A survey of machine learning

First edition

by Carl Burch

for the Pennsylvania Governor's School for the Sciences

Contents

Chapter 0

Introduction

Machine learning attempts to tell how to automatically find a good predictor based on past experiences. Although you might argue that machine learning has been around as long as statistics has, it really only became a separate topic in the 1990's. It draws its inspiration from a variety of academic disciplines, including computer science, statistics, biology, and psychology.

In this class we're going to look at some of the significant results from machine learning. One goal is to learn some of the techniques of machine learning, but also, just as significant, we are going to get a glimpse of the research front and the sort of approaches researchers have taken toward this very nebulous goal of automatically finding predictors.

Researchers have approached the general goal of machine learning from a variety of approaches. Before we delve into details about these, let's do a general overview of machine learning research. This also constitutes something of an outline of this text: We'll spend a chapter on each of the following topics.

0.1 Data mining

With the arrival of computing in all facets of day-to-day business, the amount of accessible data has exploded. Employers keep information about employees, businesses keep information about customers, hospitals keep information about patients, factories get information about instrument performance, scientists collect information about the natural world — and it's all stored in computers, ready to access in mass.

Data mining is gradually proving itself as an important tool for people who wish to analyze all this data for patterns. One of the most famous examples is from the late 1970s, when data mining proved itself as potentially important for both scientific and commercial purposes on a particular test application of diagnosing diseases in soybean plants [MC80].

First the researchers found an expert, who they interviewed for a list of rules about how to diagnose diseases in a soybean plant. Then they collected about 680 diseased soybean plants and determined about 35 pieces of data on each case (such as the month the plant was found to have a disease, the amount of recent precipitation, the size of the plant's seeds, the condition of its roots). They handed the plants to the expert to diagnose, and then they performed some data mining techniques to look for rules predicting the disease based on the characteristics they measured.

What they found is that the rules the expert gave during the interview were accurate only 70% of the time, while the rules discovered through data mining were accurate 97.5% of the time. Moreover, after revealing these rules to the expert, the expert reportedly was impressed enough to adopt some of the rules in place of the ones given during the interview!

In this text, we'll see a few of the more important machine learning techniques used in data mining, as

well as surrounding issues that apply regardless of the learning algorithm. We'll emerge from this familiar with much of the established machine learning results and prepared to study less polished research.

0.2 Neural networks

Cognitive science aims to understand how the human brain works. One important part of cognitive science involves simulating models of the human brain on a computer to learn more about the model and also potentially to instill additional intelligence into the computer.

Though scientists are far from understanding the brain, machine learning has already reaped the reward of *artificial neural networks* (ANNs). ANNs are deployed in a variety of situations. One of the more impressive is ALVINN, a system that uses an ANN to steer a car on a highway [Pom93]. ALVINN consists of an array of cameras on the car, whose inputs are fed into a small ANN whose outputs control the steering wheel. The system has been tested going up to 70 miles per hour over 90 miles of a public divided highway (with other vehicles on the road).

Here we'll look briefly at human neurons and how artificial neurons model their behavior. Then we'll see how they might be networked together to form an artificial neural network that improves with training.

0.3 Reinforcement learning

Through this point, we'll have worked exclusively with systems that need immediate feedback from their actions in order to learn. This is **supervised learning**, which though useful is a limited goal.

Reinforcement learning (sometimes called **unsupervised learning**) refers to a brand of learning situation where a machine should learn to behave in situations where feedback is not immediate. This is especially applicable in robotics situations, where you might hope for a robot to learn how to accomplish a certain task in the same way a dog learns a trick, without explicit programming.

Probably the most famous success story from reinforcement learning is TD-Gammon, a program that learns to play the game of Backgammon [Tes95]. TD-Gammon uses a neural network coupled with reinforcement learning techniques. After playing against itself for 1.5 million games, the program learned enough to rank among the world's best backgammon players (including both humans and computers).

Reinforcement learning is much more challenging than supervised learning, and researchers still don't have a good grasp on it. We'll see a few of the proposed techniques, though, and how they can be applied in situations like the one that TD-Gammon tackles.

0.4 Artificial life

Finally, **artificial life** seeks to emulate living systems with a computer. Our study of artificial life will concentrate on genetic algorithms, where systems loosely based on evolution are simulated to see what might evolve. They hope to evolve very simple behavior, like that of amoebas, thus gaining a greater understanding of how evolution works and what effects it has.

In a sense, the evolutionary process *learn* — though usually the verb we use is *adapt.* If you understanding learning as improving performance based on past experience, the word *learn* has a similar denotation to *adapt*, even if the connotation is different.

Genetic algorithms appear to be a promising technique for learning from past experience, even outside simulations of pseudo-biological. We'll look at this technique, and then we'll look at its use in attempting to evolve simple behaviors.

Chapter 1

Data mining

Data is abundant in today's society. Every transaction, every accomplishment gets stored away somewhere. We get much more data than we could ever hope to analyze by hand. A natural hope is to analyze the data by computer. **Data mining** refers to the various tasks of analyzing stored data for patterns — seeking clusters, trends, predictors, and patterns in a mass of stored data.

For example, many grocery stores now have customer cards, rewarding frequent users of the grocery store with discounts on particular items. The stores gives these cards to encourage customer loyalty and to collect data — for, with this cards, they can track a customer between visits to the store and potentially mine the collected data to determine patterns of customer behavior. This is useful for determining what to promote through advertisement, shelf placement, or discounts.

A large banking corporation makes many decisions about whether to accept a loan application or not. On hand is a variety of information about the applicant — age, employment history, salary, credit history — and about the loan application — amount, purpose, interest rate. Additionally, the bank has the same information about thousands of past loans, plus whether the loan proved to be a good investment or not. From this, the bank wants to know whether it should make the loan. Data mining can potentially improve the loan acceptance rate without sacrificing on the default rate, profiting both the bank and its customers.

1.1 Predicting from examples

We'll emphasize a particular type of data mining called **predicting from examples**. In this scenario, the algorithm has access to several **training examples**, representing the past history. Each training example is a vector of values for the different **attributes** measured. In our banking application, each example represents a single loan application, represented by a vector holding the customer age, customer salary, loan amount, and other characteristics of the loan application. Each example has a **label**, which in the banking application might be a rating of how well the loan turned out.

The learning algorithm sees all these labeled examples, and then it should produce a **hypothesis** — a new algorithm that, given a new vector as an input, produces a prediction of its label. (That a learning algorithm produces another algorithm may strike you as odd. But, given the variety of types of hypotheses that learning algorithms produce, this is the best we can do formally.)

In Section 1.1, we briefly looked at some data mining results for soybean disease diagnosis by Michalski and Chilausky [MC80]. Figure 1.1 illustrates a selection of the data they used. (I've taken just seven of the 680 plants and just four of the 35 attributes.) There are six training examples here, each with four attributes (plant growth, stem condition, leaf-spot halo, and seed mold) and a label (the disease). We would feed those into the learning algorithm, and it would produce a hypothesis that labels any plant with a supposed disease.

	plant growth	stem condition	halo on leaf spots	mold on seed	disease
Plant A	normal	abnormal	no	no	frog eye leaf spot
Plant B	abnormal	abnormal	no	no	herbicide injury
Plant C	normal	normal	yes	yes	downy mildew
Plant D	abnormal	normal	yes	yes	bacterial pustule
Plant E	normal	normal	yes	no	bacterial blight
Plant F	normal	abnormal	no	no	frog eye leaf spot
Plant Q	normal	normal	no	yes	???

Figure 1.1: A prediction-from-examples problem of diagnosing soybean diseases.

	sepal length	sepal width	petal length	petal width	species
Iris A	4.7	3.2	1.3	0.2	*setosa*
Iris B	6.1	2.8	4.7	1.2	*versicolor*
Iris C	5.6	3.0	4.1	1.3	*versicolor*
Iris D	5.8	2.7	5.1	1.9	*virginica*
Iris E	6.5	3.2	5.1	2.0	*virginica*
Iris Q	5.8	2.7	3.9	1.2	???

Figure 1.2: A prediction-from-examples problem of Iris classification.

Plant Q, representing a new plant whose disease we want to diagnose, illustrates what we might feed to the hypothesis to arrive at a prediction.

In the soybean data, each of the four attributes has only two possible values (either normal and abnormal, or yes and no); the full data set has some attributes with more possible values (the month found could be any month between April or October; the precipitation could be below normal, normal, or above normal). But each attribute has just a small number of possible values. Such attributes are called **discrete attributes**.

In some domains, attributes may be **numeric** instead. A numeric attribute has several possible values, in a meaningful linear order. Consider the classic data set created by Fisher, a statistician working in the mid-1930s [Fis36]. Fisher measured the sepals and petals of 150 different irises, and labeled them with the specific species of iris. For illustration purposes, we'll just work with the five labeled examples of Figure 1.2 and the single unlabeled example whose identity we want to predict.

These two examples of data sets come from actual data sets, but they are simpler than what one normally encounters in practice. Normally, data has many more attributes and several times the number of examples. Moreover, some data will be missing, and attributes are usually mixed in character — some of them will be discrete (like a loan applicant's state of residence) while others are numeric (like the loan applicant's salary). But these things complicate the basic ideas, which is what we want to look at here, so we'll keep with the idealized examples.

1.2 Techniques

We're going to look at three important techniques for data mining. The first two — linear regression and nearest-neighbor search — are best suited for numeric data. The last, ID3, is aimed at discrete data.

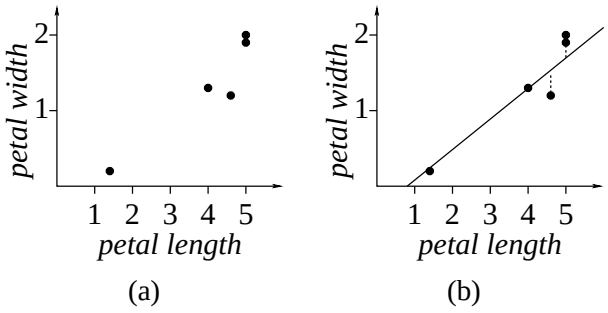


Figure 1.3: Fitting a line to a set of data.

1.2.1 Linear regression

Linear regression is one of the oldest forms of machine learning. It is a long-established statistical technique that involves simply fitting a line to some data.

Single-attribute examples The easiest case for linear regression is when the examples have a single numeric attribute and a numeric label; we'll look at this case first. Say we have n examples, where the attribute for each example is called x_i, and the label for each is y_i. We can envision each example as being a point in 2-dimensional space, with an x-coordinate of x_i and a y-coordinate of y_i. (See Figure 1.3(a).)

Linear regression would seek the line $f(x) = mx + b$ (specifying the prediction $f(x)$ for any given single-attribute example $\langle x \rangle$) that minimizes the **sum-of-squares error** for the training examples,

$$\sum_{i=1}^{n} (y_i - f(x_i))^2 \, .$$

(See Figure 1.3(b).) The quantity $|y_i - f(x_i)|$ is the distance from the value predicted by the hypothesis line to the actual value — the error of the hypothesis for this training example. Squaring this value gives greater emphasis to larger errors and saves us dealing with complicated absolute values in the mathematics.

For example, we might want to predict the petal width of an iris given its petal length using the data of Figure 1.2. Here x_1 is 1.3 (the petal length of Iris A) and y_1 is 0.2. From Iris B, we get $x_2 = 4.7$ and $y_2 = 1.2$. We get similar data from Iris C, Iris D, and Iris E. Figure 1.3 graphs the points.

With a little bit of calculus, it's not too hard to compute the exact values of m and b that minimize the sum-of-squares error. We'll skip the derivation and just show the result. Let $\bar{x}$ be the average x value ($(\sum_i x_i)/n$) and $\bar{y}$ be the average y value ($(\sum_i y_i)/n$). The optimal choices for m and b are

$$m = \frac{(\sum_i x_i y_i) - n\bar{x}\bar{y}}{(\sum_i x_i^2) - n\bar{x}^2}, b = \bar{y} - m\bar{x} \, .$$

For the irises, we compute that $\sum_i x_i = 20.3$ and hence $\bar{x} = 4.05$, $\sum_i y_i = 6.6$ and hence $\bar{y} = 1.32$, $\sum_i x_i y_i = 31.12$, $\sum_i x_i^2 = 92.61$. Knowing these, we compute m to be $(31.12 - 5 \cdot 4.05 \cdot 1.32)/(92.61 - 5 \cdot 4.05^2) = 4.39/10.5975 \approx 0.4142$ and b to be $1.32 - 0.4142 \cdot 4.05 = -0.3577$. So our hypothesis is that, if the petal length is x, then the petal width will be

$$0.4142x - .3577 \, .$$

On Iris Q, with a petal length of 3.9, this hypothesis would predict a petal width of 1.26, not too far from the actual value of 1.2. (On the other hand, the hypothesis would predict a negative *petal width for a very short petal, which is clearly not appropriate.)*

Multiple-attribute examples When each piece of training data is a vector of *several* attributes, the problem becomes more complicated. What we're going to do is to expand the single-attribute example by expanding the dimensions of the space. If each example has just two attributes, we could view each labeled example in three-dimensional space, with an x-coordinate corresponding to the first attribute, the y-coordinate corresponding to the second attribute, and the z-coordinate corresponding to the label. We'd look for a plane $f(x, y) = m_x x + m_y y$ that minimizes the sum-of-squares error.

For a general number of attributes d, we'll view each labeled example as a point in $(d+1)$-dimensional space, with a coordinate for each attribute, plus a coordinate for the label. We'll look for a d-dimensional hyperplane $f(x_1, x_2, \ldots, x_d) = \sum_{i=1}^{d} m_i x_i$ that minimizes the sum-of-squares error.

This is slightly different than the two-dimensional case because this $f(x)$ is forced to go through the origin $(0, 0, \ldots, 0)$. Forcing this keeps the mathematics prettier. If we want to be able to learn a hyperplane that doesn't necessarily go through the origin, we can get around this limitation by simply inserting an additional attribute to every vector that is always 1 (so that $m_i x_i$ for that coordinate is just the constant m_i).

For the iris example, we're in 6*-dimensional space. We're adding the extra always-*1 *attribute, giving us a total of* $d = 5$ *attributes in each example, and then we have the label. The label needs to be numeric, though the iris example labels aren't. What we'll do is say that the label is* 1 *if the species is* versicolor *and* 0 *otherwise. Thus if the predicted value is large (at least* 0.5*), the hypothesis is that the iris is probably* versicolor *and if the predicted value is small (less than* 0.5*), the hypothesis is that it is probably not. Thus Iris A is at the point* $(1, 4.7, 3.2, 1.3, 0.2, 0)$*. Iris B is at the point* $(1, 6.1, 2.8, 4.7, 1.2, 1)$*.*

We'll use the notation x_{ij} to refer to the ith example's jth attribute value and the notation y_i to refer to the ith example's label. Thus our examples are as follows.

$$\begin{array}{cccccccc}
(& x_{11}, & x_{12}, & x_{13}, & \ldots, & x_{1d}, & y_1 &) \\
(& x_{21}, & x_{22}, & x_{23}, & \ldots, & x_{2d}, & y_2 &) \\
 & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \\
(& x_{n1}, & x_{n2}, & x_{n3}, & \ldots, & x_{nd}, & y_n &)
\end{array}$$

We want to find a set of coefficients m_i so that the function $f(x) = \sum_{i=1}^{d} m_i x_i$ as closely approximates the y_i as possible (still using the sum-of-squares error). To compute this $f(x)$, it turns out that we need to solve a set of equations.

$$\begin{array}{ccccccccc}
(\sum_i x_{i1}x_{i1})\, m_1 & + & (\sum_i x_{i1}x_{i2})\, m_2 & + & \cdots & + & (\sum_i x_{i1}x_{id})\, m_d & = & \sum_i x_{i1}y_i \\
(\sum_i x_{i2}x_{i1})\, m_1 & + & (\sum_i x_{i2}x_{i2})\, m_2 & + & \cdots & + & (\sum_i x_{i2}x_{id})\, m_d & = & \sum_i x_{i2}y_i \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
(\sum_i x_{id}x_{i1})\, m_1 & + & (\sum_i x_{id}x_{i2})\, m_2 & + & \cdots & + & (\sum_i x_{id}x_{id})\, m_d & = & \sum_i x_{id}y_i
\end{array}$$

Here we have d equations and d unknowns (namely, m_1 through m_d). After we solve for the m_i, the best-fit hyperplane is the function $f(x) = \sum_i m_i x_i$. Given an example $\langle x_1, \ldots, x_d \rangle$ for which we want to make a prediction, we would predict $f(\langle x_1, \ldots, x_d \rangle)$.

So to compute the best-fit hyperplane for our versicolor *labeling, we'll have to find our set of equa-*

tions. To do that, *we need to compute a lot of sums. Here we go.*

$$\begin{array}{lcll}
\sum_i x_{i1}x_{i1} & = & 1^2 + 1^2 + 1^2 + 1^2 + 1^2 & = & 5.0 \\
\sum_i x_{i1}x_{i2} & = & 1 \cdot 4.7 + 1 \cdot 6.1 + 1 \cdot 5.6 + 1 \cdot 5.8 + 1 \cdot 6.5 & = & 28.7 \\
\sum_i x_{i1}x_{i3} & = & 1 \cdot 3.2 + 1 \cdot 2.8 + 1 \cdot 3.0 + 1 \cdot 2.7 + 1 \cdot 3.2 & = & 14.9 \\
\sum_i x_{i1}x_{i4} & = & 1 \cdot 1.3 + 1 \cdot 4.7 + 1 \cdot 4.1 + 1 \cdot 5.1 + 1 \cdot 5.1 & = & 20.3 \\
\sum_i x_{i1}x_{i5} & = & 1 \cdot 0.2 + 1 \cdot 1.2 + 1 \cdot 1.3 + 1 \cdot 1.9 + 1 \cdot 2.0 & = & 6.6 \\
\sum_i x_{i1}y_i & = & 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 & = & 2.0 \\
\sum_i x_{i2}x_{i2} & = & 4.7^2 + 6.1^2 + 5.6^2 + 5.8^2 + 6.5^2 & = & 166.55 \\
\sum_i x_{i2}x_{i3} & = & 4.7 \cdot 3.2 + 6.1 \cdot 2.8 + 5.6 \cdot 3.0 + 5.8 \cdot 2.7 + 6.5 \cdot 3.2 & = & 85.38 \\
\sum_i x_{i2}x_{i4} & = & 4.7 \cdot 1.3 + 6.1 \cdot 4.7 + 5.6 \cdot 4.1 + 5.8 \cdot 5.1 + 6.5 \cdot 5.1 & = & 120.47 \\
\sum_i x_{i2}x_{i5} & = & 4.7 \cdot 0.2 + 6.1 \cdot 1.2 + 5.6 \cdot 1.3 + 5.8 \cdot 1.9 + 6.5 \cdot 2.0 & = & 39.56 \\
\sum_i x_{i2}y_i & = & 4.7 \cdot 0 + 6.1 \cdot 1 + 5.6 \cdot 1 + 5.8 \cdot 0 + 6.5 \cdot 0 & = & 11.7 \\
\sum_i x_{i3}x_{i3} & = & 3.2^2 + 2.8^2 + 3.0^2 + 2.7^2 + 3.2^2 & = & 44.61 \\
\sum_i x_{i3}x_{i4} & = & 3.2 \cdot 1.3 + 2.8 \cdot 4.7 + 3.0 \cdot 4.1 + 2.7 \cdot 5.1 + 3.2 \cdot 5.1 & = & 59.71 \\
\sum_i x_{i3}x_{i5} & = & 3.2 \cdot 0.2 + 2.8 \cdot 1.2 + 3.0 \cdot 1.3 + 2.7 \cdot 1.9 + 3.2 \cdot 2.0 & = & 19.43 \\
\sum_i x_{i3}y_i & = & 3.2 \cdot 0 + 2.8 \cdot 1 + 3.0 \cdot 1 + 2.7 \cdot 0 + 3.2 \cdot 0 & = & 5.8 \\
\sum_i x_{i4}x_{i4} & = & 1.3^2 + 4.7^2 + 4.1^2 + 5.1^2 + 5.1^2 & = & 92.61 \\
\sum_i x_{i4}x_{i5} & = & 1.3 \cdot 0.2 + 4.7 \cdot 1.2 + 4.1 \cdot 1.3 + 5.1 \cdot 1.9 + 5.1 \cdot 2.0 & = & 31.12 \\
\sum_i x_{i4}y_i & = & 1.3 \cdot 0 + 4.7 \cdot 1 + 4.1 \cdot 1 + 5.1 \cdot 0 + 5.1 \cdot 0 & = & 8.8 \\
\sum_i x_{i5}x_{i5} & = & 0.2^2 + 1.2^2 + 1.3^2 + 1.9^2 + 2.0^2 & = & 10.78 \\
\sum_i x_{i5}y_i & = & 0.2 \cdot 0 + 1.2 \cdot 1 + 1.3 \cdot 1 + 1.9 \cdot 0 + 2.0 \cdot 0 & = & 2.5 \\
\sum_i x_{i6}y_i & = & 0^2 + 1^2 + 1^2 + 0^2 + 0^2 & = & 2.0
\end{array}$$

From these we derive a set of five equations with five unknowns.

$$\begin{array}{rcrcrcrcrcl}
5.0m_1 & + & 28.70m_2 & + & 14.90m_3 & + & 20.30m_4 & + & 5.50m_5 & = & 2.0 \\
28.7m_1 & + & 166.55m_2 & + & 85.38m_3 & + & 120.47m_4 & + & 39.56m_5 & = & 11.7 \\
14.9m_1 & + & 85.38m_2 & + & 44.61m_3 & + & 59.71m_4 & + & 19.43m_5 & = & 5.8 \\
20.3m_1 & + & 120.47m_2 & + & 59.71m_3 & + & 92.61m_4 & + & 31.12m_5 & = & 8.8 \\
5.5m_1 & + & 39.56m_2 & + & 19.43m_3 & + & 31.12m_4 & + & 10.78m_5 & = & 2.5
\end{array}$$

*Now we solve this to get the hyperplane hypothesis. (*You *can try to do it by hand, but at this point I broke down and went to a computer.) The answer turns out to be the following.*

$$f(x_1, x_2, x_3, x_4, x_5) = 0.308x_1 + 0.736x_2 - 1.00x_3 - 0.278x_4 - 0.020x_5$$

For Iris Q, the predicted answer is

$$0.308 \cdot 1 + 0.736 \cdot 5.8 - 1.00 \cdot 2.7 - 0.278 \cdot 3.9 - 0.020 \cdot 1.2 = 0.767 \; .$$

Thus linear regression indicates that we are fairly confident that Iris Q is versicolor *(as indeed it is).*

Analysis Linear regression is really best suited for problems where the attributes and labels are all numeric and there is reason to expect that a linear function will approximate the problem. This is rarely a reasonable expectation — linear functions are just too restricted to represent a wide variety of hypotheses.
Advantages:

- Has real mathematical rigor.

- Handles irrelevant attributes somewhat well. (Irrelevant attributes tend to get a coefficient close to zero in the hyperplane's equation.)
- Hypothesis function is easy to understand.

Disadvantages:

- Oversimplifies the classification rule. (Why should we expect a hyperplane to be a good approximation?)
- Difficult to compute.
- Limited to numeric attributes.
- Doesn't at all represent how humans learn.

1.2.2 Nearest neighbor

Our second approach originates in the idea that given a query, the training example that is most similar to it probably has the same label. To determine what we mean by most "similar", we have to design some sort of **distance function**. In many cases, the best choice would be the **Euclidean distance function**, the straight-line distance between the two points $\langle x_{i1}, x_{i2}, \ldots, x_{id} \rangle$ and $\langle x_{j1}, x_{j2}, \ldots, x_{jd} \rangle$ in d-dimensional space. The formula for this would be

$$\sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + \cdots + (x_{id} - x_{jd})^2} \, .$$

Another common possibility is to use the **Manhattan distance function**,

$$|x_{i1} - x_{j1}| + |x_{i2} - x_{j2}| + \cdots + |x_{id} - x_{jd}| \ .$$

In practice, people generally go for the Euclidean distance function. They don't really have much mathematical reasoning to back this up; it just seems to work well in general.

In the Iris example, we'll compute the Euclidean distance from each of the vectors for the training examples to the query vector representing Iris Q.

example	*distance*		*label*
Iris A	$\sqrt{(4.7-5.8)^2 + (3.2-2.7)^2 + (1.3-3.9)^2 + (0.2-1.2)^2}$	$= 3.04$	setosa
Iris B	$\sqrt{(6.1-5.8)^2 + (2.8-2.7)^2 + (4.7-3.9)^2 + (1.2-1.2)^2}$	$= 0.86$	versicolor
Iris C	$\sqrt{(5.6-5.8)^2 + (3.0-2.7)^2 + (4.1-3.9)^2 + (1.3-1.2)^2}$	$= 0.42$	versicolor
Iris D	$\sqrt{(5.8-5.8)^2 + (2.7-2.7)^2 + (5.1-3.9)^2 + (1.9-1.2)^2}$	$= 1.39$	virginica
Iris E	$\sqrt{(6.5-5.8)^2 + (3.2-2.7)^2 + (5.1-3.9)^2 + (2.0-1.2)^2}$	$= 1.68$	virginica

Since Iris C is the closest, the nearest-neighbor algorithm would predict that Iris Q has the same label as Iris C: versicolor.

There are two common refinements to the nearest-neighbor algorithm to address common issues with the data. The first is that the the raw Euclidean distance overemphasizes attributes that have broader ranges. In the Iris example, the petal length is given more emphasis than the petal width (since the length varies up to four centimeters, while the width only varies up to two centimeters). This is easy to fix by scaling each attribute by the maximum difference in the attribute values, to ensure that the distance between two attribute values in the same attribute never exceeds 1.

$$v_{ij} = \frac{x_{ij}}{\max_k x_{kj} - \min_k x_{kj}}$$

By applying the distance function to the v_{ij} instead of the x_{ij}, this artificial overemphasis disappears.

The second common refinement is to address the issue of **noise** — typically, a small but unknown fraction of the data might be mislabeled or unrepresentative. In the iris example, Iris C may not *really* be *versicolor*. Or perhaps Iris C's dimensions are uncharacteristic of *versicolor* specimens. We can get around this using the law of large numbers: Select some number k and use the k nearest neighbors. The prediction can be the average of these k nearest neighbors if the label values are numeric, or the plurality if the label values are discrete (breaking ties by choosing the closest).

> *If we choose k to be 3, we find that the three closest irises to Iris Q are Irises B, C, and D, of which two are* versicolor *and one is* virginica*. Therefore we still predict that Iris Q is* versicolor*.*

Analysis The nearest-neighbor algorithm and its variants are particularly well-suited to **collaborative filtering**, where a system is to predict a given person's preference based on other people's preferences. For example, a movie Web site might ask you to rate some movies and then try to find movies you'd like to see. Here, each attribute is a single movie that you have seen, and the Web site looks for people whose movie preferences are close to yours and then predicts movies that these neighbors liked but that you have not seen. Or you might see this on book-shopping sites, where the site makes book recommendations based on your past order history.

Collaborative filtering fits into the nearest-neighbor search well because attributes tend to be numeric and similar in nature, so it makes sense to give them equal weight in the distance computation.

Advantages:

- Represents complex spaces very well.
- Easy to compute.

Disadvantages:

- Does not handle many irrelevant attributes well. If we have lots of irrelevant attributes, the distance between examples is dominated by the differences in these irrelevant attributes and so becomes meaningless.
- Still doesn't look much like how humans learn.
- Hypothesis function is too complex to describe easily.

1.2.3 ID3

Linear regression and nearest-neighbor search don't work very well when the data is discrete — they're really designed for numeric data. **ID3** is designed with discrete data in mind.

ID3's goal is to generate a decision tree that seems to describe the data. Figure 1.4 illustrates one decision tree. Given a vector, the decision tree predicts a label. To get its prediction, we start at the top and work downward. Say we take Plant Q. We start at the top node. Since Plant Q's stem is normal, we go to the right. Now we look for mold on Plant Q's seeds, and we find that it has some, so we go to the left from there. Finally we examine the spots on Plant Q's leaves; since they don't have yellow halos, we go to the right and conclude that Plant Q must have downy mildew.

Decision trees are well-suited for discrete data. They represent a good compromise between simplicity and complexity. Recall that one of our primary complaints about linear regression was that its hypothesis was too constricted to represent very many types of data, while one of our primary complaints about nearest-neighbor search was that its hypothesis was too complex to be understandable. Decision trees are easy to interpret but can represent a wide variety of functions.

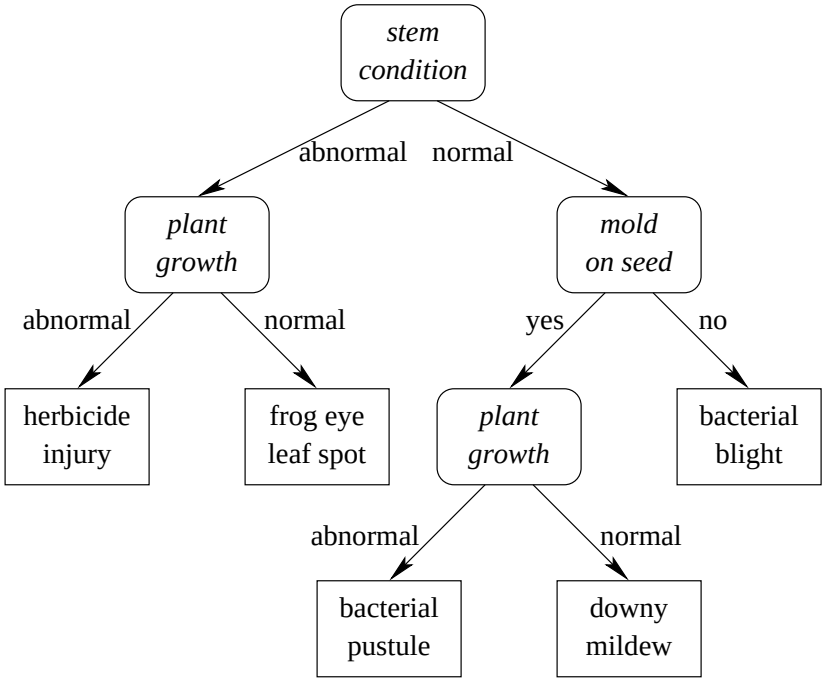
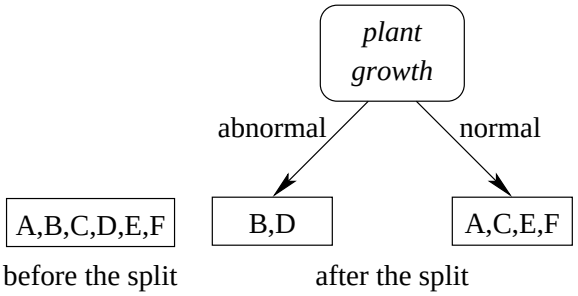


Figure 1.4: A decision tree example

Given a set of data, the goal of ID3 is to find a "good" decision tree that represents it. By *good*, the general goal is to find a small decision tree that approximates the true label pretty well.

Constructing the tree automatically ID3 follows a simple technique for constructing such a decision tree. We begin with a single node containing all the training data. Then we continue the following process: We find some node containing data with different labels, and we *split* it based on some attribute we select. By *split*, I mean that we take the node and replace it with new nodes for each possible value of the chosen attribute. For example, with the soybean data, if we have a node containing all the examples and choose to split on plant growth, the effect would be as follows.



We stop splitting nodes when every node of the tree is either labeled unanimously or contains indistinguishable vectors.

How does ID3 decide on which attribute to split a node? Before we answer this, we need to define the entropy of a set of examples. The **entropy** of a set S is defined by the following equation.

$$Entropy(S) = \sum_{\text{labels } \ell} -p_\ell \ln p_\ell$$

Here p_ℓ is the fraction of points in the set with the label ℓ.

In the training data of Figure 1.1, there are five labels: $2/6$ *of the examples have frog eye leaf spot, and* $1/6$ *of the examples have each of the four other diseases. So the entropy is*

$$-\frac{2}{6}\ln\frac{2}{6} + -\frac{1}{6}\ln\frac{1}{6} + -\frac{1}{6}\ln\frac{1}{6} + -\frac{1}{6}\ln\frac{1}{6} + -\frac{1}{6}\ln\frac{1}{6} = 1.5607\ .$$

This entropy is rather large, quantifying the fact that the set isn't labeled consistently at all.

The entropy is a weird quantity that people use more or less just because it works. When the entropy is small, this indicates that things are labeled pretty consistently. As an extreme example, if everything has the same label ℓ, then since $p_\ell = 1$ the entropy would be just $-p_\ell \ln p_\ell = 0$. We're aiming for a small decision tree where the total entropy of all the leaves is as small as possible.

Now let's look at what happens when we split a node. This splits the set of examples it represents into pieces based on the values of the attribute. To quantify how good the split is, we define the **gain** of splitting a node on a particular attribute: It is the entropy of the old set it represented, minus the weighted sum of the entropies of the new sets. Say S represents the old set, and S_v represents the examples of S with the value v for the attribute under consideration. The gain would be

$$Entropy(S) - \sum_{\text{values } v} \frac{|S_v|}{|S|} Entropy(S_v)\ .$$

At the beginning of the algorithm, all the examples are in a single node. Let's consider splitting on the plant growth. We just computed the entropy of all six training examples to be 1.5607*. After we split on plant growth, we get two sets of plants: B and D have abnormal plant growth (this set has entropy* $-(1/2)\ln(1/2) + -(1/2)\ln(1/2) = 0.6931$*); and A, C, E, and F have normal plant growth (this set has entropy* 1.0397*). So the gain of splitting on plant growth is*

$$1.5607 - \left(\frac{2}{6}0.6931 + \frac{4}{6}1.0397\right) = 0.6365\ .$$

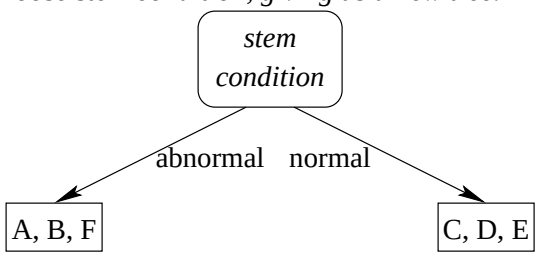
We can do similar calculations to compute the gain for stem condition instead. This divides the plants into a set of A, B, and F (entropy 0.6365*) and a set of C, D, and E (entropy* 1.0986*). The gain of splitting on stem condition is*

$$1.5607 - \left(\frac{3}{6}0.6365 + \frac{3}{6}1.0986\right) = 0.6931$$

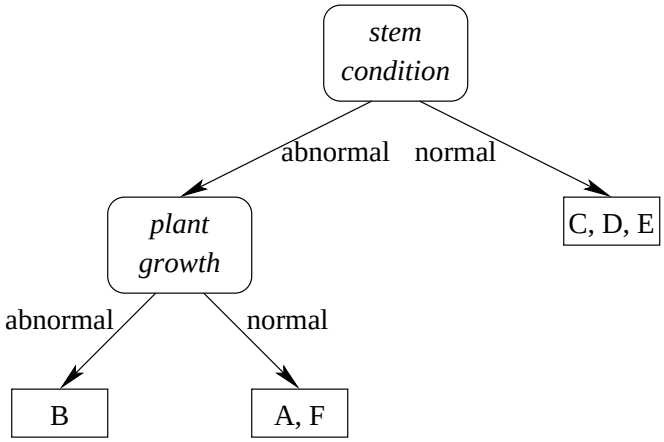
This is larger than the gain for plant growth, so splitting on stem condition is preferable.

After computing the gains of splitting for each of the attributes, we choose the attribute that gives the largest gain and then split on it, giving us a new tree. We continue splitting nodes until the examples in every node either have identical labels or indistinguishable attributes.

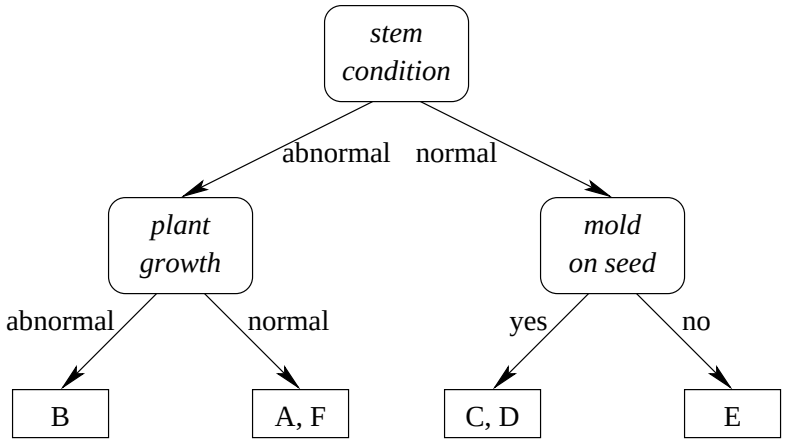
We would also consider splitting on leaf-spot halos (gain of 0.6931) and splitting on seed mold (gain of 0.6367). Of these, we could go with either stem condition or leaf-spot halos: They both have the same gain, 0.6931. We'll choose stem condition, giving us a new tree.



We repeat the process for each of the remaining sets. We first consider Plants A, B, and F. They don't have the same disease, so we'll look for an attribute on which to split them. That turns out to be easy, as they disagree only on the plant-growth attribute (none of them have leaf-spot halos or seed mold), so we'll split that node based on plant growth. (The gain of this would be 0.6365, while splitting on leaf-spot halos or seed mold gives a gain of 0.) Now we have the following tree.



The set of A and F isn't a problem, as they both have the same disease. So the next set we'll consider for splitting is C, D, and E. There, the gain for splitting based on plant growth is $1.0986 - ((2/3)0.6931 + (1/3)0) = 0.6365$. *The gain for splitting on leaf-spot halos is 0, since they all have leaf-spot halos. The gain for splitting on seed mold is* $1.0986 - ((2/3)0.6931 + (1/3)0) = 0.6365$. *We could go for either plant growth or seed mold; say we choose seed mold.*



Finally, we consider how to split Plants C and D. Splitting on plant growth gives a gain of 0.6931, while splitting on leaf-spot halos gives a gain of 0. We must split on plant growth, giving us the tree of Figure 1.4.

Analysis Advantages:

- Represents complex spaces well.
- Generates a simple, meaningful hypothesis.
- Filters out irrelevant data.

Disadvantages:

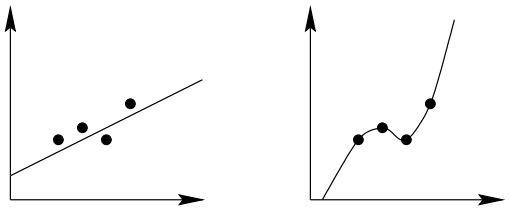


Figure 1.5: Overfitting some data.

- Doesn't handle numeric attributes well.
- Still doesn't look much like how humans learn.

1.3 General issues

There are a number of overarching data mining issues that apply regardless of the technique. The importance of these issues makes them worth studying separately. We'll look at just a few.

Data selection Choosing the data is an important element of applying data mining techniques. Of course we want the data to have integrity — though there's a natural tradeoff between integrity and quantity that we have to balance. We certainly need a good sample of data, or else the learned hypothesis won't be worth much. But we also need a large sample to work from.

Less obvious is the importance of a selecting good attributes. Sometimes we need to do some preprocessing to get good data. For example, with loan applications, the applicant's date of birth is probably what is really stored on the computer, but really this isn't significant to the application (neglecting astrological effects) — what's important is the applicant's age. So, though the database probably holds the birthday, we should compute the age to give to the learning algorithm. We can see a similar thing happening in the soybean data (Figure 1.1): Instead of giving the plant height (which after all, coupled with the month the plant was found implies whether growth is stunted), the data set just includes a feature saying whether plant growth is normal or abnormal. The researchers here did some preprocessing to simplify the task for the learner.

Overfitting In applying a machine learning technique, we need to be careful to avoid **overfitting**. This occurs when the algorithm adapts very carefully to the specific training data without improving general performance. For example, consider the two graphs in Figure 1.5. Although the graph at the right fits the data perfectly, it's likely that the graph at left is a better hypothesis.

Overfitting applies to just about any learning algorithm. ID3 is particularly prone to overfitting, as it continues growing the tree until it fits the data perfectly. Machine learning researchers have ways of working around this, but they get rather complicated, and so we're choosing to skip their approaches.

There's a tradeoff: Do we go for the perfect fit (which may be an overfit), or do we settle for a simpler hypothesis that seems to be pretty close? The answer is that this is part of the art of applying machine learning techniques to data mining. But an aid to this is to be able to evaluate the error of a given hypothesis.

Evaluating hypotheses Once we get a hypothesis from a learning algorithm, we often need to get some sort of estimate of how good it is. The most typical measure is the **error**: the probability that the hypothesis predicts the wrong label for a randomly chosen new example.

It's very tempting to feed all the data we have into the learning algorithm, but if we need to report the hypothesis error, this is a mistake. We also need to use some data to compute the error, and we can't reuse the training data for this computation. (This would be analogous to giving students all the answers to a test the day before: Of course they'll do well on the test, but what have they learned?)

So in situations where we need to compute the error, we separate the data into two sets — the **training set**, which holds the examples we give to the learning algorithm — and the **test set**, which we use for computing the error. The error we report would be the fraction of the examples in the test set on which the learning algorithm's hypothesis predicts wrongly. Typically two-thirds of the data might go into the training set and a third into the test set, to give a reasonable tradeoff on the accuracy of the hypothesis and the accuracy of the reported error.

In many situations, the data just isn't plentiful enough to allow this. U S Presidential elections would be a good example: It's not as if we can go out and generate new elections, so we're stuck with the handful we have. If we want to apply a learning algorithm to past polling data and their effect on the final result, we want to use all the data we can find. Machine learning has proposed several techniques for both using all the data and getting a close estimate of the error, but they're beyond the scope of this survey.

Ethics Obviously any time you deal with personal information, ethical considerations arise. Databases often include sensitive information that shouldn't be released. Social Security numbers are just one of several pieces of data that one can use to gain access to a person's identity, which you don't want to get around *too* much.

Less obviously, data miners also need to be careful to avoid discrimination. For example, a bank that intends to use data mining to determine whether to approve loans should think twice before including race or gender as one of the attributes given to the learning algorithm. Even zip codes should probably not go into the learning algorithm, as implicit discrimination can arise due to communities with a particularly high density of one race.

For these applications, the data mining practitioner should review the generated hypothesis to look for unethical or illegal discrimination. Algorithms that generate meaningful hypothesis (like linear regression or ID3, but not nearest-neighbor search) are particularly useful for such applications that need human review at the end.

1.4 Conclusion

We have seen a sample of data mining techniques — linear regression, nearest-neighbor search, and ID3 — and other issues that data mining practitioners must heed. The topic of data mining is rich and just now becoming a widely applied field. We could easily spend a complete semester studying it — it involves many interesting applications of mathematics to this goal of data analysis. I'd personally love to spend more time on it — but I also know what's to come, and it's every bit as intriguing!

Chapter 2

Neural networks

One approach to learning is to try to simulate the human brain in the computer. To do this, we need to know roughly how a brain works.

A brain neuron (see Figure 2.1) is a cell with two significant pieces: some **dendrites**, which can receive impulses, and an **axon**, which can send impulses. When the dendrites receive enough impulses, the neuron becomes excited and sends an impulse down the axon. In the brain, this axon is next to other neurons' dendrites. These neurons receive the impulse and may themselves become excited, propagating the signal further. This connection between an axon and a dendrite is a **synapse**.

Over time, the connections between axons and dendrites change, and so the brain "learns" as the conditions under which neurons become excited change. How exactly neurons map onto concepts, and how the changing of synapses represents an actual change in knowledge, remains a very difficult question for psychologists and biologists.

2.1 Perceptron

Using what we know about a neuron, though, it's easy to build a mechanism that approximates one. The simplest attempt at this is called the **perceptron**. Although the perceptron isn't too useful as a learning technique on its own, it's worth studying due to its role in artificial neural networks (ANNs), which we'll investigate in Section 2.2.

2.1.1 The perceptron algorithm

The perceptron has a number of inputs corresponding to the axons of other neurons. We'll call these inputs x_i for $i = 1, \ldots, n$ (where n is the number of inputs). It also has a weight w_i for each input (corresponding to the synapses). It becomes excited whenever

$$\sum_{i=1}^{n} w_i x_i > 0 \, .$$

When it is excited, it outputs 1, and at other times it outputs -1. The perceptron can output only these two values.

Let us return to classifying irises. As with linear regression, we'll add a constant-one attribute to give added flexibility to the hypothesis. We'll also make the label numeric by labeling an example 1 *if it is* versicolor *and* -1 *if it is not. Figure 2.2 contains the data we'll use.*

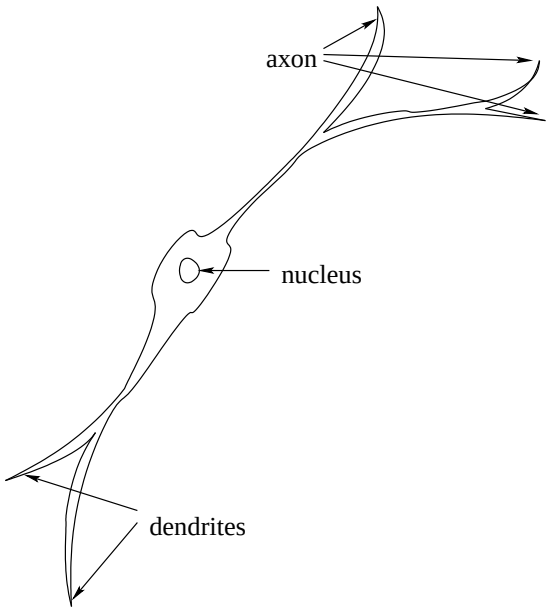


Figure 2.1: A depiction of a human neuron.

	constant one	sepal length	sepal width	petal length	petal width	is species *versicolor*
Iris A	1	4.7	3.2	1.3	0.2	−1
Iris B	1	6.1	2.8	4.7	1.2	1
Iris C	1	5.6	3.0	4.1	1.3	1
Iris D	1	5.8	2.7	5.1	1.9	−1
Iris E	1	6.5	3.2	5.1	2.0	−1

Figure 2.2: A prediction-from-examples problem of Iris classification.

We'll choose random weights with which to begin the perceptron. Say our perceptron begins with the weights $w_1 = 1$, $w_2 = 0$, $w_3 = 0$, $w_4 = 0$, $w_5 = 1$. *For the first prediction, we need to compute whether*

$$\sum_{i=1}^{5} w_i x_i = 1 \cdot 1 + 0 \cdot 4.7 + 0 \cdot 3.2 + 0 \cdot 1.3 + 1 \cdot 0.2 = 1.2$$

exceeds 0. *Since it does, the perceptron predicts that this example's label will be* 1.

To learn, a perceptron must adapt over time by changing its weights w_i over time to more accurately match what happens. A perceptron normally starts with random weights. But each time it makes a mistake by predicting 1 when the correct answer is -1, we change all weights as follows.

$$w_i \leftarrow w_i - r x_i$$

Here r represents the **learning rate**, which should be chosen to be some small number like 0.05. (If you choose r too large, the perceptron may erratically oscillate rather than settle down to something meaningful.)

In our example, we just predicted that Iris A's label would be 1 *when in fact the correct label is* -1 *(according to Figure 2.2). So we'll update the weights.*

$$\begin{aligned}
w_1 &\leftarrow w_1 - r x_1 = 1 - 0.05 \cdot 1 &&= 0.95 \\
w_2 &\leftarrow w_2 - r x_2 = 0 - 0.05 \cdot 4.7 &&= -0.24 \\
w_3 &\leftarrow w_3 - r x_3 = 0 - 0.05 \cdot 3.2 &&= -0.16 \\
w_4 &\leftarrow w_4 - r x_4 = 0 - 0.05 \cdot 1.3 &&= -0.07 \\
w_5 &\leftarrow w_5 - r x_5 = 1 - 0.05 \cdot 0.2 &&= 0.99
\end{aligned}$$

These are the weights we'll use for the next training example.

Similarly, if the perceptron predicts -1 when the answer is 1, we change the weights again.

$$w_i \leftarrow w_i + r x_i$$

To compute the prediction for Iris B, we determine whether

$$\sum_{i=1}^{5} w_i x_i = 0.95 \cdot 1 + (-0.24) \cdot 6.1 + (-0.16) \cdot 2.8 + (-0.07) \cdot 4.7 + 0.99 \cdot 1.2 = -0.05$$

exceeds 0. *Since it does not, the perceptron predicts that Iris B is labeled* -1.

This is wrong: According to Figure 2.2, the correct label for Iris B is 1. *So we'll update the weights.*

$$\begin{aligned}
w_1 &\leftarrow w_1 + r x_1 = 0.95 + 0.05 \cdot 1 &&= 1.00 \\
w_2 &\leftarrow w_2 + r x_2 = -0.24 + 0.05 \cdot 6.1 &&= 0.07 \\
w_3 &\leftarrow w_3 + r x_3 = -0.16 + 0.05 \cdot 2.8 &&= -0.02 \\
w_4 &\leftarrow w_4 + r x_4 = -0.07 + 0.05 \cdot 4.7 &&= 0.17 \\
w_5 &\leftarrow w_5 + r x_5 = 0.99 + 0.05 \cdot 1.2 &&= 1.05
\end{aligned}$$

These are the weights we'll use for the next training example.

Let me make an intuitive argument for why this training rule makes sense. Say we get an example where the perceptron predicts 1 when the answer is -1. In this case, each input contributed $w_i x_i$ to a total that ended up being positive (and so the perceptron got improperly excited). After the training, the new weight is $w_i - r x_i$, and so if the perceptron saw the same example again, this input would contribute $(w_i - r x_i) x_i = w_i x_i - r x_i^2$. Since r is positive, and since x_i^2 must be positive regardless of x_i's value,

this contribution is smaller than before. Thus if we repeat the same example immediately, the sum will be smaller than before — and hence closer to being labeled -1 correctly. (The same line of argument applies when the perceptron predicts -1 when the answer is 1.)

The perceptron algorithm will iterate over and over through the training examples until either it predicts all training examples correctly or until somebody decides it's time to stop.

To train the perceptron on the irises, we'd go through the examples several times. The following table demonstrates how the weights change while going through all the examples three times over.

iris	$\sum_i w_i x_i$	w_1	w_2	w_3	w_4	w_5
A	1.20	0.95	−0.24	−0.16	−0.07	0.99
B	−0.05	1.00	0.07	−0.02	0.17	1.05
C	3.39	1.00	0.07	−0.02	0.17	1.05
D	4.21	0.95	−0.22	−0.16	−0.08	0.96
E	0.50	0.90	−0.55	−0.32	−0.34	0.86
A	−2.94	0.90	−0.55	−0.32	−0.34	0.86
B	−3.88	0.95	−0.24	−0.18	−0.10	0.92
C	−0.16	1.00	0.04	−0.03	0.10	0.98
D	3.54	0.95	−0.25	−0.16	−0.15	0.89
E	−0.21	0.95	−0.25	−0.16	−0.15	0.89
A	−0.76	0.95	−0.25	−0.16	−0.15	0.89
B	−0.69	1.00	0.05	−0.02	0.08	0.95
C	2.80	1.00	0.05	−0.02	0.08	0.95
D	3.47	0.95	−0.24	−0.16	−0.17	0.85
E	−0.27	0.95	−0.24	−0.16	−0.17	0.85

At first glance, this looks quite nice. On the first pass through the examples, the perceptron classified only 1 *of the* 5 *flowers correctly. On the second pass, it got* 2 *correct. And on the third pass, it got* 3 *correct.*

If you continue to see how it improves, though, the perceptron doesn't label 4 *of the flowers correct until* 34 *times through the training set. It finally gets all* 5 *flowers right on the* $1,358$*th iteration through the examples. (The training rate doesn't affect this too much: Increasing the training rate even quite a bit doesn't speed it up much, and decreasing the rate only slows it a little.)*

Incidentally, after going through all these iterations using $r = 0.05$*, the final weights would be* $w_1 = -1.80$*,* $w_2 = -0.30$*,* $w_3 = -0.19$*,* $w_4 = 4.65$*, and* $w_5 = -11.56$*.*

2.1.2 Analysis

Although the approaches are very different, it's instructive to compare linear regression with the perceptron. Their prediction techniques are identical: They have a set of weights, and they predict according to whether the weighted sum of the attributes exceeds a threshold.

We know that linear regression has a strong mathematical foundation, and we know that the perceptron hypothesis isn't any more powerful than that used by linear regression. So why would you ever use a perceptron instead?

You wouldn't. I don't know of any reason to use a single-perceptron predictor, when you could just as easily do linear regression. Perceptrons are easier to program, sure, and easier to understand. But they take a lot more computation to get the same answer, if you're lucky enough to get an answer. (Perceptrons aren't even guaranteed to converge to a single answer, unless there's a hyperplane that separates the data perfectly.)

So what's the point of studying perceptrons? They're inspired by human biology, and the human brain is the best learning device known to humankind. But we need to keep in mind that, though the human brain

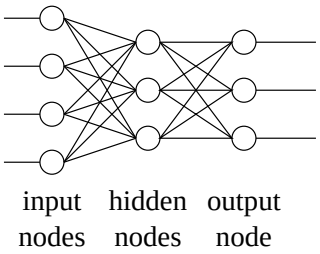


Figure 2.3: An artificial neural network.

is extraordinarily powerful, each individual neuron is relatively worthless. To get good performance, we're going to have to network neurons together. That's what we're going to look at next, and that's when we'll start to see some dividends from our study of perceptrons.

Advantages:

- Simple to understand.
- Generates a simple, meaningful hypothesis.
- Inspired by human biology.
- Adapts to new data simply.

Disadvantages:

- The hypothesis is a simple linear combination of inputs, just like linear regression, but linear regression has much more solid mathematical grounds.
- Only works for predicting yes/no values.

2.2 Artificial neural networks

Artificial neural networks (ANNs) are relatively complex learning devices. We'll look first at the overall architecture, then at the individual neurons, and finally at how the network predicts and adapts for training examples.

2.2.1 ANN architecture

Figure 2.3 pictures the layout of one ANN. This particular network has three layers. The first is the **input layer**, including four nodes in Figure 2.3. These nodes aren't really neurons of the network — they just feed the attributes of an example to the neurons of the next layer. (We'll have four inputs when we look at the irises, as each iris has four attributes.)

The next layer is the **hidden layer**. This layer has several neurons that should adapt to the input. These neurons are meant to process the inputs into something more useful, like detecting particular features of a picture if the inputs represent the pixels of a picture — but they'll automatically adapt, so what they detect isn't necessarily meaningful. Choosing the right number of neurons for this layer is an art. Figure 2.3 includes three hidden neurons in the hidden layer, with every input node connected to every neuron, but really an ANN could have any number of hidden neurons in any configuration.

The final layer is the **output layer**. It has an output neuron for each output that the ANN should produce.

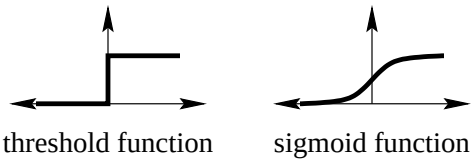
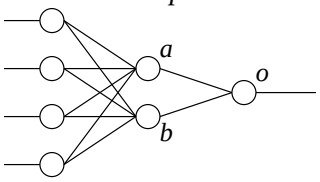


Figure 2.4: The threshold function and the sigmoid function.

One could conceivably have more hidden layers, or delete or add links different from those diagrammed in Figure 2.3, or even use a more complex unlayered design. But the most common design is the **two-layer network**, like that of Figure 2.3. In this architecture, there is a layer of some number of hidden neurons, followed by a layer of some number of output neurons. It has each input connected to each hidden neuron, and it has each hidden neuron connected to each output neuron. People usually use two-layer networks because more complex designs just haven't proven useful in many situations.

In our walk-through example of how the ANN works, we'll use the following very simple two-layer network with two hidden units a and b and one output unit o.



As is commonly done with ANNs, each of the neurons in our network has an additional constant-one input. We do this instead of adding a new constant-1 *attribute (as we did with linear regression and perceptrons), because that wouldn't give a constant-one input into the output unit o.*

2.2.2 The sigmoid unit

Each unit of the network is to resemble a neuron. You might hope to use perceptrons, but in fact that doesn't work very well. The problem is one of feedback: In a neural network, we must sometimes attribute errors in the ANN prediction to mistakes by the hidden neurons, if the hidden neurons are to change behavior over time at all. (If they don't change behavior, there isn't much point in having them.) Researchers just haven't found a good way of doing this with regular perceptrons.

But researchers *have* figured out a practical way to do this attribution of error using a slight modification of the perceptron, called a **sigmoid unit**. Sigmoid units produce an output between 0 and 1, using a slightly different procedure from before. (Having the low output being 0 is just a minor difference from the perceptron, whose low output is -1. We'll just rework our training example labels by replacing -1 labels with 0.)

A sigmoid unit still has a weight w_i for each input x_i, but it processes the weighted sum slightly differently, using the **sigmoid function**, defined as

$$\sigma(y) = \frac{1}{1 + e^{-y}} \, .$$

This is a smoothed approximation to the threshold function used by perceptrons, as Figure 2.4 illustrates.

To make a prediction given the inputs x_i, the sigmoid unit computes and outputs the value

$$\sigma\left(\sum_i w_i x_i\right) \, .$$

Notice that this means it will output some number *between* 0 and 1. It will never actually be 0 or 1, but it can get very close.

The difference between a sigmoid unit and a perceptron is quite small. The only real reason to use the sigmoid unit is so that the mathematics behind the analysis — which we won't examine — works out to show that the neural network will approach some solution. The advantage of a sigmoid unit is that its behavior is smooth and never flat. This makes mathematical analysis easier, since it means we can always improve the situation by climbing one way or the other along the sigmoid curve. If the output is too high, we'll try to go downhill a bit. Too small? Go uphill a bit. But we won't get into the details of why it works.

We have three sigmoid units in our architecture: a, b, and o. The hidden units a and b have five inputs and hence five weights each — one from each input node, plus the constant-one input. The output unit o has three inputs and hence three weights — one from each hidden unit, plus the constant-one input. To begin the network, we initialize each of these weights using small random values as follows.

$w_{0a} = 0.0$	*weight of constant-one input into hidden unit a*
$w_{1a} = 0.1$	*weight for hidden unit a from first input node*
$w_{2a} = -0.1$	*weight for hidden unit a from second input node*
$w_{3a} = -0.1$	*weight for hidden unit a from third input node*
$w_{4a} = 0.0$	*weight for hidden unit a from fourth input node*
$w_{0b} = 0.0$	*weight of constant-one input into hidden unit b*
$w_{1b} = -0.1$	*weight for hidden unit b from first input node*
$w_{2b} = 0.2$	*weight for hidden unit b from second input node*
$w_{3b} = 0.1$	*weight for hidden unit b from third input node*
$w_{4b} = -0.1$	*weight for hidden unit b from fourth input node*
$w_{0o} = 0.1$	*weight of constant-one input into output unit o*
$w_{ao} = 0.2$	*weight for output unit o from hidden unit a*
$w_{bo} = -0.1$	*weight for output unit o from hidden unit b*

2.2.3 Prediction and training

Handling a single training example is a three-step process. We'll see how to do each of these steps in detail soon, but here's the overview.

1. We run the training example through the network to see how it behaves (the *prediction step*).

2. We assign an "error" to each sigmoid unit in the network (the *error attribution step*).

3. We update all the weights of the network (the *weight update step*).

The whole process is akin to the perceptron training process, except here we'll *always* update the weights. (Recall that the perceptron weights got updated only the perceptron erred.)

Prediction step In a two-layer network like that of Figure 2.3, the prediction step is straightforward: We take the example attributes, feed them into the hidden sigmoid units to get those units' output values, and then we feed these hidden units' outputs to the output units. The output units' outputs are the ANN's prediction.

Given Iris A, we first compute the output of hidden node a.

$$\begin{aligned} o_a &= \sigma(w_{0a} + w_{1a}x_1 + w_{2a}x_2 + w_{3a}x_3 + w_{4a}x_4) \\ &= \sigma(0.0 + 0.1 \cdot 4.7 + (-0.1) \cdot 3.2 + (-0.1) \cdot 1.3 + 0.0 \cdot 0.2) \\ &= \sigma(0.02) = \frac{1}{1 + e^{-0.02}} = 0.5050 \end{aligned}$$

Similarly, we compute the output of hidden node b.

$$\begin{aligned} o_b &= \sigma(w_{0b} + w_{1b}x_1 + w_{2b}x_2 + w_{3b}x_3 + w_{4b}x_4) \\ &= \sigma(0.0 + (-0.1) \cdot 4.7 + 0.2 \cdot 3.2 + 0.1 \cdot 1.3 + (-0.1) \cdot 0.2) \\ &= \sigma(0.28) = \frac{1}{1 + e^{-0.28}} = 0.5695 \end{aligned}$$

Finally, we can compute the output of the output node o.

$$\begin{aligned} o_o &= \sigma(o_{0o} + w_{ao}o_a + w_{bo}o_b) = \sigma(0.1 + 0.2 \cdot 0.5050 + (-0.1) \cdot 0.5695) \\ &= \sigma(0.1440) = \frac{1}{1 + e^{-0.1440}} = 0.5359 \end{aligned}$$

Thus the computed prediction for Iris A is 0.5359.

Error attribution step The error attribution step uses a technique called **backpropagation**. What we'll do is to look at the entire network's output (made by the output layer) and determine the error of each output unit. Then we'll move backward and attribute errors to the units of the hidden layer.

To assign the error of an output unit, say the desired output for the unit is t_o, but the actual output made by the unit was o_o. We compute the error of that output unit as $o_o(1 - o_o)(t_o - o_o)$.

After we compute the errors of all outputs units, we backpropagate to the hidden layer. Consider a hidden unit h, whose weight connecting it to an output unit o is w_{ho}. Moreover, call h's output o_h. (This is the output that was sent forward to the output node when making a prediction.) The error that we'll attribute to the hidden node h from o is $o_h(1 - o_h)w_{ho}\delta_o$. (If there are multiple output units, we'll sum these errors over all outputs to get the total error attributed to h.)

In our example, the correct answer t_o was 0 *(Iris A is not* versicolor*), while the network output o_o was* 0.5359. *Thus the error of unit o (which we'll represent by δ_o) is*

$$\delta_o = o_o(1 - o_o)(t_o - o_o) = 0.5359(1 - 0.5359)(0 - .5359) = -0.1333 \; .$$

Now that we have errors attributed to the output layer, we can compute the error of the hidden units. We compute δ_a, the error attributed to hidden unit a.

$$\delta_a = o_a(1 - o_a)w_{ao}\delta_o = 0.5050(1 - 0.5050)0.2 \cdot (-0.1333) = -0.0067$$

And we compute the error δ_b of the hidden unit b.

$$\delta_b = o_b(1 - o_b)w_{bo}\delta_o = 0.5359(1 - 0.5359)(-0.1) \cdot (-0.1333) = 0.0033$$

Weight update step The last step of handling the training example is updating the weights. We update *all* weights as follows (both those going from inputs to hidden nodes, and those going from hidden nodes to output nodes).

Consider one input to a sigmoid unit in the ANN. Say x is the value being fed into the input during the prediction step, and δ is the error attributed during the error-attribution step to the sigmoid unit receiving the input. We'll add $r\delta x$ to the weight associated with this input.

We'll use a learning rate r *of* 0.1 *here. Here's how the weights are changed for this training example.*

$$\begin{aligned}
w_{0a} \leftarrow w_{0a} + r\delta_a 1 &= 0.0 + 0.1 \cdot (-0.0067) \cdot 1 &&= -0.0007 \\
w_{1a} \leftarrow w_{1a} + r\delta_a x_1 &= 0.1 + 0.1 \cdot (-0.0067) \cdot 4.7 &&= 0.0969 \\
w_{2a} \leftarrow w_{2a} + r\delta_a x_2 &= -0.1 + 0.1 \cdot (-0.0067) \cdot 3.2 &&= -0.1021 \\
w_{3a} \leftarrow w_{3a} + r\delta_a x_3 &= -0.1 + 0.1 \cdot (-0.0067) \cdot 1.2 &&= -0.1009 \\
w_{4a} \leftarrow w_{4a} + r\delta_a x_4 &= 0.0 + 0.1 \cdot (-0.0067) \cdot 0.2 &&= -0.0001 \\
w_{0b} \leftarrow w_{0b} + r\delta_b 1 &= 0.0 + 0.1 \cdot 0.0032 \cdot 1 &&= 0.0003 \\
w_{1b} \leftarrow w_{1b} + r\delta_b x_1 &= -0.1 + 0.1 \cdot 0.0032 \cdot 4.7 &&= -0.0985 \\
w_{2b} \leftarrow w_{2b} + r\delta_b x_2 &= 0.2 + 0.1 \cdot 0.0032 \cdot 3.2 &&= 0.2010 \\
w_{3b} \leftarrow w_{3b} + r\delta_b x_3 &= 0.1 + 0.1 \cdot 0.0032 \cdot 1.2 &&= 0.1004 \\
w_{4b} \leftarrow w_{4b} + r\delta_b x_4 &= -0.1 + 0.1 \cdot 0.0032 \cdot 0.2 &&= -0.0999 \\
w_{0o} \leftarrow w_{0o} + r\delta_o 1 &= 0.1 + 0.1 \cdot (-0.1333) \cdot 1 &&= 0.0867 \\
w_{ao} \leftarrow w_{ao} + r\delta_o o_a &= 0.2 + 0.1 \cdot (-0.1333) \cdot 0.5050 &&= 0.1933 \\
w_{bo} \leftarrow w_{bo} + r\delta_o o_b &= -0.1 + 0.1 \cdot (-0.1333) \cdot 0.5695 &&= -0.1076
\end{aligned}$$

These are the weights we'll use for our next training example.

Conclusion This is *all* just for a single training example. Like in training a perceptron, we'd do all of this for each of the examples in the training set. And we'd repeat it several times over. It's not the sort of thing you can do by hand, though a computer can do it pretty easily for small networks.

Just to demonstrate that we've made progress, let's see what the network would predict if we tried Iris A again. We propagate its attributes through the network.

$$\begin{aligned}
o_a &= \sigma(-0.0007 + 0.0969 \cdot 4.7 + (-0.1021) \cdot 3.2 + (-0.1009) \cdot 1.3 + (-0.0001) \cdot 0.2) \\
&= \sigma(-0.0032) = 0.4992 \\
o_b &= \sigma(0.0003 + (-0.0985) \cdot 4.7 + 0.2010 \cdot 3.2 + 0.1004 \cdot 1.3 + (-0.0999) \cdot 0.2) \\
&= \sigma(0.2911) = 0.5724 \\
o_o &= \sigma(0.0867 + 0.1933 \cdot 0.4992 + (-0.1076) \cdot 0.5724) \\
&= \sigma(0.1440) = 0.5304
\end{aligned}$$

Thus the computed prediction for Iris A is 0.5304*, closer to the correct answer of* 0 *than the previously predicted* 0.5359*. This provides some evidence that the ANN has learned* something *through this training process.*

2.2.4 Example

Computationally, backpropagated ANNs are so complex that it's difficult to get a strong handle on how they work. It's instructive to look at a more industrial-strength example to see how they might actually be used. Let's consider the full iris classification example of Fisher [Fis36].

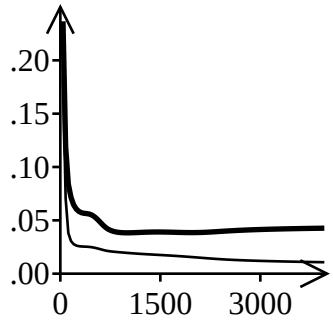


Figure 2.5: ANN error against iterations through training set. The thick line is the average error per evaluation example, and the thin line is the average error per training example.

Recall that this set of examples includes 150 irises, with each iris having four numeric attributes and a label classifying it among one of three species. Of the 150 irises, we choose a random subset of 100 as training examples and the remaining 50 for evaluation purposes.

We'll model this as a two-layer ANN. For each of the four numeric attributes, we'll include an input node in the input layer. In the hidden layer, we'll choose to use two sigmoid units. And the output layer will have a unit for each of the three possible labels. When we interpret the outputs, we'll find the output unit emitting the greatest output and interpret the ANN to be predicting the corresponding label.

We use a learning rate of 0.1. For training purposes, a label is encoded as $\langle 0.9, 0.1, 0.1 \rangle$ for *setosa*, $\langle 0.1, 0.9, 0.1 \rangle$ for *versicolor*, or $\langle 0.1, 0.1, 0.9 \rangle$ for *virginica*. We use 0.1 and 0.9 instead of 0 and 1 because a sigmoid unit never actually reaches an output of 0 or 1; if we encode labels using 0s and 1s, the weights will gradually become more and more extreme.

When we train, we go through the 100 training examples several times. It's interesting to see how the ANN improves as this continues. We'll look at the average sum-of-squares error: If the ANN outputs $\langle o_1, o_2, o_3 \rangle$ and the desired output is $\langle t_1, t_2, t_3 \rangle$, the sum-of-squares error is

$$\sum_{i=1}^{3} (t_i - o_i)^2 \, .$$

This is an odd quantity to investigate. We choose it because the mathematics behind the error attribution and weight update is motivated by trying to decrease the sum-of-squares error for any given training example. (And the researchers who did the derivation chose it basically because they found a technique for decreasing it. That circularity — they discovered that the mathematics worked if they just chose to try to minimize this peculiar error function — is the spark of genius that makes backpropagation work.)

Figure 2.5 graphs the sum-of-squares error on the y-axis and the number of iterations through the data set on the x-axis. The thick line is the important number — it's the average sum-of-squares error over the 50 evaluation examples. The thin line is the average sum-of-squares error over the 100 training examples.

Since the mathematics behind the backpropagation update rule works with a goal of minimizing the sum-of-squares error, we expect that the thin line in this graph is constantly decreasing. But, then, that improvement is on the training examples: It's not indicative of general performance. The thick line represents performance on the evaluation examples the ANN never trains on, which *is* indicative of general performance. You can see that it flattens out after 1,000 iterations through the examples and then takes a upward turn after another 1,000 iterations, despite continued improvement on the training examples.

What's happening beyond the 2,000th iteration is *overfitting.* The neural net is adapting to specific

anomalies in the training data, not the general picture. If we were running the data, we'd want to stop around this point rather than continue onward.

Of course, one could argue that what we *really* care about is the number of evaluation examples classified correctly, not the average sum-of-squares error. But this supports the same conclusion we reached using the sum-of-squares error.

- By the 100th iteration through the examples, the ANN got 47 of the 50 correct.
- By the 1,000th, it was getting 49 of 50.
- Around the 2,000th iteration, the ANN was back to 48 of 50 and remains there.

So it was probably best to stop after the first 1,000 iterations. (This is just for a single run-through over the data. Different runs give slightly different numbers, due to the small random numbers chosen for the initial weights in the network.)

It's also worthwhile looking at how the behavior changes with different numbers of hidden units. Adding additional hidden units in this case keeps the picture more or less the same — maybe even slightly worse than 2 units. Only one hidden unit isn't powerful enough — the network never gets above 40 correct.

Typically, people find that there's some critical number of hidden units. Below this, the ANN performs poorly. At the critical number, it does well. And additional hidden units provide only marginal help if it helps at all, at the added expense of much more computation.

2.2.5 Analysis

ANNs have proven to be a useful, if complicated, way of learning. They adapt to strange concepts relatively well in many situations. They are one of the more important results coming out of machine learning.

One of the complexities with using ANNs is the number of parameters you can tweak to work with the data better. You choose the representation of attribute vectors and labels, the architecture of the network, the training rate, and how many iterations through the examples you want to do. The process is much more complicated than simply feeding the data into a linear regression program. The extended example presented in this chapter is intended to illustrate a realistic example, where we had to make a variety of decisions in order to get a good neural network for predicting iris classification.

Advantages:

- Very flexible in the types of hypotheses it can represent.
- Bears some resemblance to a very small human brain.
- Can adapt to new data with labels.

Disadvantages:

- Very difficult to interpret the hypothesis as a simple rule.
- Difficult to compute.

Chapter 3

Reinforcement learning

Until now we've been working with **supervised learning**, where the learner gets instant feedback about each example it sees. While this is useful and interesting, it doesn't model all types of learning.

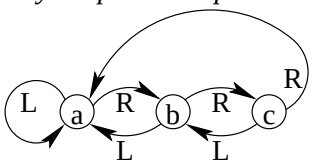
Consider, for example, a mouse trying to learn how to find cheese in a given maze. This doesn't really fit precisely into the supervised learning model. Whether the mouse chooses to go forward, right, or left doesn't result in immediate feedback — there is a whole sequence of actions that the mouse must learn in order to arrive at the cheese. There is sensory feedback along the way, but no information about whether the mouse is on the right track or not.

The field of **reinforcement learning** looks at how one might approach a problem like this. Two of the most prominent potential applications of reinforcement learning are game playing and robot learning. In both, the learner has to make a series of actions before achieving the goal, with no direct feedback about whether it's making the correct action.

3.1 Modeling the environment

Before we can analyze this problem, we need some way of thinking about it. One of the most useful techniques for thinking about the problem is as a **state space**. This is a diagram of how our world operates, including a **state** for each situation the learner might be in, and **transitions** representing how an action changes the current situation.

We'll be working with the following very simple state space with just three states, a, b, and c.



From each state, there are two actions: We can move left (action L) or we can move right (action R). This state space is meant to represent a very simple maze — actually, just a short hallway — with four locations. The cheese is to the right of c; when the mouse reaches it, the psychologist picks up the mouse and puts it back at location a for another trial.

We'll think of the mouse as starting out at location c.

This is an unrealistically simple state space. But we need it that simple to keep within the scope of hand calculation.

The notion of **reward** is crucial to the problem, so we need to incorporate it into our model too. We'll place a number on some transitions representing how good that action is. In game playing, for example,

the last move before a game is won would get a positive number, while the last move before a game is lost would get a negative number.

In our example, there is a reward every time the mouse goes right from c, arriving at the cheese. So we'll place a reward of 64 on the long transition from c back to state a.

Our goal as the learner is to find a strategy — that is, a mapping associating an action with each state. Which strategy do we want? To compare strategies, we define **discounted cumulative reward** (or just the **discounted reward**) by the infinite summation

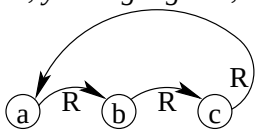
$$r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \gamma^4 r_4 + \cdots .$$

In this expression, r_t represents the reward we receive for the tth step if we follow the strategy, while γ is a number ($0 < \gamma < 1$) that gives more distant rewards less emphasis. Part of the point of γ is to emphasize rewards received earlier. But more importantly, it keeps the sum finite. If we were to leave out the γ altogether, the cumulative reward would be infinite.

The strategy that we want to learn is the strategy with the maximum discounted reward. Why the discounted reward and not something else? Basically, we use the discounted reward because it's convenient. It keeps the mathematics simple to have our goal be a combination of additions and multiplications. You could use another criterion for comparing strategies, and researchers have examined several, but it makes things more complicated.

We'll always use $\gamma = \frac{1}{2}$ in this chapter.

The optimal strategy for our example, you might guess, is the following.



To compute the discounted reward, we think as following. Recall that we decided to begin in state c. Our first action, therefore, moving back to state a, gives us a reward of 64; thus $r_0 = 64$. Our second action takes us to state b, at a reward r_1 of 0. The third has a reward r_2 of 0, leaving us at c. The fourth has a reward r_3 of 64, leaving us at a. And this will continue in cycles of 3.

$$\begin{aligned} & 64 + \left(\frac{1}{2}\right) 0 + \left(\frac{1}{2}\right)^2 0 + \left(\frac{1}{2}\right)^3 64 + \left(\frac{1}{2}\right)^4 0 + \left(\frac{1}{2}\right)^5 0 + \left(\frac{1}{2}\right)^6 64 + \cdots \\ = \;& 64 + \left(\frac{1}{2}\right)^3 64 + \left(\frac{1}{2}\right)^6 64 + \left(\frac{1}{2}\right)^9 64 + \cdots \\ = \;& 64 + \left(\frac{1}{8}\right) 64 + \left(\frac{1}{8}\right)^2 64 + \left(\frac{1}{8}\right)^3 64 + \cdots \\ = \;& \left(\frac{8}{7}\right) 64 = 73\frac{1}{7} \end{aligned}$$

Thus $73\frac{1}{7}$ is the discounted reward of the strategy depicted above. In the last step of this computation, we used the fact that the infinite sum $1 + (\frac{1}{8}) + (\frac{1}{8})^2 + \cdots$ is exactly $\frac{8}{7}$. You may have learned how to do these infinite sums before, but if not, don't worry — we won't use it again.

3.2 Q learning

Finding the optimal strategy when we have a map of the maze is an interesting problem, but it's not learning — it's just computation. In reality, we won't have a map. Either we won't understand our environment (like

the feeble mouse) or the map will be too large to explicitly remember (like in game playing). At any rate, we can't just look at the maze, determine the optimal strategy, and then play it out.

In this section, we'll look at Q learning, a specific algorithm for learning the optimal policy developed in 1989 by Watkins [Wat89].

Model What we'll actually learn are a set of numbers derived from the discounted reward. For each transition of the state space, we'll learn a number: For a state s and action A from that state, we'll use $Q(s, A)$ to refer to this number, representing the discounted reward if we start at s, take action A, and then follow the optimal strategy from there on. (Don't worry — we won't compute $Q(s, A)$ directly in our algorithm. We can't, since we don't have a map to work with. Instead, we'll gradually zero in onto the correct values of $Q(s, A)$ through a series of trials, and derive our strategy from that. But understanding Q is key to understanding Q learning.)

Here are the exact Q values for our space.

Q	a	b	c
L	$9\frac{1}{7}$	$9\frac{1}{7}$	$18\frac{2}{7}$
R	$18\frac{2}{7}$	$36\frac{4}{7}$	$73\frac{1}{7}$

For example, we know $Q(c, R)$ is $73\frac{1}{7}$ because it represents the discounted reward of going right from c and following the optimal strategy thereafter — that's exactly the action sequence we considered when we computed discounted reward in Section 3.1.

Or consider $Q(a, L)$. Here we want the discounted reward of starting in a, going left, and then following the optimal strategy thereafter. After our first step of going left, we get a reward r_0 of 0 and are in state a. The second step (now following the optimal strategy) takes us right from a, getting a reward r_1 of 0 and putting us in state b. The third step places us in c at a reward r_2 of 0. The fourth step places us in a at a reward r_3 of 64. The fifth step places us in b at a reward r_4 of 0. And so on.

$$\begin{aligned} & 0 + \left(\frac{1}{2}\right) 0 + \left(\frac{1}{2}\right)^2 0 + \left(\frac{1}{2}\right)^3 64 + \left(\frac{1}{2}\right)^4 0 + \left(\frac{1}{2}\right)^5 0 + \left(\frac{1}{2}\right)^6 64 + \cdots \\ = \;& \left(\frac{1}{2}\right)^3 64 + \left(\frac{1}{2}\right)^6 64 + \left(\frac{1}{2}\right)^9 64 + \cdots \\ = \;& 8 + \left(\frac{1}{8}\right) 8 + \left(\frac{1}{8}\right)^2 8 + \left(\frac{1}{8}\right)^3 8 + \cdots \\ = \;& \left(\frac{8}{7}\right) 8 = 9\frac{1}{7} \end{aligned}$$

And that's why we have $9\frac{1}{7}$ in our table for $Q(a, L)$.

The nice thing about knowing the Q values is that knowing them gives us the optimal strategy: For a particular state, to find the optimal action, we'll look at the Q values for each action starting at that state, and then take the action associated with the largest of these. Thus the discounted reward starting at state s is defined by the quantity

$$\max_A Q(s, A) \; .$$

Notice the following property of the Q values: For any action A taking us from state s to state s', giving us an immediate reward r, we have

$$Q(s, A) = r + \gamma \max_{A'} Q(s', A') \; .$$

That is, we get a reward of r immediately, followed by the discounted reward of following the optimal strategy starting at s'. (We scaled this discounted reward by γ because we start the optimal strategy at the

second step and beyond, whereas the discounted reward starting at s' follows the optimal strategy from the first step on.)

For example, $Q(b, R)$ in our example is $36\,\frac{4}{7}$. The immediate reward r is 0 for going right from b, and $Q(c, R)$ is the best action starting from the new state, c, having a discounted reward of $73\,\frac{1}{7}$. Using our formula, we find that

$$r + \gamma \max_{A'} Q(s', A') = 0 + \left(\frac{1}{2}\right) 73\frac{1}{7} = 36\frac{4}{7}\,.$$

Algorithm We'll use this last observation, that $Q(s, a) = r + \gamma \max_{A'} Q(s', A')$ to define our algorithm.

Start with $Q(s, A) = 0$ for all states s and actions A.
repeat indefinitely:
$s \leftarrow$ current state.
$A \leftarrow$ some action we select.
Perform action A.
$r \leftarrow$ reward for performing A.
$s' \leftarrow$ new state after performing A.
$Q(s, A) \leftarrow r + \gamma \max_{A'} Q(s', A')$.
end repeat

This is really quite simple: We begin by estimating all Q values as 0. Then we wander around the state space and update the Q values using our formula as we go.

We start at state c; say we choose to move right first. We get a reward of 64 and end up in state a. According to our current Q-value estimates, the estimated discounted reward starting at a is 0, so we update $Q(c, R)$ to be $64 + (\frac{1}{2})0 = 64$.

Q	a	b	c
L	0	0	0
R	0	0	64

Now say we move to the right again. We get a reward of 0 and end up in state b. Our estimate of the discounted reward starting in b, according to the current Q values, is 0. So we update $Q(a, R)$ to be $0 + (\frac{1}{2})0 = 0$ (which was the value we had before).

And if we move to the right again, we get a reward of 0 and end up in state c. Our estimate of the discounted reward starting in c, according to the current Q values, is 64. So we update $Q(b, R)$ to be $0 + (\frac{1}{2})64 = 32$.

Q	a	b	c
L	0	0	0
R	0	32	64

We continue doing this over time until we're satisfied. Here are the next several steps.

s	A	r	s'	update of $Q(s, A)$
c	L	0	b	$Q(c, L) \leftarrow 0 + (\frac{1}{2})32 = 16$
b	R	0	c	$Q(b, R) \leftarrow 0 + (\frac{1}{2})64 = 32$
c	R	64	a	$Q(c, R) \leftarrow 64 + (\frac{1}{2})\ 0 = 64$
a	R	0	b	$Q(a, R) \leftarrow 0 + (\frac{1}{2})32 = 16$
b	L	0	a	$Q(b, L) \leftarrow 0 + (\frac{1}{2})16 = 8$
a	L	0	a	$Q(a, L) \leftarrow 0 + (\frac{1}{2})16 = 8$

So after all these steps our final estimates of the Q values are the following.

Q	a	b	c
L	8	8	16
R	16	32	64

This isn't too far off the correct answer, and if we continued wandering we would get closer and closer to the correct Q values.

It doesn't matter how we choose to move here. As long as we occasionally try out all the possible transitions, any way of making this selection is fine. If you wanted to accumulate rewards as you learn, you might tend to select moves that have higher Q values.

In situations where you want to accumulate reward as you learn, you have a conflict of interest. The learner will want to try out actions that haven't been tried too often before in the same state, to gain more experience with the consequences. But also the learner wants to accumulate rewards when it can. This is known as the issue of *exploration vs exploitation* — balancing it is an interesting problem. But we'll sidestep it with Q learning, because it actually doesn't affect whether Q learning works.

At this point our goal is simply to learn the optimal strategy, not to garner points particularly. Thus we can choose to move however we want. Choosing randomly each step from the available actions, for example, isn't a bad idea.

Analysis The neat thing about Q learning is that it works, though the algorithm is very simple. Moreover, it works without constructing an explicit map of the state space, which is important from a practical point of view, since in practice the map can get very complicated. But Q learning has some problems that are worth pointing out.

- The algorithm doesn't work well in a nondeterministic world. Often, an action doesn't always have the same consequences. This is obviously true in game playing, when each action is followed by an opponent's move, which the player cannot predict. It's also true in robot learning, as the real world often doesn't behave exactly as predicted. For example, a robot building a tower of blocks may find that a gust of wind blows the tower over unexpectedly.

  The Q learning algorithm we've seen forgets everything that's happened before when it's performed the action and replaces it with the new observation. That works fine in a perfect nonrandom world, when every action invariably leads to the same situation, but not in more realistic situations.

- The algorithm learns more slowly than we might hope. What will happen is that each time through the maze, the algorithm extends its knowledge one step backward. In our example, we learned that moving from c to a was a good move the first time; then the next time through the maze we learned to move from b to c; and then the next time through we learned to move from a to b. But we should have figured out earlier that it is a good idea to move from a to b.

- For problems of a realistic size, hoping to go through all possible states is simply not realistic. In a chess game, for example, there are so many states that we can never hope to visit every single possible board that might show up during a game.

In the next two sections, we'll see ways of refining Q learning to address the first two issues. The third issue, however, is more endemic to the overall model that a simple patch isn't going to solve.

This problem is one of *generalizing* from what we have learned. That is, we need to somehow recognize that two states are similar and so if an action is good for one, a similar action is good for the other. This is something that will be more domain-specific than our state-space abstraction allows: In the state space abstraction, each state is completely incomparable to the rest. We've abstracted away all details about the characteristics of the state.

People don't really have a good way of handling this in general. We'll look at the two most prominent successful applications of reinforcement learning techniques at the close of this chapter. In both applications, they faced this problem of not being able to hope to visit all possible states, and we'll see how they resolve it in their particular situations.

3.3 Nondeterministic worlds

To address nondeterministic consequences of actions, we'll be less extreme in our update of Q. Before, we changed $Q(s, A)$ to $r + \gamma \max_{A'} Q(s, A')$, forgetting everything we had learned about $Q(s, A)$ before. Instead, what we'll do is *average* the current value of $Q(s, A)$ with this new observation.

The algorithm proceeds in the same way, but we'll alter the update rule to the following.

$$Q(s, A) \leftarrow (1 - \alpha)Q(s, A) + \alpha \left(r + \gamma \max_{A'} Q(s', A') \right)$$

Here, α (which we choose from $0 < \alpha \leq 1$) is a parameter that tells us how much weight to give the new observation and how much weight to give to the value we were remembering earlier. If α is small, this indicates that we want to give lots of weight to old values of Q, so that we improve our estimate only slowly over time. If α is large, we want to give more weight to the new values, so that we learn more rapidly at the expense of the risk of overemphasizing rare consequences of our actions.

In practice, you want to use a large α at the beginning when the Q values don't reflect any experience (and so you might as well adapt quickly), and you want to reduce α as you gain more experience. A very sensible choice changes α with each action to $1/(1 + v)$, where v represents the number of times we've performed the action from that particular state previously.

In our world, we don't need this alternative formulation because our world is deterministic. Thus we might as well use $\alpha = 1$, which gives us the same algorithm we used before. But if we instead choose α to be $1/(1 + v)$, we end up with the following table after the same sequence of actions we performed before.

Q	a	b	c
L	4	4	16
R	8	32	64

We haven't gotten as close to the correct Q values here as we did in Scetion 3.2. But, then, we intentionally *modified our original algorithm to adapt more slowly.*

3.4 Fast adaptation

A more radical change is trying to make the algorithm so that each reward is immediately propagated backward to actions performed in the past. The idea here is to speed up the learning process, so less wandering is required.

Here's one solution. What we're going to do is to maintain an h-value for each transition of the state space. This h-value continually decays, at a rate of $\gamma\lambda$ after each action. The λ parameter (with $0 \leq \lambda \leq 1$) controls how much effect a future reward has on a current action.

The h-values start out at 0, but each time we use a transition we'll add one to its h-value. But it will rapidly decay back to zero (at a rate of $\gamma\lambda$ per time step) as future actions are performed.

Start with $Q(s, A) = 0$ and $h(s, A) = 0$ for all states s and actions A.

repeat indefinitely:

$s \leftarrow$ current state.

$A \leftarrow$ some action we select.
Perform action A.
$r \leftarrow$ reward for performing A.
$s' \leftarrow$ new state after performing A.
$\delta \leftarrow -Q(s, A) + r + \gamma \max_{A'} Q(s', A')$.
$h(s, A) \leftarrow h(s, A) + 1$.
for each state s'' and each action A'', **do:**
$Q(s'', A'') \leftarrow Q(s'', A'') + \alpha\delta h(s'', A'')$.
$h(s'', A'') \leftarrow \gamma\lambda h(s'', A'')$.
end for
end repeat

We'll use $\alpha = 1$, $\gamma = \frac{1}{2}$, *and* $\lambda = \frac{1}{2}$.

Things start out the same. If we begin at c *and move right, we get a reward of* 64 *and go into state* a, *whose estimated discounted reward is* 0. *So we'll compute* δ *as* $-0 + 64 + (\frac{1}{2})0 = 64$, *and we'll add* 1 *to* $h(c, R)$.

Now when we go through each s'' *and* A'', *the only non-zero* h*-value is* $h(c, R) = 1$. *So the only* Q*-value that changes is* $Q(c, R)$, *which changes to* $0 + \alpha\delta 1 = 64$. *Each* h*-value decays by* $\gamma\lambda = (\frac{1}{2})(\frac{1}{2}) = \frac{1}{4}$. *Thus after our first action the* Q*- and* h*-values are as follows.*

Q	a	b	c
L	0	0	0
R	0	0	64

h	a	b	c
L	0	0	0
R	0	0	$\frac{1}{4}$

Our second action gives us a reward of 0 *and places us in state* b, *whose estimated discounted reward is* 0. *We compute* δ *as* $-0 + 0 + (\frac{1}{2})0 = 0$. *Since* $\delta = 0$, *no* Q*-values will change, though of course the* h*-values decay by a factor of* $\frac{1}{4}$ *again.*

Q	a	b	c
L	0	0	0
R	0	0	64

h	a	b	c
L	0	0	0
R	$\frac{1}{4}$	0	$\frac{1}{16}$

So far the Q*-values are just as in our first example. But for the third action, when we move right from* b *to earn a reward of* 0 *and end up in state* c, *where the estimated discounted reward is* 64, *we'll see something different. This time we compute* δ *to be* $-0 + 0 + (\frac{1}{2})64 = 32$. *As before,* $Q(b, R)$ *will change to* $0 + 32 \cdot 1 = 32$. *But also, since* $h(a, R) = \frac{1}{4}$, *the value* $Q(a, R)$ *changes to* $0 + 32 \cdot \frac{1}{4} = 8$. *Moreover,* $Q(c, R)$ *changes to* $64 + 32 \cdot \frac{1}{16} = 66$. *So we now have the following.*

Q	a	b	c
L	0	0	0
R	8	32	66

h	a	b	c
L	0	0	0
R	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{64}$

What's happened here is that we got an estimated future reward for moving into c, *and this new algorithm has propagated this future reward to our past actions also — a fraction of it to the action of moving right from* a, *and even a fraction to the action of moving right from* c.

Were we to continue through the entire example we've been using, we'd find that the final Q*-values are as follows.*

Q	a	b	c
L	9.44	10.75	15.59
R	19.47	34.24	71.11

These have gotten very close to the exact values with what's really a pretty short time wandering through the state space.

3.5 TD learning

Temporal difference learning is an alternative algorithm to Q learning originally proposed by Sutton in 1988 [Sut88].

Actually, it's closely related to Q learning. It's even a little simpler. The difference is that, while the Q learning algorithm learns the discounted reward beginning at each transition in the state space, TD learning learns the discounted reward $V(s)$ for each *state* s of the state space. Everything else remains the same.

Thus, the simplest version of TD learning (analogous to the Q learning algorithm of Section 3.2) revises the update rule to the following.

$$V(s) \leftarrow r + \gamma V(s')$$

Like Q learning, TD learning can be modified as in Sections 3.3 and 3.4.

In situations where the state space is known (as in game playing), TD learning makes sense. Because you're learning a value for each state, not each possible action from each state, you have many fewer numbers to learn. Thus less exploration is required to find a good solution.

But in situations where the state space is unknown, TD learning is impractical. After all, what we're after is learning the optimal action from each state of the space. To figure out the optimal action for s given the space's V-values, we would look through each possible action from s, take the reward r and subsequent state s' for that action, and say that the optimal action is the action for which $r + \gamma V(s')$ is largest. But if we don't understand the state space, we don't know r or s', and so we can't perform this computation.

Thus TD learning is frequently superior to Q learning when we know the state space, and TD learning is inappropriate if we don't.

3.6 Applications

As we saw in Section 3.2, Q learning — and, indeed, the whole state-space model — has a fatal problem: The world is just too complicated for us to hope to go through most real-life state spaces.

Much of current research around reinforcement learning involves looking at practical applications demonstrating techniques for getting around this difficulty of tackling huge state spaces. Results have been mixed. We'll look at two of the more significant successes.

3.6.1 TD-Gammon

The first big reinforcement learning success is certainly TD-Gammon, a backgammon-playing system developed by Tesauro in the early 1990's [Tes95].

Although we're not going to look at the specifics of backgammon here, the important thing to know that there is a set of pieces you are trying to move to goal positions. Each turn, the player rolls a pair of dice and must choose from a set of moves dictated by that roll. Conventional game-playing techniques have not applied well to Backgammon, as the number of possible moves in each turn is unusually large (about 400, versus about 40 for chess), especially if you must consider all possible dice rolls.

Tesauro used TD learning, but the number of states in backgammon in too overwhelming to consider. Rather than tabulate all possible states, TD-Gammon employed a neural network to approximate the V-values for each state of the network. In the first version (version 0.0), the neural network had 198 input units for representing the current board, 40 hidden units, and a single output to represent the estimate for the input board's V-value.

To choose a move, therefore, you had only to try each of the possible moves and see which resulting board state gives the greatest V-value. Then you take that move. Generally the reward will be 0, as you don't receive a reward or penalty until the game is won or lost. From here you can train the network, just as

is done in regular TD learning. (Tesauro used the TD equivalent of the Q-learning algorithm of Section 2.4, using $\lambda = 0.7$.)

By training the network on 200,000 games against itself, Tesauro found that TD-Gammon performed about as well as one of the best computer backgammon players of the time, Neurogammon (also by Tesauro). This was shocking, as Neurogammon had been written specifically to play backgammon, with much specialized knowledge built into it, while TD-Gammon had nearly no backgammon knowledge outside of the rules.

Encouraged by this work, Tesauro developed TD-Gammon 1.0. The basic change for this was to change the inputs to the neural network to represent more complex concepts that backgammon masters feel are important things about a board state — whether crucial positions are blocked, how close pieces are to winning, etc. In other words, this new version incorporated some of the advanced backgammon theory that Neurogammon had. TD-Gammon 1.0 had 80 hidden units and trained on itself with 300,000 games. It played competitively with human champions, but not at a level where it might win a world championship.

Subsequent versions of TD-Gammon have made the program look more than one move in advance, taking some advantage of classical minimax search. The current version, TD-Gammon 3.1, is widely regarded as ranking among the best backgammon players in the world (human or computer), and very capable of winning a world championship if admitted.

People have tried duplicating this work in other games, especially chess and go. The results of these attempts have been encouraging but not as noteworthy as TD-Gammon: The programs learn to play well, but not near the level of human champions.

3.6.2 Elevator scheduling

Scheduling elevators in a building is an important problem. Researchers like it because it's simple and practical. Unfortunately, they don't get to see their work put into practice — or at least they don't know about it: Elevator manufacturers jealously protect the scheduling algorithms they use, as this is part of their competitive edge over other elevator manufacturers.

Crites and Barto, however, can pride themselves on having one of the best *published* algorithms for scheduling elevators [CB96]. (We don't have a way of comparing to the manufacturers' unpublished algorithms.) Moreover, their technique uses reinforcement learning.

In their study, Crites and Barto concentrated on scheduling four elevators in a ten-story building during the 5:00 rush (when nearly all passengers wish to go to the lobby to leave for home). In their simulated system, each of the four elevators has within it ten buttons, and each floor has an up and a down button on it. At any point in time, each elevator has a location, direction, and speed; and each button pressed on a floor has a "waiting time" associated with it, saying how long it has been since the prospective passenger requested an elevator.

This is a world with many states. Crites and Barto conservatively estimate that it has at least 10^{22} states, far more than we could visit in the course of learning how to behave in each state.

An action in this state space gives the behavior of the elevators. An elevator may decide to go up, go down, or stay in its current location to unload or load passengers. The penalty for an action can be computed as the sums of the squares of the waiting time for any passengers loaded by the elevators during the time step. (We square the waiting time because realistically we want to give passengers who have been waiting quite long a larger advantage.)

Notice that one aspect of this scenario is that we can't predict the result of an action, because we can't predict where potential passengers are going to appear requesting an elevator. Thus, Q learning is more applicable than TD learning for this problem.

But we still have the problem of all those states that we can't visit thoroughly during training. Crites and

Barto used several techniques to bring the state space down to a more reasonable level. One thing they did to simplify the problem was to add additional constraints on possible actions — constraints that you probably need anyway. For example, an elevator cannot pass a floor if a passenger within the elevator has requested a stop on that floor. Nor can an elevator reverse direction if a passenger has requested a floor in the current direction. These constraints reduce the number of decisions that the elevators have to make.

Despite these constraints, elevators must still make some choices. Namely, when an elevator reaches a floor where a passenger is waiting, should it stop? (You may expect it to always stop for potential passengers, but if another elevator is stopping there anyway because it is carrying a passenger there, why waste time? Or perhaps the elevator wants desperately to reach a person who has been waiting longer.)

Crites and Barto decided to make the elevators make their decisions independently. This will hurt performance, as more centralized control will allow elevators to band together to tackle their passengers more effectively. But making them independent simplifies the set of actions from which the learning algorithm must choose to just two: When we reach a waiting passenger, do we stop or not?

But there are still lots of states to consider — too many to hope to apply Q learning directly. The final technique that Crites and Barto employ is to use a neural network to approximate the Q values, similar to how Tesauro used a neural network to approximate V values in TD learning.

Recall that Crites and Barto have their elevators make decisions independently, so each network computes the Q-value for only two actions. After much experimentation, they settled on a network of 47 input units and 2 output units. The two output units represent the two possible actions.

The 47 input units represent different characteristics of the current state, and it is here that you can see how Crites and Barto must have tried a lot of things before settling down on these 47 units. 18 of the units represented the buttons in the hallways — one unit for each floor representing whether a down passenger is still pending, and one unit for each floor representing how long that down passenger has been pending. 16 units represented the current location and direction of the elevator in question. Another 10 units say which floors have the other elevators. One unit says whether the elevator in question is at the highest floor with an elevator request, one unit says whether the elevator in question is at the floor with the oldest unsatisfied elevator request, and the final unit is a constant-1 input.

They trained their network in a simulated system and compared their results with those of many other algorithms. What they found is that the strategy learned by Q learning and their neural network outperformed the previous *ad hoc* algorithms designed by humans for elevator scheduling. This doesn't necessarily imply that Q learning is the best algorithm for elevator scheduling, but it does illustrate a situation where it has proven successful.

Chapter 4

References

One of the best machine learning textbooks is *Machine Learning*, by Tom Mitchell [Mit97]. Nearly everything covered in these chapters is also covered in Mitchell's textbook, and of course Mitchell covers much more. The book is generally aimed at somebody with several years of computer science training (like a senior), but it's thorough.

On data mining, I found the textbook *Data Mining*, by Witten and Frank, useful [WF00]. It's really a misnamed book, as it's more of a practitioner's guide to machine learning than a textbook about data mining. But it does a good job of that. Its presentation is typically simpler (if less rigorous) way than Mitchell's; it is well-suited for experienced programmers who want to use machine learning.

Even Mitchell's book, thorough in many other respects, neglects reinforcement learning to a large degree. The only reasonable reference is Sutton and Barto's book *Reinforcement Learning: An Introduction* [SB98]. (These machine learning authors don't seem to have any desire for interesting titles.)

Finally, I drew much of the material about artificial life from Franklin's *Artificial Minds*. The book is really a non-specialist's take on artificial intelligence, written for a popular audience (i.e., non-specialists). So you can expect some handwaving of the same sort you would see in other mass-market science books. But Franklin writes well, and he covers the philosophical issues that authors of artificial intelligence textbooks typically avoid.

[AL92] D. Ackley and M. Littman. Interactions between learning and evolution. In C. Langton et al., editor, *Artificial Life II*, pages 487–509. Redwood City, Calif.: Addison-Wesley, 1992.

[CB96] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advanced in Neural Information Processing Systems*, volume 8. Cambridge, Mass.: MIT Press, 1996.

[Fis36] R. Fisher. The use of multiple measurements in taxonomic problems. *Annual Eugenics*, 7 (part II), 1936.

[MC80] R. Michalski and R. Chilausky. Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *International Journal of Policy Analysis and Information Systems*, 4(2), 1980.

[Mit97] T. Mitchell. *Machine Learning*. Boston: McGraw-Hill, 1997.

[Pom93] D. Pomerleau. Knowledge-based training of artificial neural networks for autonomuous robot driving. In J. Connell and S. Mahadevan, editors, *Robot Learning*, pages 19–43. Boston: Kluwer Academic Publishers, 1993.

[SB98] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* Cambridge, Mass.: MIT Press, 1998.

[Sut88] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[Tes95] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3), 1995.

[Wat89] C. Watkins. *Learning from delayed rewards.* PhD thesis, Cambridge University, 1989.

[WF00] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* San Francisco: Morgan Kaufmann, 2000.

[Wil85] S. Wilson. Knowledge growth in an artificial animal. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 16–23. Hillsdale, N.J.: Erlbaum, 1985.