a function of the parameter c_1 . The range of display and the value of c_2 , which is kept fixed, can be entered.
- **f**: The program prompts for the entry of the constants c_1 and c_2 . Then a plot of the function $g(x)$ is drawn together with the input values $g(x_i)$ (marked by + symbols) and the output values $g(x'_i)$ produced by the network (marked by x).

After a final examination of the network performance the couplings can be stored on a data file.

D.2 Numerical experiments

Since the learning of continuous functions is rather time consuming, it might be wise to start with very simple examples. There is only one case where an exact solution is possible: the linear network trained to represent a linear function. Indeed, for a 1–1 network trained with the general linear function $g_1(x)$ (`nlayers` = 0, `nin` = 2, `nout` = 1, `nparm` = 1) the learning algorithm quickly finds the solution, which is $w_{11} = \frac{x_2-x'}{x_2-x_1}$, $w_{12} = \frac{x'-x_1}{x_2-x_1}$, $\vartheta_1 = 0$. This is quite insensitive to the details of the training rule as long as the gradient parameter `epsilon` is not too large. Otherwise the algorithm becomes unstable and the weights ‘explode’, leading to an exponentially growing error. Learning is achieved both with a fixed set of ‘patterns’ (i.e. combinations of c_1 and c_2) and with patterns which are selected at random in each step, although in the latter case the convergence is markedly slower. If there are more input values than necessary, the problem is underdetermined. This does not impede the learning process, but different sets of weights w_{1i} are found in each run, because of the random initial conditions.

For a network with a *nonlinear* transfer function $f(x)$ the representation of a linear function will not be exact. For the trivial 1–1–1 architecture (`nlayers` = 1, `nin` = 1, `nhid` = 1, `nout` = 1) trained with the one-parameter linear function the network represents the general function $S = w \tanh(\bar{w}\sigma - \bar{\vartheta}) - \vartheta$. This has to provide a fit to the function $\zeta = (x'/x)\sigma \equiv c\sigma$. In principle this can be accomplished with arbitrary precision by scaling the weights so that the linear region of the tanh function is probed: $\bar{\vartheta} = \vartheta = 0$, $\bar{w} = \varepsilon$, $w = c/\varepsilon$ with $\varepsilon \rightarrow 0$. Experiments with the program show that a fairly good representation of the linear function can be achieved, but the relative size of the weights w and \bar{w} remains of the order of one instead of diverging.

To study the learning of more challenging tasks, i.e. nonlinear functions depending on two parameters, the reader has to muster some patience. It can take several thousand learning epochs until a satisfactory representation of the function is reached. Provided that the gradient factor `epsilon` has a sufficiently small value, some degree of convergence can be achieved for various combinations of the parameters which determine the training process. However, it seems to be necessary that the same patterns are presented repeatedly in subsequent training epochs. If in each epoch new values of c_1 and c_2 are chosen at random (`fixpat` = 0), the errors remain at a high level.

The data files `PERFUNC1.DAT` and `PERFUNC2.DAT` contain some typical results. Networks comprising one (or two) hidden layers with the topology 5–20–5 (or 5–10–10–5) have been trained to represent the two-parameter sine function $g_2(x)$. The networks provide a mapping from the function values at the input grid $x_i = 0.1, 0.2, 0.3, 0.4, 0.5$ to the output grid $x'_i = 0.6, 0.7, 0.8, 0.9, 1.0$. The mean deviation is of the order of about $S_i - \zeta_i \simeq 0.2$. The error, starts to grow at the edges of the range of parameter values for which training examples have been provided. The network cannot be expected to (and indeed shows no tendency to) “generalize” in the sense of, e.g., extrapolating to sine functions with higher frequency. This is easily observed by plotting the error as a function of c_1 (using the option `p` in the ‘Display network performance’ panel) in the range of, say, $-2 < c_1 < +2$. As shown in Fig. D.1 the error rapidly grows outside the training interval $-1 < c_1 < +1$. Pictorially speaking, the effect of the learning is to “dig a hole” into the error surface in the region where training patterns are provided

Experience with the back-propagation algorithm shows that learning becomes very slow

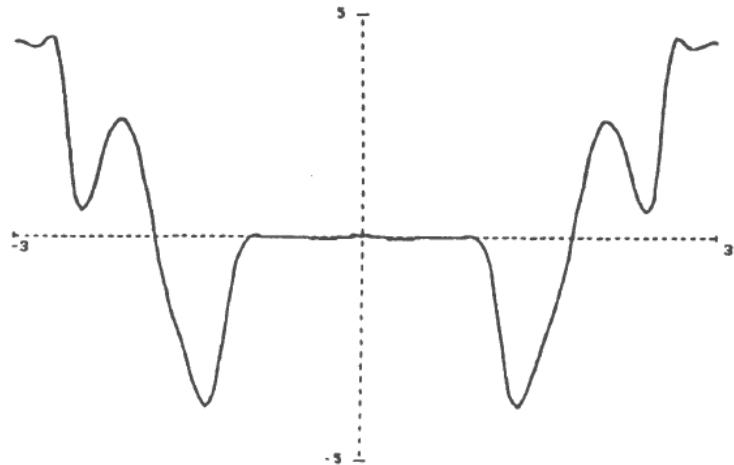


Figure D.1: Values of the error of the output signals of a network trained to represent the sine function $g_2(x)$ at five grid points, drawn as a function of the frequency parameter c_1 . Outside the region used for training, $-1 < c_1 < +1$, the error rapidly increases.

when many hidden layers of some complexity are involved. To train networks with a large number of hidden layers the use of alternative strategies, such as the genetic algorithms seems to be advantageous.

D.3 Snapshots

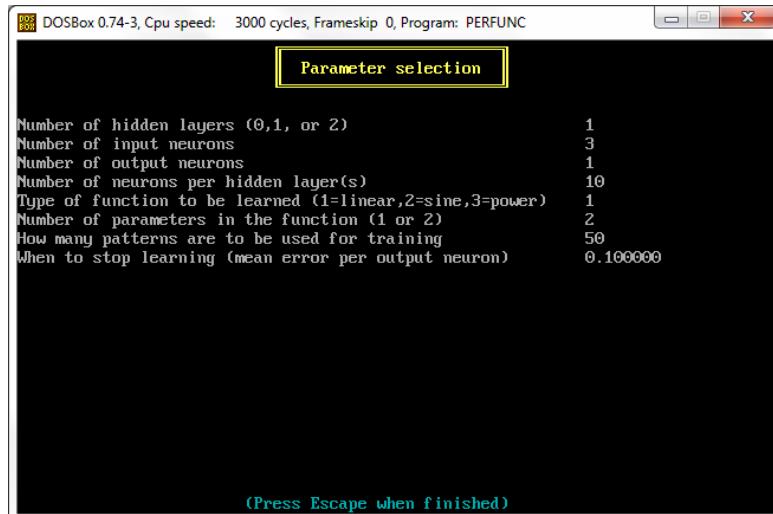
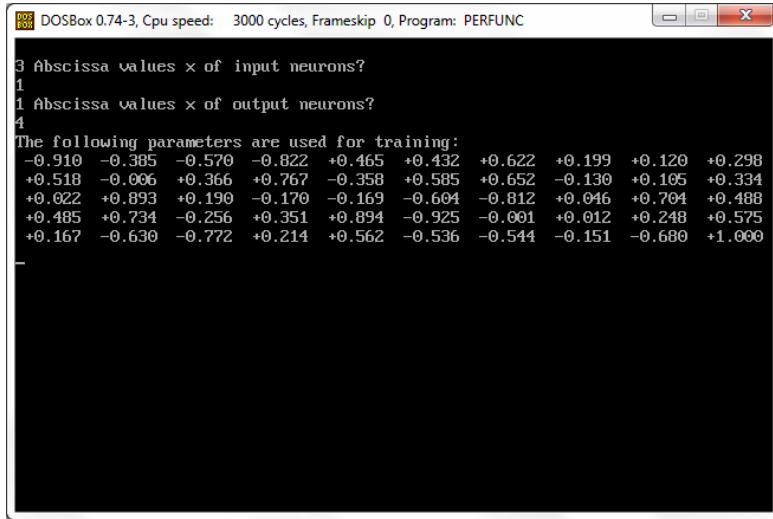


Figure D.2: Selection of the network architecture and parameters.



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: PERFUNC

```

3 Abscissa values x of input neurons?
1
1 Abscissa values x of output neurons?
4
The following parameters are used for training:
-0.910 -0.385 -0.570 -0.822 +0.465 +0.432 +0.622 +0.199 +0.120 +0.298
+0.518 -0.006 +0.366 +0.767 -0.358 +0.585 +0.652 -0.130 +0.105 +0.334
+0.022 +0.893 +0.190 -0.170 -0.169 -0.604 -0.812 +0.046 +0.704 +0.488
+0.485 +0.734 -0.256 +0.351 +0.894 -0.925 -0.001 +0.012 +0.248 +0.575
+0.167 -0.630 -0.772 +0.214 +0.562 -0.536 -0.544 -0.151 -0.680 +1.000
-
```

Figure D.3: Insertion of the learning dataset ($x_i = 1, 3, 5, x'_i = 4$).

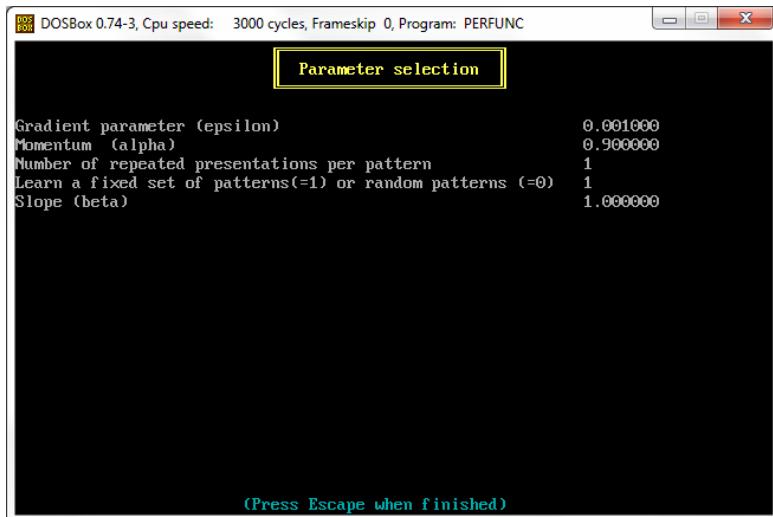
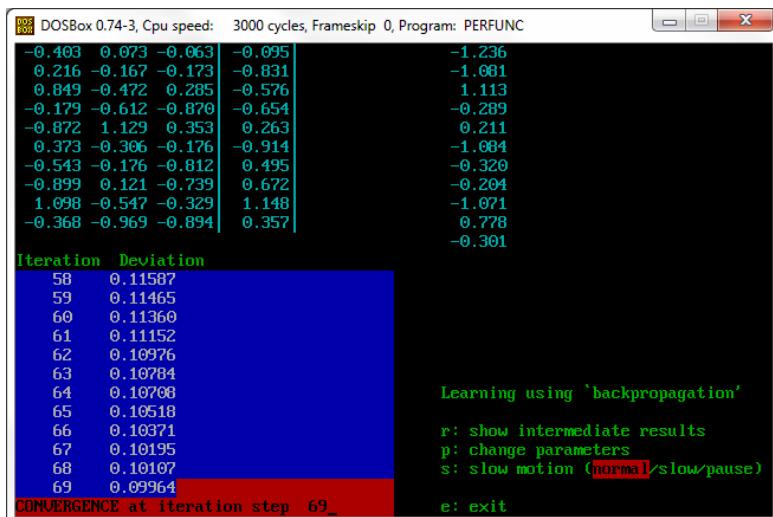


Figure D.4: Parameters of the learning protocol.



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: PERFUNC

-0.403 0.073 -0.063	-0.095	-1.236
0.216 -0.167 -0.173	-0.831	-1.001
0.849 -0.472 0.285	-0.576	1.113
-0.179 -0.612 -0.870	-0.654	-0.289
-0.872 1.129 0.353	0.263	0.211
0.373 -0.306 -0.176	-0.914	-1.084
-0.543 -0.176 -0.812	0.495	-0.320
-0.899 0.121 -0.739	0.672	-0.204
1.098 -0.547 -0.329	1.148	-1.071
-0.368 -0.969 -0.894	0.357	0.778

Iteration Deviation

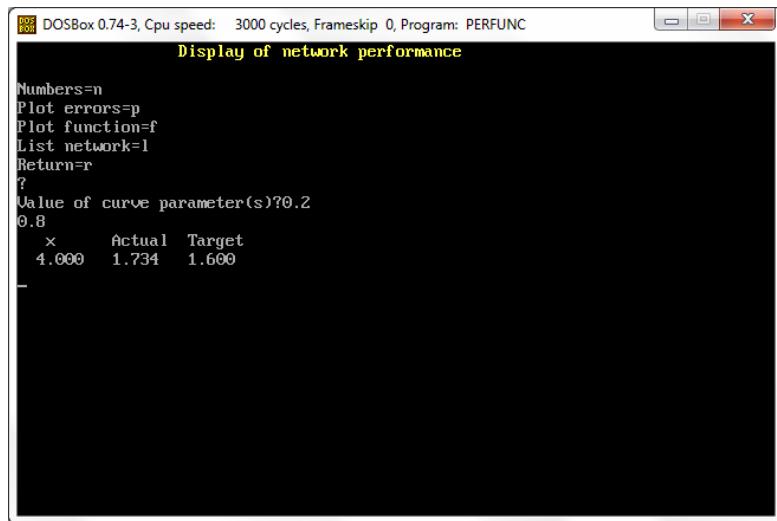
58	0.11587
59	0.11465
60	0.11360
61	0.11152
62	0.10976
63	0.10784
64	0.10708
65	0.10518
66	0.10371
67	0.10195
68	0.10107
69	0.09964

CONVERGENCE at iteration step 69

Learning using 'backpropagation'

r: show intermediate results
 p: change parameters
 s: slow motion (normal|slow/pause)
 e: exit

Figure D.5: Display of the training iterations.



```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: PERFUNC
Display of network performance

Numbers=n
Plot errors=p
Plot function=f
List network=l
Return=r
?
Value of curve parameter(s)?0.2
0.8
    x      Actual   Target
  4.000   1.734   1.600
-
```

Figure D.6: Network performance.

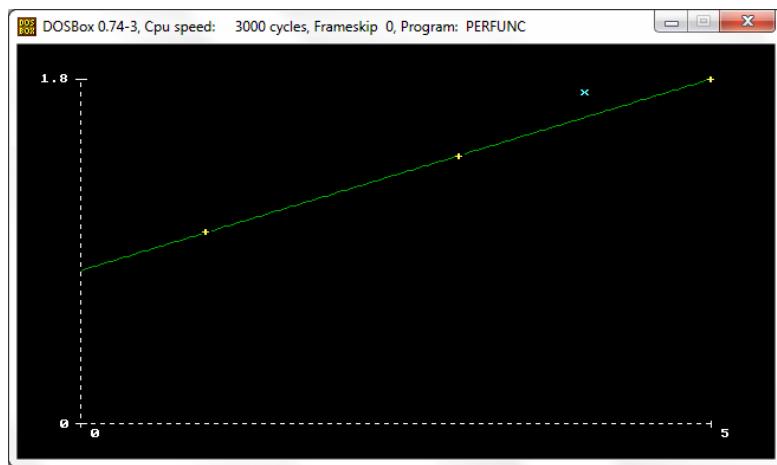


Figure D.7: Graphic representation ($c_1 = 0.2$, $c_2 = 0.8$).

References

Books

- [1] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall, 2010. ISBN: 978-0136042594.
- [2] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson, 2020. ISBN: 978-0123821881.
- [3] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press, 2007. ISBN: 978-0521880688.
- [4] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery. *Fortran Numerical Recipes*. Vol. 1: *Numerical Recipes in Fortran 77: : The Art of Scientific Computing* (2nd ed., with corrections). Cambridge University Press, 1997. ISBN: 978-0521430647.
- [5] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery. *Fortran Numerical Recipes*. Vol. 2: *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing* (2nd ed., with corrections). Cambridge University Press, 1997. ISBN: 978-0521574396.
- [6] B. Müller, J. Reinhardt and M. T. Strickland. *Neural Networks: An Introduction* (2nd ed.). Springer-Verlag, 1995. ISBN: 978-3540602071.
- [7] P. Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems* (1st ed.). The MIT Press, 2005. ISBN: 978-0262541855.
- [8] A. Géron. *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow* (1st ed.). O'Reilly Media, 2017. ISBN: 978-1491962299.
- [9] A. Géron. *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow* (2nd ed.). O'Reilly Media, 2019. ISBN: 978-1492032649.

Articles

- [10] S. Lin. “Computer Solutions of the Traveling Salesman Problem”. In: *The Bell System Technical Journal* **44**:10 (Dec. 1965), pp. 2245–2269. DOI: [10.1002/j.1538-7305.1965.tb04146.x](https://doi.org/10.1002/j.1538-7305.1965.tb04146.x).
- [11] J. J. Hopfield, D. I. Feinstein and R. G. Palmer. “‘Unlearning’ Has a Stabilizing Effect in Collective Memories”. In: *Nature* **304**:5922 (July 1983), pp. 158–159. DOI: [10.1038/304158a0](https://doi.org/10.1038/304158a0).
- [12] T. J. Sejnowski, P. K. Kienker and G. E. Hinton. “Learning Symmetry Groups with Hidden Units: Beyond the Perceptron”. In: *Physica* **D**:22 (Oct. 1986), pp. 260–275.

References

- [13] P. Ramachandran, B. Zoph and Q. Le. “Searching for Activation Functions”. In: *arXiv e-prints* (Oct. 2017). arXiv: 1710.05941v2 [cs.NE].
- [14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou et al. “Mastering the Game of Go Without Human Knowledge”. In: *Nature* **550** (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.
- [15] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou et al. “A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play”. In: *Science* **362**:6419 (Dec. 2018), pp. 1140–1144. DOI: 10.1126/science.aar6404.
- [16] D. Misra. “Mish: A Self Regularized Non-Monotonic Activation Function”. In: *arXiv e-prints* (Aug. 2020). arXiv: 1908.08681v3 [cs.LG].

Incollections

- [17] S. Amarel. “On Representations of Problems of Reasoning about Actions”. In: *Machine Intelligence 3*. Ed. by D. Michie. American Elsevier Publisher, 1968, pp. 131–171. DOI: 10.1016/B978-0-934613-03-3.50006-4.

Technical reports

- [18] R. J. Williams. *Reinforcement-learning in Connectionist Networks: A Mathematical Analysis*. 8605. Institute for Cognitive Science, University of California, San Diego, 1986.

Lectures notes

- [19] P. Compeau. *02-251: Great Ideas in Computational Biology Notes on Neural Networks*. 2019. URL: https://www.cs.cmu.edu/~02251/lectures/neural_nets.pdf.

Online & websites

- [20] D. Biebighauser. *Testing Random Number Generators*. 2000. URL: <http://www-users.math.umn.edu/~garrett/students/reu/pRNGs.pdf>.
- [21] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou et al. *Supplementary Materials for A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play*. URL: <https://science.sciencemag.org/content/suppl/2018/12/05/362.6419.1140.DC1>.
- [22] URL: <https://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/attachments.html?highlight=blacklist>.
- [23] URL: <https://github.com/aimacode/aima-python>.
- [24] URL: <https://github.com/ageron/handson-ml2>.
- [25] URL: <https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/>.
- [26] URL: <https://deniz.co/8-puzzle-solver/>.
- [27] URL: https://en.wikipedia.org/wiki/Eight_queens_puzzle.
- [28] URL: <https://pip.pypa.io/en/stable/>.

- [29] URL: <https://pypi.org/project/geneticalgorithm/>.
- [30] URL: <https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>.
- [31] URL: <https://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/>.
- [32] URL: <https://www.tensorflow.org/install>.
- [33] URL: <https://docs.anaconda.com/anaconda/user-guide/tasks/tensorflow/>.
- [34] URL: <https://distill.pub/2017/momentum/>.
- [35] URL: <https://pytorch.org/get-started/locally/>.
- [36] URL: [https://en.wikipedia.org/wiki/Go_\(game\)](https://en.wikipedia.org/wiki/Go_(game)).
- [37] URL: https://en.wikipedia.org/wiki/Rules_of_Go.
- [38] URL: <https://senseis.xmp.net/>.
- [39] URL: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.
- [40] URL: <https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238>.
- [41] URL: <https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/>.
- [42] URL: <https://arxiv.org/pdf/2103.04931.pdf>.
- [43] URL: <https://gym.openai.com/>.
- [44] URL: <https://medium.com/applied-data-science/how-to-build-your-own-alpha-zero-ai-using-python-and-keras-7f664945c188>.
- [45] URL: <https://web.stanford.edu/~surag/posts/alphazero.html>.
- [46] URL: <https://www.dosbox.com/>.