Information Theory, Inference, and Learning Algorithms

David J.C. MacKay
Contents

Preface ..................................................... v
1 Introduction to Information Theory ............... 3
2 Probability, Entropy, and Inference ............... 22
3 More about Inference .................................. 48

I Data Compression ........................................ 65
4 The Source Coding Theorem ......................... 67
5 Symbol Codes ............................................ 91
6 Stream Codes ........................................... 110
7 Codes for Integers ..................................... 132

II Noisy-Channel Coding ................................. 137
8 Correlated Random Variables ....................... 138
9 Communication over a Noisy Channel .............. 146
10 The Noisy-Channel Coding Theorem ............... 162
11 Error-Correcting Codes and Real Channels ....... 177

III Further Topics in Information Theory .......... 191
12 Hash Codes: Codes for Efficient Information Retrieval .... 193
13 Binary Codes ............................................ 206
14 Very Good Linear Codes Exist ..................... 229
15 Further Exercises on Information Theory .......... 233
16 Message Passing ......................................... 241
17 Communication over Constrained Noiseless Channels .... 248
18 Crosswords and Codebreaking ...................... 260
19 Why have Sex? Information Acquisition and Evolution .... 269

IV Probabilities and Inference ......................... 281
20 An Example Inference Task: Clustering ............ 284
21 Exact Inference by Complete Enumeration .......... 293
22 Maximum Likelihood and Clustering ............... 300
23 Useful Probability Distributions ................... 311
24 Exact Marginalization ................................ 319
25 Exact Marginalization in Trellises ................. 324
26 Exact Marginalization in Graphs ............ 334
Part V

Neural networks
In the field of neural networks, we study the properties of networks of idealized ‘neurons’. Three motivations underlie work in this broad and interdisciplinary field.

**Biology.** The task of understanding how the brain works is one of the outstanding unsolved problems in science. Some neural network models are intended to shed light on the way in which computation and memory are performed by brains.

**Engineering.** Many researchers would like to create machines that can ‘learn’, perform ‘pattern recognition’ or ‘discover patterns in data’.

**Complex systems.** A third motivation for being interested in neural networks is that they are complex adaptive systems whose properties are interesting in their own right.

I should emphasize several points at the outset.

- This book gives only a taste of this field. There are many interesting neural network models which we will not have time to touch on.
- The models that we discuss are not intended to be faithful models of biological systems. If they are at all relevant to biology, their relevance is on an abstract level.
- I will describe some neural network methods that are widely used in nonlinear data modelling, but I will not be able to give a full description of the state of the art. If you wish to solve real problems with neural networks, please read the relevant papers.

### 38.1 Memories

In the next few chapters we will meet several neural network models which come with simple learning algorithms which make them function as memories. Perhaps we should dwell for a moment on the conventional idea of memory in digital computation. A memory (a string of 5000 bits describing the name of a person and an image of their face, say) is stored in a digital computer at an address. To retrieve the memory you need to know the address. The address has nothing to do with the memory itself. Notice the properties that this scheme does not have:

1. **Address-based memory is not associative.** Imagine you know half of a memory, say someone’s face, and you would like to recall the rest of the
38.1: Memories

memory – their name. If your memory is address-based then you can’t get at a memory without knowing the address. [Computer scientists have devoted effort to wrapping traditional address-based memories inside cunning software to produce content-addressable memories, but content-addressability does not come naturally. It has to be added on.]

2. Address-based memory is *not* robust or fault-tolerant. If a one-bit mistake is made in specifying the *address* then a completely different memory will be retrieved. If one bit of a *memory* is flipped then whenever that memory is retrieved the error will be present. Of course, in all modern computers, error-correcting codes are used in the memory, so that small numbers of errors can be detected and corrected. But this error-tolerance is not an intrinsic property of the memory system. If minor damage occurs to certain hardware that implements memory retrieval, it is likely that all functionality will be catastrophically lost.

3. Address-based memory is not distributed. In a serial computer that is accessing a particular memory, only a tiny fraction of the devices participate in the memory recall: the CPU and the circuits that are storing the required byte. All the other millions of devices in the machine are sitting idle.

Are there models of truly parallel computation, in which multiple devices participate in all computations? [Present-day parallel computers scarcely differ from serial computers from this point of view. Memory retrieval works in just the same way, and control of the computation process resides in CPUs. There are simply a few more CPUs. Most of the devices sit idle most of the time.]

Biological memory systems are completely different.

1. Biological memory is associative. Memory recall is *content-addressable*. Given a person’s name, we can often recall their face; and *vice versa*. Memories are apparently recalled spontaneously, not just at the request of some CPU.

2. Biological memory recall is error-tolerant and robust.

   - Errors in the cues for memory recall can be corrected. An example asks you to recall ‘An American politician who was very intelligent and whose politician father did not like broccoli’. Many people think of president Bush – even though one of the cues contains an error.
   - Hardware faults can also be tolerated. Our brains are noisy lumps of meat that are in a continual state of change, with cells being damaged by natural processes, alcohol, and boxing. While the cells in our brains and the proteins in our cells are continually changing, many of our memories persist unaffected.

3. Biological memory is parallel and distributed – not *completely* distributed throughout the whole brain: there does appear to be some functional specialization – but in the parts of the brain where memories are stored, it seems that many neurons participate in the storage of multiple memories.

These properties of biological memory systems motivate the study of ‘artificial neural networks’ – parallel distributed computational systems consisting
of many interacting simple elements. The hope is that these model systems might give some hints as to how neural computation is achieved in real biological neural networks.

38.2 Terminology

Each time we describe a neural network algorithm we will typically specify three things. [If any of this terminology is hard to understand, it’s probably best to dive straight into the next chapter.]

**Architecture.** The architecture specifies what variables are involved in the network and their topological relationships – for example, the variables involved in a neural net might be the weights of the connections between the neurons, along with the activities of the neurons.

**Activity rule.** Most neural network models have short time-scale dynamics: local rules define how the activities of the neurons change in response to each other. Typically the activity rule depends on the weights (the parameters) in the network.

**Learning rule.** The learning rule specifies the way in which the neural network’s weights change with time. This learning is usually viewed as taking place on a longer time scale than the time scale of the dynamics under the activity rule. Usually the learning rule will depend on the activities of the neurons. It may also depend on the values of target values supplied by a teacher and on the current value of the weights.

Where do these rules come from? Often, activity rules and learning rules are invented by imaginative researchers. Alternatively, activity rules and learning rules may be derived from carefully chosen objective functions.

Neural network algorithms can be roughly divided into two classes.

**Supervised neural networks** are given data in the form of inputs and targets, the targets being a teacher’s specification of what the neural network’s response to the input should be.

**Unsupervised neural networks** are given data in an undivided form – simply a set of examples \( \{x\} \). Some learning algorithms are intended simply to memorize these data in such a way that the examples can be recalled in the future. Other algorithms are intended to ‘generalize’, to discover ‘patterns’ in the data, or extract the underlying ‘features’ from them.

Some unsupervised algorithms are able to make predictions – for example, some algorithms can ‘fill in’ missing variables in an example \( x \) – and so can also be viewed as supervised networks.
The Single Neuron as a Classifier

39.1 The single neuron

We will study a single neuron for two reasons. First, many neural network models are built out of single neurons, so it is good to understand them in detail. And second, a single neuron is itself capable of 'learning' — indeed, various standard statistical methods can be viewed in terms of single neurons — so this model will serve as a first example of a supervised neural network.

Definition of a single neuron

We will start by defining the architecture and the activity rule of a single neuron, and we will then derive a learning rule.

Architecture. A single neuron has a number $I$ of inputs $x_i$ and one output which we will here call $y$. (See figure 39.1.) Associated with each input is a weight $w_i$ ($i = 1, \ldots, I$). There may be an additional parameter $w_0$ of the neuron called a bias which we may view as being the weight associated with an input $x_0$ which is permanently set to 1. The single neuron is a feedforward device — the connections are directed from the inputs to the output of the neuron.

Activity rule. The activity rule has two steps.

1. First, in response to the imposed inputs $\mathbf{x}$, we compute the activation of the neuron,

$$a = \sum_i w_i x_i,$$

(39.1)

where the sum is over $i = 0, \ldots, I$ if there is a bias and $i = 1, \ldots, I$ otherwise.

2. Second, the output $y$ is set as a function $f(a)$ of the activation. The output is also called the activity of the neuron, not to be confused with the activation $a$. There are several possible activation functions; here are the most popular.

(a) Deterministic activation functions:

i. Linear.

$$y(a) = a.$$  

(39.2)

ii. Sigmoid (logistic function).

$$y(a) = \frac{1}{1 + e^{-a}} \quad (y \in (0, 1)).$$  

(39.3)
iii. Sigmoid (tanh).

\[ y(a) = \tanh(a) \quad (y \in (-1, 1)). \tag{39.4} \]

iv. Threshold function.

\[ y(a) = \Theta(a) = \begin{cases} 
1 & a > 0 \\
-1 & a \leq 0.
\end{cases} \tag{39.5} \]

(b) Stochastic activation functions: \( y \) is stochastically selected from \( \pm 1 \).

i. Heat bath.

\[ y(a) = \begin{cases} 
1 & \text{with probability } \frac{1}{1 + e^{-a}} \\
-1 & \text{otherwise.}
\end{cases} \tag{39.6} \]

ii. The Metropolis rule produces the output in a way that depends on the previous output state \( y \):

Compute \( \Delta = ay \)

If \( \Delta \leq 0 \), flip \( y \) to the other state

Else flip \( y \) to the other state with probability \( e^{-\Delta} \).

39.2 Basic neural network concepts

A neural network implements a function \( y(x; w) \); the `output' of the network, \( y \), is a nonlinear function of the `inputs' \( x \); this function is parameterized by `weights' \( w \).

We will study a single neuron which produces an output between 0 and 1 as the following function of \( x \):

\[ y(x; w) = \frac{1}{1 + e^{-w \cdot x}}. \tag{39.7} \]

Exercise 39.1. In what contexts have we encountered the function \( y(x; w) = 1/(1 + e^{-w \cdot x}) \) already?

Motivations for the linear logistic function

In section 11.2 we studied `the best detection of pulses', assuming that one of two signals \( x_0 \) and \( x_1 \) had been transmitted over a Gaussian channel with variance–covariance matrix \( A^{-1} \). We found that the probability that the source signal was \( s = 1 \) rather than \( s = 0 \), given the received signal \( y \), was

\[ P(s = 1|y) = \frac{1}{1 + \exp(-a(y))}, \tag{39.8} \]

where \( a(y) \) was a linear function of the received vector,

\[ a(y) = w^T y + \theta, \tag{39.9} \]

with \( w \equiv A(x_1 - x_0) \).

The linear logistic function can be motivated in several other ways – see the exercises.
39.2: Basic neural network concepts

Figure 39.2. Output of a simple neural network as a function of its input.

Input space and weight space

For convenience let us study the case where the input vector $\mathbf{x}$ and the parameter vector $\mathbf{w}$ are both two-dimensional: $\mathbf{x} = (x_1, x_2)$, $\mathbf{w} = (w_1, w_2)$. Then we can spell out the function performed by the neuron thus:

$$
    y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2)}}.
$$

(39.10)

Figure 39.2 shows the output of the neuron as a function of the input vector, for $\mathbf{w} = (0, 2)$. The two horizontal axes of this figure are the inputs $x_1$ and $x_2$, with the output $y$ on the vertical axis. Notice that on any line perpendicular to $\mathbf{w}$, the output is constant; and along a line in the direction of $\mathbf{w}$, the output is a sigmoid function.

We now introduce the idea of weight space, that is, the parameter space of the network. In this case, there are two parameters $w_1$ and $w_2$, so the weight space is two dimensional. This weight space is shown in figure 39.3. For a selection of values of the parameter vector $\mathbf{w}$, smaller inset figures show the function of $\mathbf{x}$ performed by the network when $\mathbf{w}$ is set to those values. Each of these smaller figures is equivalent to figure 39.2. Thus each point in $\mathbf{w}$ space corresponds to a function of $\mathbf{x}$. Notice that the gain of the sigmoid function (the gradient of the ramp) increases as the magnitude of $\mathbf{w}$ increases.

Now, the central idea of supervised neural networks is this. Given examples of a relationship between an input vector $\mathbf{x}$, and a target $t$, we hope to make the neural network ‘learn’ a model of the relationship between $\mathbf{x}$ and $t$. A successfully trained network will, for any given $\mathbf{x}$, give an output $y$ that is close (in some sense) to the target value $t$. Training the network involves searching in the weight space of the network for a value of $\mathbf{w}$ that produces a function that fits the provided training data well.

Typically an objective function or error function is defined, as a function of $\mathbf{w}$, to measure how well the network with weights set to $\mathbf{w}$ solves the task. The objective function is a sum of terms, one for each input/target pair $(\mathbf{x}, t)$, measuring how close the output $y(\mathbf{x}; \mathbf{w})$ is to the target $t$. The training process is an exercise in function minimization — i.e., adjusting $\mathbf{w}$ in such a way as to find a $\mathbf{w}$ that minimizes the objective function. Many function-minimization algorithms make use not only of the objective function, but also its gradient with respect to the parameters $\mathbf{w}$. For general feedforward neural networks the backpropagation algorithm efficiently evaluates the gradient of the output $y$ with respect to the parameters $\mathbf{w}$, and thence the gradient of the objective function with respect to $\mathbf{w}$. 

\[ w = (0, 2) \]
Figure 39.3. Weight space.
39.3 Training the single neuron as a binary classifier

We assume we have a data set of inputs \( \{x^{(n)}\}_{n=1}^{N} \) with binary labels \( \{t^{(n)}\}_{n=1}^{N} \), and a neuron whose output \( y(x; w) \) is bounded between 0 and 1. We can then write down the following error function:

\[
G(w) = - \sum_{n} \left[ t^{(n)} \ln y(x^{(n)}; w) + (1 - t^{(n)}) \ln(1 - y(x^{(n)}; w)) \right]. \tag{39.11}
\]

Each term in this objective function may be recognized as the information content of one outcome. It may also be described as the relative entropy between the empirical probability distribution \( \{t^{(n)}\}, \{1 - t^{(n)}\} \) and the probability distribution implied by the output of the neuron \( \{y, 1 - y\} \). The objective function is bounded below by zero and only attains this value if \( y(x^{(n)}; w) = t^{(n)} \) for all \( n \).

We now differentiate this objective function with respect to \( w \).

Exercise 39.2. The backpropagation algorithm. Show that the derivative \( g = \partial G / \partial w \) is given by:

\[
g_j = \frac{\partial G}{\partial w_j} = \sum_{n=1}^{N} -(t^{(n)} - y^{(n)}) x_j^{(n)}. \tag{39.12}
\]

Notice that the quantity \( e^{(n)} = t^{(n)} - y^{(n)} \) is the error on example \( n \) – the difference between the target and the output. The simplest thing to do with a gradient of an error function is to descend it (even though this is often dimensionally incorrect, since a gradient has dimensions \([1/\text{parameter}]\), whereas a change in a parameter has dimensions \([\text{parameter}]\)). Since the derivative \( \partial G / \partial w \) is a sum of terms \( g^{(n)} \) defined by

\[
g_j^{(n)} = -(t^{(n)} - y^{(n)}) x_j^{(n)} \tag{39.13}
\]

for \( n = 1, \ldots, N \), we can obtain a simple on-line algorithm by putting each input through the network one at a time, and adjusting \( w \) a little in a direction opposite to \( g^{(n)} \).

We summarize the whole learning algorithm.

The on-line gradient-descent learning algorithm

Architecture. A single neuron has a number \( I \) of inputs \( x_i \) and one output \( y \). Associated with each input is a weight \( w_i \) \( (i = 1, \ldots, I) \).

Activity rule. 1. First, in response to the received inputs \( x \) (which may be arbitrary real numbers), we compute the activation of the neuron,

\[
a = \sum_{i} w_i x_i, \tag{39.14}
\]

where the sum is over \( i = 0, \ldots, I \) if there is a bias and \( i = 1, \ldots, I \) otherwise.

2. Second, the output \( y \) is set as a sigmoid function of the activation.

\[
y(a) = \frac{1}{1 + e^{-a}}. \tag{39.15}
\]

This output might be viewed as stating the probability, according to the neuron, that the given input is in class 1 rather than class 0.
**Learning rule.** The teacher supplies a target value \( t \in \{0, 1\} \) which says what the correct answer is for the given input. We compute the error signal

\[
e = t - y
\]

then adjust the weights \( w \) in a direction that would reduce the magnitude of this error:

\[
\Delta w_i = \eta e x_i,
\]

where \( \eta \) is the ‘learning rate’. Commonly \( \eta \) is set by trial and error to a constant value or to a decreasing function of simulation time \( \tau \) such as \( \eta_0/\tau \).

The activity rule and learning rule are repeated for each input/target pair \((x, t)\) that is presented. If there is a fixed data set of size \( N \), we can cycle through the data multiple times.

**Batch learning versus on-line learning**

Here we have described the on-line learning algorithm, in which a change in the weights is made after every example is presented. An alternative paradigm is to go through a batch of examples, computing the outputs and errors and accumulating the changes specified in equation (39.17) which are then made at the end of the batch.

**Batch learning for the single neuron classifier**

For each input/target pair \((x^{(n)}, t^{(n)})\) \((n = 1, \ldots, N)\), compute

\[y^{(n)} = y(x^{(n)}; w),\]

where

\[y(x; w) = \frac{1}{1 + \exp(-\sum_i w_i x_i)},\]

define \( e^{(n)} = t^{(n)} - y^{(n)} \), and compute for each weight \( w_i \)

\[g_i^{(n)} = -e^{(n)} x_i^{(n)}.\]

Then let

\[\Delta w_i = -\eta \sum_n g_i^{(n)}.\]

This batch learning algorithm is a gradient descent algorithm, whereas the on-line algorithm is a stochastic gradient descent algorithm. Source code implementing batch learning is given in algorithm 39.5. This algorithm is demonstrated in figure 39.4 for a neuron with two inputs with weights \( w_1 \) and \( w_2 \) and a bias \( w_0 \), performing the function

\[y(x; w) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}.\]

The bias \( w_0 \) is included, in contrast to figure 39.3, where it was omitted. The neuron is trained on a data set of ten labelled examples.
39.3: Training the single neuron as a binary classifier

Figure 39.4. A single neuron learning to classify by gradient descent. The neuron has two weights $w_1$ and $w_2$ and a bias $w_0$. The learning rate was set to $\eta = 0.01$ and batch-mode gradient descent was performed using the code displayed in algorithm 39.5. (a) The training data. (b) Evolution of weights $w_0$, $w_1$ and $w_2$ as a function of number of iterations (on log scale). (c) Evolution of weights $w_1$ and $w_2$ in weight space. (d) The objective function $G(w)$ as a function of number of iterations. (e) The magnitude of the weights $E_W(w)$ as a function of time. (f-k) The function performed by the neuron (shown by three of its contours) after 30, 80, 500, 3000, 10000 and 40000 iterations. The contours shown are those corresponding to $a = 0, \pm 1$, namely $y = 0.5, 0.27$ and 0.73. Also shown is a vector proportional to $(w_1, w_2)$. The larger the weights are, the bigger this vector becomes, and the closer together are the contours.
Algorithm 39.5. Octave source code for a gradient descent optimizer of a single neuron, batch learning, with optional weight decay (rate \( \alpha \)).

Octave notation: the instruction \( a = x \ast w \) causes the \((N \times I)\) matrix \( x \) consisting of all the input vectors to be multiplied by the weight vector \( w \), giving the vector \( a \) listing the activations for all \( N \) input vectors; \( x' \) means \( x \)-transpose; the single command \( y = \text{sigmoid}(a) \) computes the sigmoid function of all elements of the vector \( a \).

```octave
global x; # x is an N * I matrix containing all the input vectors
global t; # t is a vector of length N containing all the targets
for l = 1:L  # loop L times
    a = x * w;  # compute all activations
    y = sigmoid(a);  # compute outputs
    e = t - y;  # compute errors
    g = - x' * e;  # compute the gradient vector
    w = w - eta * ( g + alpha * w );  # make step, using learning rate eta and weight decay alpha
endfor
function f = sigmoid(v)
    f = 1.0 ./ ( 1.0 .+ exp ( -v ) ) ;
endfunction
```

Figure 39.6. The influence of weight decay on a single neuron’s learning. The objective function is \( M(w) = G(w) + \alpha E_W(w) \). The learning method was as in figure 39.4. (a) Evolution of weights \( w_0 \), \( w_1 \) and \( w_2 \). (b) Evolution of weights \( w_1 \) and \( w_2 \) in weight space shown by points, contrasted with the trajectory followed in the case of zero weight decay, shown by a thin line (from figure 39.4). Notice that for this problem weight decay has an effect very similar to ‘early stopping’. (c) The objective function \( M(w) \) and the error function \( G(w) \) as a function of number of iterations. (d) The function performed by the neuron after 40000 iterations.
39.4 Beyond descent on the error function: regularization

If the parameter $\eta$ is set to an appropriate value, this algorithm works: the algorithm finds a setting of $w$ which correctly classifies as many of the examples as possible.

If the examples are in fact linearly separable then the neuron finds this linear separation and its weights diverge to ever-larger values as the simulation continues. This can be seen happening in figure 39.4(f–k). This is an example of overfitting, where a model fits the data so well that its generalization performance is likely to be adversely affected.

This behaviour may be viewed as undesirable. How can it be rectified?

An ad hoc solution to overfitting is to use early stopping, that is, use an algorithm originally intended to minimize the error function $G(w)$, then prevent it from doing so by halting the algorithm at some point.

A more principled solution to overfitting makes use of regularization. Regularization involves modifying the objective function in such a way as to incorporate a bias against the sorts of solution $w$ which we dislike. In the above example, what we dislike is the development of a very sharp decision boundary in figure 39.4k; this sharp boundary is associated with large weight values, so we use a regularizer that penalizes large weight values. We modify the objective function to:

$$M(w) = G(w) + \alpha E_W(w)$$

where the simplest choice of regularizer is the weight decay regularizer

$$E_W(w) = \frac{1}{2} \sum_i w_i^2.$$  \hspace{1cm} (39.23)

The regularization constant $\alpha$ is called the weight decay rate. This additional term favours small values of $w$ and decreases the tendency of a model to overfit fine details of the training data. The quantity $\alpha$ is known as a hyperparameter. Hyperparameters play a role in the learning algorithm but play no role in the activity rule of the network.

Exercise 39.3. Compute the derivative of $M(w)$ with respect to $w_i$. Why is the above regularizer known as the ‘weight decay’ regularizer?

The gradient descent source code of algorithm 39.5 implements weight decay. This gradient descent algorithm is demonstrated in figure 39.6 using weight decay rates $\alpha = 0.01, 0.1$ and $1$. As the weight decay rate is increased the solution becomes biased towards broader sigmoid functions with decision boundaries that are closer to the origin.

Note

Gradient descent with a step size $\eta$ is in general not the most efficient way to minimize a function. A modification of gradient descent known as momentum, while improving convergence, is also not recommended. Most neural network experts use more advanced optimizers such as conjugate gradient algorithms. [Please do not confuse momentum, which is sometimes given the symbol $\alpha$, with weight decay.]
39.5 Further exercises

More motivations for the linear neuron.

Exercise 39.4. Consider the task of recognizing which of two Gaussian distributions a vector \( z \) comes from. Unlike the case studied in section 11.2, where the distributions had different means but a common variance-covariance matrix, we will assume that the two distributions have exactly the same mean but different variances. Let the probability of \( z \) given \( s \in \{0, 1\} \) be

\[
P(z|s) = \prod_{i=1}^{l} \text{Normal}(z_i; 0, \sigma_{z_i}^2),
\]

(39.24)

where \( \sigma_{z_i}^2 \) is the variance of \( z_i \) when the source symbol is \( s \). Show that \( P(s=1|z) \) can be written in the form

\[
P(s=1|z) = \frac{1}{1 + \exp(-w^T x + \theta)},
\]

(39.25)

where \( x_i \) is an appropriate function of \( z_i \), \( x_i = g(z_i) \).

Exercise 39.5. The noisy LED.

Consider an LED display with 7 elements numbered as shown above. The state of the display is a vector \( x \). When the controller wants the display to show character number \( s \), e.g. \( s = 2 \), each element \( x_j \) (\( j = 1, \ldots, 7 \)) either adopts its intended state \( c_j(s) \), with probability \( 1 - f \), or is flipped, with probability \( f \). Let’s call the two states of \( x \) ‘+1’ and ‘−1’.

(a) Assuming that the intended character \( s \) is actually a 2 or a 3, what is the probability of \( s \), given the state \( x \)? Show that \( P(s=2|x) \) can be written in the form

\[
P(s=2|x) = \frac{1}{1 + \exp(-w^T x + \theta)},
\]

(39.26)

and compute the values of the weights \( w \) in the case \( f = 0.1 \).

(b) Assuming that \( s \) is one of \( \{0, 1, 2, \ldots, 9\} \), with prior probabilities \( p_s \), what is the probability of \( s \), given the state \( x \)? Put your answer in the form

\[
P(s|x) = \sum_{s'} \frac{e^{a_s}}{\sum_{s'} e^{a_s}},
\]

(39.27)

where \( \{a_s\} \) are functions of \( \{c_j(s)\} \) and \( x \).

Could you make a better alphabet of 10 characters for a noisy LED, i.e., an alphabet less susceptible to confusion?

Exercise 39.6. A \((3, 1)\) error-correcting code consists of the two codewords \( x^{(1)} = (1, 0, 0) \) and \( x^{(2)} = (0, 0, 1) \). A source bit \( s \in \{1, 2\} \) having probability distribution \( \{p_1, p_2\} \) is used to select one of the two codewords for transmission over a binary symmetric channel with noise level \( f \). The

Table 39.7. An alternative 15-character alphabet for the 7-element LED display.
received vector is \( \mathbf{r} \). Show that the posterior probability of \( s \) given \( \mathbf{r} \) can be written in the form

\[
P(s = 1 | \mathbf{r}) = \frac{1}{1 + \exp(-w_0 - \sum_{n=1}^{3} w_n r_n)}.
\]

and give expressions for the coefficients \( \{w_n\}_{n=1}^{3} \) and the bias, \( w_0 \).

Describe, with a diagram, how this optimal decoder can be expressed in terms of a ‘neuron’.
Problems to look at before Chapter 40

▷ Exercise 40.1.[2] What is $\sum_{K=0}^{N} \binom{N}{K}$?
[The symbol $\binom{N}{K}$ means the combination $\frac{N!}{K!(N-K)!}$.]

▷ Exercise 40.2.[2] If the top row of Pascal’s triangle (which contains the single number ‘1’) is denoted row zero, what is the sum of all the numbers in the triangle above row $N$?

▷ Exercise 40.3.[2] 3 points are selected at random on the surface of a sphere. What is the probability that all of them lie on a single hemisphere?

This chapter’s material is originally due to Polya (1954) and Cover (1965) and the exposition that follows is Yaser Abu-Mostafa’s.
40

Capacity of a Single Neuron

40.1 Neural network learning as communication

Many neural network models involve the adaptation of a set of weights \( w \) in response to a set of data points, for example a set of \( N \) target values \( D_N = \{t_n\}_{n=1}^N \) at given locations \( \{x_n\}_{n=1}^N \). The adapted weights are then used to process subsequent input data. This process can be viewed as a communication process, in which the sender examines the data \( D_N \) and creates a message \( w \) that depends on those data. The receiver then uses \( w \); for example, the receiver might use the weights to try to reconstruct what the data \( D_N \) was. [In neural network parlance, this is using the neuron for ‘memory’ rather than for ‘generalization’; ‘generalizing’ means extrapolating from the observed data to the value of \( t_{N+1} \) at some new location \( x_{N+1} \).] Just as a disk drive is a communication channel, the adapted network weights \( w \) therefore play the role of a communication channel, conveying information about the training data to a future user of that neural net. The question we now address is, ‘what is the capacity of this channel?’ – that is, ‘how much information can be stored by training a neural network?’

If we had a learning algorithm that either produces a network whose response to all inputs is +1 or a network whose response to all inputs is 0, depending on the training data, then the weights allow us to distinguish between just two sorts of data set. The maximum information such a learning algorithm could convey about the data is therefore 1 bit, this information content being achieved if the two sorts of data set are equiprobable. How much more information can be conveyed if we make full use of a neural network’s ability to represent other functions?

40.2 The capacity of a single neuron

We will look at the simplest case, that of a single binary threshold neuron. We will find that the capacity of such a neuron is two bits per weight. A neuron with \( K \) inputs can store \( 2^K \) bits of information.

To obtain this interesting result we lay down some rules to exclude less interesting answers, such as: ‘the capacity of a neuron is infinite, because each
of its weights is a real number and so can convey an infinite number of bits'.

We exclude this answer by saying that the receiver is not able to examine the weights directly, nor is the receiver allowed to probe the weights by observing the output of the neuron for arbitrarily chosen inputs. We constrain the receiver to observe the output of the neuron at the same fixed set of \( N \) points \( \{x_n\} \) that were in the training set. What matters now is how many different distinguishable functions our neuron can produce, given that we can only observe the function at these \( N \) points. How many different binary labellings of \( N \) points can a linear threshold function produce? And how does this number compare with the maximum possible number of binary labellings, \( 2^N \)? If nearly all of the \( 2^N \) labellings can be realized by our neuron, then it is a communication channel that can convey all \( N \) bits (the target values \( \{t_n\} \)) with small probability of error. We will identify the capacity of the neuron as the maximum value that \( N \) can have such that the probability of error is very small. [We are departing a little from the definition of capacity in Chapter 9.]

We thus examine the following scenario. The sender is given a neuron with \( K \) inputs and a data set \( D_N \) which is a labelling of \( N \) points. The sender uses an adaptive algorithm to try to find a \( w \) that can reproduce this labelling exactly. We will assume the algorithm finds such a \( w \) if it exists. The receiver then evaluates the threshold function on the \( N \) input values. What is the probability that all \( N \) bits are correctly reproduced? How large can \( N \) become, for a given \( K \), without this probability becoming substantially less than one?

**General position**

One technical detail needs to be pinned down: what set of inputs \( \{x_n\} \) are we considering? Our answer might depend on this choice. We will assume that the points are in *general position* (p. 484), which means in \( K = 3 \) dimensions, for example, that no three points are colinear and no four points are coplanar.

**Definition 40.1** A set of points \( \{x_n\} \) in \( K \)-dimensional space are in general position if any subset of size \( \leq K \) is linearly independent.

In \( K = 3 \) dimensions, for example, a set of points are in general position if no three points are colinear and no four points are coplanar. The intuitive idea is that points in general position are like random points in the space, in terms of the linear dependences between points. You don’t expect three random points in three dimensions to lie on a straight line.

**The linear threshold function**

The neuron we will consider performs the function

\[
y = f \left( \sum_{k=1}^{K} w_k x_k \right)
\]  

where

\[
f(a) = \begin{cases} 
1 & a > 0 \\
0 & a \leq 0. 
\end{cases}
\]  

(40.2)

We will not have a bias \( w_0 \); the capacity for a neuron with a bias can be obtained by replacing \( K \) by \( K + 1 \) in the final result below, i.e., considering one of the inputs to be fixed to 1. (These input points would not then be in general position; the derivation still works.)
40.3 Counting threshold functions

Let us denote by $T(N, K)$ the number of distinct threshold functions on $N$ points in general position in $K$ dimensions. We will derive a formula for $T(N, K)$.

To start with, let us work out a few cases by hand.

In $K = 1$ dimension, for any $N$

The $N$ points lie on a line. By changing the sign of the one weight $w_1$ we can label all points on the right side of the origin 1 and the others 0, or vice versa. Thus there are two distinct threshold functions. $T(N, 1) = 2$.

With $N = 1$ point, for any $K$

If there is just one point $x^{(1)}$ then we can realize both possible labellings by setting $w = \pm x^{(1)}$. Thus $T(1, K) = 2$.

In $K = 2$ dimensions

In two dimensions with $N$ points, we are free to spin the separating line around the origin. Each time the line passes over a point we obtain a new function. Once we have spun the line through 360 degrees we reproduce the function we started from. Because the points are in general position, the separating plane (line) crosses only one point at a time. In one revolution, every point is passed over twice. There are therefore $2N$ distinct threshold functions. $T(N, 2) = 2N$.

Comparing with the total number of binary functions, $2^N$, we may note that for $N \geq 3$, not all binary functions can be realized by a linear threshold function. One famous example of an unrealizable function with $N = 4$ and $K = 2$ is the exclusive-or function on the points $x = (\pm 1, \pm 1)$. [These points are not in general position, but you may confirm that the function remains unrealizable even if the points are perturbed into general position.]

In $K = 2$ dimensions, from the point of view of weight space

There is another way of visualizing this problem. Instead of visualizing a plane separating points in the two-dimensional input space, we can consider the two-dimensional weight space, colouring regions in weight space different colours if they label the given datapoints differently. We can then count the number of threshold functions by counting how many distinguishable regions there are in weight space. Consider first the set of weight vectors in weight space.
space that classify a particular example $x^{(n)}$ as a 1. For example, figure 40.2a shows a single point in our two-dimensional $x$-space, and figure 40.2b shows the two corresponding sets of points in $w$-space. One set of weight vectors occupy the half space

$$ x^{(n)}.w > 0, $$(40.3)

and the others occupy $x^{(n)}.w < 0$. In figure 40.3a we have added a second point in the input space. There are now 4 possible labellings: (1,1), (1,0), (0,1), and (0,0). Figure 40.3b shows the two hyperplanes $x^{(1)}.w = 0$ and $x^{(2)}.w = 0$ which separate the sets of weight vectors that produce each of these labellings. When $N = 3$ (figure 40.4), weight space is divided by three hyperplanes into six regions. Not all of the eight conceivable labellings can be realized. Thus $T(3,2) = 6$.

In $K = 3$ dimensions

We now use this weight space visualization to study the three dimensional case.

Let us imagine adding one point at a time and count the number of threshold functions as we do so. When $N = 2$, weight space is divided by two hyperplanes $x^{(1)}.w = 0$ and $x^{(2)}.w = 0$ into four regions; in any one region all vectors $w$ produce the same function on the 2 input vectors. Thus $T(2,3) = 4$.

Adding a third point in general position produces a third plane in $w$ space, so that there are 8 distinguishable regions. $T(3,3) = 8$. The three bisecting planes are shown in figure 40.5a.

At this point matters become slightly more tricky. As figure 40.5b illustrates, the fourth plane in the three-dimensional $w$ space cannot transect all eight of the sets created by the first three planes. Six of the existing regions are cut in two and the remaining two are unaffected. So $T(4,3) = 14$. Two
40.3: Counting threshold functions

Figure 40.5. Weight space illustrations for $T(3, 3)$ and $T(4, 3)$. (a) $T(3, 3) = 8$. Three hyperplanes (corresponding to three points in general position) divide 3-space into 8 regions, shown here by colouring the relevant part of the surface of a hollow, semi-transparent cube centred on the origin. (b) $T(4, 3) = 14$. Four hyperplanes divide 3-space into 14 regions, of which this figure shows 13 (the 14th region is out of view on the right-hand face. Compare with figure 40.5a: all of the regions that are not coloured white have been cut into two.

<table>
<thead>
<tr>
<th>$N$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 40.6. Values of $T(N, K)$ deduced by hand.

Figure 40.7. Illustration of the cutting process going from $T(3, 3)$ to $T(4, 3)$. (a) The eight regions of figure 40.5a with one added hyperplane. All of the regions that are not coloured white have been cut into two. (b) Here, the hollow cube has been made solid, so we can see which regions are cut by the fourth plane. The front half of the cube has been cut away. (c) This figure shows the new two dimensional hyperplane, which is divided into six regions by the three one-dimensional hyperplanes (lines) which cross it. Each of these regions corresponds to one of the three-dimensional regions in figure 40.7a which is cut into two by this new hyperplane. This shows that $T(4, 3) - T(3, 3) = 6$. Figure 40.7c should be compared with figure 40.4b.
of the binary functions on 4 points in 3 dimensions cannot be realized by a linear threshold function.

We have now filled in the values of $T(N, K)$ shown in table 40.6. Can we obtain any insights into our derivation of $T(4, 3)$ in order to fill in the rest of the table for $T(N, K)$? Why was $T(4, 3)$ greater than $T(3, 3)$ by six?

Six is the number of regions that the new hyperplane bisected in $w$-space (figure 40.7a,b). Equivalently, if we look in the $K-1$ dimensional subspace that is the $N$th hyperplane, that subspace is divided into six regions by the $N-1$ previous hyperplanes (figure 40.7c). Now this is a concept we have met before. Compare figure 40.7c with figure 40.4b. How many regions are created by $N-1$ hyperplanes in a $K-1$ dimensional space? Why, $T(N-1, K-1)$, of course! In the present case $N = 4$, $K = 3$, we can look up $T(3, 2) = 6$ in the previous section. So

$$T(4, 3) = T(3, 3) + T(3, 2). \quad (40.4)$$

**Recurrence relation for any $N, K$**

Generalizing this picture, we see that when we add an $N$th hyperplane in $K$ dimensions, it will bisect $T(N-1, K-1)$ of the $T(N-1, K)$ regions that were created by the previous $N-1$ hyperplanes. Therefore, the total number of regions obtained after adding the $N$th hyperplane is $2T(N-1, K-1)$ (since $T(N-1, K-1)$ out of $T(N-1, K)$ regions are split in two) plus the remaining $T(N-1, K) - T(N-1, K-1)$ regions not split by the $N$th hyperplane, which gives the following equation for $T(N, K)$:

$$T(N, K) = T(N-1, K) + T(N-1, K-1). \quad (40.5)$$

Now all that remains is to solve this recurrence relation given the boundary conditions $T(1, 1) = 2$ and $T(1, K) = 2$.

Does the recurrence relation (40.5) look familiar? Maybe you remember building Pascal’s triangle by adding together two adjacent numbers in one row to get the number below. The $N, K$ element of Pascal’s triangle is equal to

$$C(N, K) \equiv \binom{N}{K} = \frac{N!}{(N-K)!K!}. \quad (40.6)$$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1$</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2$</td>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3$</td>
<td></td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4$</td>
<td></td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$5$</td>
<td></td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 40.8. Pascal’s triangle.

Combinations $\binom{N}{K}$ satisfy the equation

$$C(N, K) = C(N-1, K-1) + C(N-1, K), \quad \text{for all } N > 0. \quad (40.7)$$

[Here we are adopting the convention that $\binom{N}{K} \equiv 0$ if $K > N$ or $K < 0$.]

So $\binom{N}{K}$ satisfies the required recurrence relation (40.5). This doesn’t mean $T(N, K) = \binom{N}{K}$, since many functions can satisfy one recurrence relation.
40.3: Counting threshold functions

Figure 40.9. The fraction of functions on $N$ points in $K$ dimensions that are linear threshold functions, $T(N,K)/2^N$, shown from various viewpoints. In (a) we see the dependence on $K$, which is approximately an error function passing through 0.5 at $K = N/2$; the fraction reaches 1 at $K = N$. In (b) we see the dependence on $N$, which is 1 up to $N = K$ and drops sharply at $N = 2K$. Panel (c) shows the dependence on $N/K$ for $K = 1000$. There is a sudden drop in the fraction of realizable labellings when $N = 2K$. Panel (d) shows the values of $\log_2 T(N,K)$ and $\log_2 2^N$ as a function of $N$ for $K = 1000$. These figures were plotted using the approximation of $T = 2^N$ by the error function.

But perhaps we can express $T(N,K)$ as a linear superposition of combination functions of the form $C_{\alpha,\beta}(N,K) \equiv \binom{N}{K+\beta}$. By comparing tables 40.8 and 40.6 we can see how to satisfy the boundary conditions: we simply need to translate Pascal’s triangle to the right by 1, 2, 3, …; superpose; add; multiply by two, and drop the whole table by one line. Thus:

$$T(N,K) = 2 \sum_{k=0}^{K-1} \binom{N-1}{k}.$$  \hfill (40.8)

Using the fact that the $N$th row of Pascal’s triangle sums to $2^N$, that is, $\sum_{k=0}^{N-1} \binom{N-1}{k} = 2^{N-1}$, we can simplify the cases where $K-1 \geq N-1$:

$$T(N,K) = \begin{cases} 2^N & K \geq N \\ 2\sum_{k=0}^{K-1} \binom{N-1}{k} & K < N. \end{cases}$$ \hfill (40.9)

**Interpretation**

It is natural to compare $T(N,K)$ with the total number of binary functions on $N$ points, $2^N$. The ratio $T(N,K)/2^N$ tells us the probability that an arbitrary labelling $\{t_n\}_{n=1}^N$ can be memorized by our neuron. The two functions are equal for all $N \leq K$. The line $N = K$ is thus a special line, defining the maximum number of points on which any arbitrary labelling can be realized. This number of points is referred to as the Vapnik–Chervonenkis dimension (VC dimension) of the class of functions. The VC dimension of a binary threshold function on $K$ dimensions is thus $K$. 
What is interesting is (for large $K$) the number of points $N$ such that almost any labelling can be realized. The ratio $T(N, K)/2^N$ is, for $N < 2K$, still greater than $1/2$, and for large $K$ the ratio is very close to 1.

For our purposes the sum in equation (40.9) is well approximated by the error function,

$$
\sum_{k=0}^{K} \binom{N}{k} \approx 2^N \Phi \left( \frac{K - (N/2)}{\sqrt{N/2}} \right),
$$

(40.10)

where $\Phi(z) = \int_{-\infty}^{z} \exp(-z^2/2)/\sqrt{2\pi}$. Figure 40.9 shows the realizability fraction $T(N, K)/2^N$ as a function of $N$ and $K$. The take-home message is shown in figure 40.9c: although the fraction $T(N, K)/2^N$ is less than 1 for $N > K$, it is only negligibly less than 1 up to $N = 2K$; there, there is a catastrophic drop to zero, so that for $N > 2K$, only a tiny fraction of the binary labellings can be realized by the threshold function.

**Conclusion**

The capacity of a linear threshold neuron, for large $K$, is 2 bits per weight.

A single neuron can almost certainly memorize up to $N = 2K$ random binary labels perfectly, but will almost certainly fail to memorize more.

### 40.4 Further exercises

- **Exercise 40.4.** Can a finite set of $2N$ distinct points in a two-dimensional space be split in half by a straight line

  - if the points are in general position?
  - if the points are not in general position?

Can $2N$ points in a $K$ dimensional space be split in half by a $K - 1$ dimensional hyperplane?

- **Exercise 40.5.** Four points are selected at random on the surface of a sphere. What is the probability that all of them lie on a single hemisphere? How does this question relate to $T(N, K)$?

- **Exercise 40.6.** Consider the binary threshold neuron in $K = 3$ dimensions, and the set of points $\{x\} = \{(1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 1, 1)\}$. Find a parameter vector $\textbf{w}$ such that the neuron memorizes the labels: (a) $\{t\} = \{1, 1, 1, 1\}$; (b) $\{t\} = \{1, 1, 0, 0\}$.

Find an unrealizable labelling $\{t\}$.

- **Exercise 40.7.** In this chapter we constrained all our hyperplanes to go through the origin. In this exercise, we remove this constraint.

  How many regions in a plane are created by $N$ lines in general position?

- **Exercise 40.8.** Estimate in bits the total sensory experience that you have had in your life – visual information, auditory information, etc. Estimate how much information you have memorized. Estimate the information content of the works of Shakespeare. Compare these with the capacity of your brain assuming you have $10^{11}$ neurons each making 1000 synaptic connections, and that the capacity result for one neuron (two bits per connection) applies. Is your brain full yet?
Exercise 40.9. What is the capacity of the axon of a spiking neuron, viewed as a communication channel, in bits per second? [See MacKay and McCulloch (1952) for an early publication on this topic.] Multiply by the number of axons in the optic nerve (about $10^6$) or cochlear nerve (about 50,000 per ear) to estimate again the rate of acquisition sensory experience.

40.5 Solutions

Solution to exercise 40.5 (p.490). The probability that all four points lie on a single hemisphere is

$$T(4, 3)/2^4 = 14/16 = 7/8.$$  (40.11)
41 Learning as Inference

41.1 Neural network learning as inference

In Chapter 39 we trained a simple neural network as a classifier by minimizing an objective function

\[ M(w) = G(w) + \alpha E_W(w) \]  \hspace{1cm} (41.1)

made up of an error function

\[ G(w) = -\sum_t \left[ t^{(n)} \ln y(x^{(n)}; w) + (1 - t^{(n)}) \ln (1 - y(x^{(n)}; w)) \right] \]  \hspace{1cm} (41.2)

and a regularizer

\[ E_W(w) = \frac{1}{2} \sum_i w_i^2. \]  \hspace{1cm} (41.3)

This neural network learning process can be given the following probabilistic interpretation.

We interpret the output \( y(x; w) \) of the neuron literally as defining (when its parameters \( w \) are specified) the probability that an input \( x \) belongs to class \( t = 1 \), rather than the alternative \( t = 0 \). Thus \( y(x; w) \equiv P(t = 1 | x, w) \). Then each value of \( w \) defines a different hypothesis about the probability of class 1 relative to class 0 as a function of \( x \).

We define the observed data \( D \) to be the targets \( \{t\} \) – the inputs \( \{x\} \) are assumed to be given, and not to be modelled. To infer \( w \) given the data, we require a likelihood function and a prior probability over \( w \). The likelihood function measures how well the parameters \( w \) predict the observed data; it is the probability assigned to the observed \( t \) values by the model with parameters set to \( w \). Now the two equations

\[ P(t = 1 | w, x) = y \]
\[ P(t = 0 | w, x) = 1 - y \]

can be rewritten as the single equation

\[ P(t | w, x) = y^t (1 - y)^{1 - t} = \exp[t \ln y + (1 - t) \ln (1 - y)]. \]  \hspace{1cm} (41.5)

So the error function \( G \) can be interpreted as minus the log likelihood:

\[ P(D | w) = \exp[-G(w)]. \]  \hspace{1cm} (41.6)

Similarly the regularizer can be interpreted in terms of a log prior probability distribution over the parameters:

\[ P(w | \alpha) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W). \]  \hspace{1cm} (41.7)
41.2: Illustration for a neuron with two weights

If $E_W$ is quadratic as defined above, then the corresponding prior distribution is a Gaussian with variance $\sigma_W^2 = 1/\alpha$, and $1/Z_W(\alpha)$ is equal to $(\alpha/2\pi)^{K/2}$, where $K$ is the number of parameters in the vector $w$.

The objective function $M(w)$ then corresponds to the inference of the parameters $w$, given the data:

$$
P(w|D, \alpha) = \frac{P(D|w)P(w|\alpha)}{P(D|\alpha)}
= \frac{e^{-G(w)} e^{-\alpha E_W(w)}/Z_W(\alpha)}{P(D|\alpha)}
= \frac{1}{Z_M} \exp(-M(w)).
$$

So the $w$ found by (locally) minimizing $M(w)$ can be interpreted as the (locally) most probable parameter vector, $w^*$. From now on we will refer to $w^*$ as $w_{MP}$.

Why is it natural to interpret the error functions as log probabilities? Error functions are usually additive. For example, $G$ is a sum of information contents, and $E_W$ is a sum of squared weights. Probabilities, on the other hand, are multiplicative: for independent events $X$ and $Y$, the joint probability is $P(x, y) = P(x)P(y)$. The logarithmic mapping maintains this correspondence.

The interpretation of $M(w)$ as a log probability has numerous benefits, some of which we will discuss in a moment.

41.2 Illustration for a neuron with two weights

In the case of a neuron with just two inputs and no bias,

$$
y(x; w) = \frac{1}{1 + e^{-(w_1x_1 + w_2x_2)}},
$$

we can plot the posterior probability of $w$, $P(w|D, \alpha) \propto \exp(-M(w))$. Imagine that we receive some data as shown in the left column of figure 41.1. Each data point consists of a two dimensional input vector $x$ and a $t$ value indicated by $\times (t = 1)$ or $\square (t = 0)$. The likelihood function $\exp(-G(w))$ is shown as a function of $w$ in the second column. It is a product of functions of the form (41.11).

The product of traditional learning is a point in $w$-space, the estimator $w^*$, which maximizes the posterior probability density. In contrast, in the Bayesian view, the product of learning is an ensemble of plausible parameter values (bottom right of figure 41.1). We do not choose one particular hypothesis $w$; rather we evaluate their posterior probabilities. The posterior distribution is obtained by multiplying the likelihood by a prior distribution over $w$ space (shown as a broad Gaussian at the upper right of figure 41.1). The posterior ensemble (within a multiplicative constant) is shown in the third column of figure 41.1, and as a contour plot in the fourth column. As the amount of data increases (from top to bottom), the posterior ensemble becomes increasingly concentrated around the most probable value $w^*$.

41.3 Beyond optimization: making predictions

Let us consider the task of making predictions with the neuron which we trained as a classifier in section 39.3. This was a neuron with two inputs and a bias.

$$
y(x; w) = \frac{1}{1 + e^{-(w_0 + w_1x_1 + w_2x_2)}}.
$$
Figure 41.1. The Bayesian interpretation and generalization of traditional neural network learning. Evolution of the probability distribution over parameters as data arrive.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Likelihood</th>
<th>Probability of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 0</td>
<td>(constant)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="image1.png" alt="Graph" /></td>
<td><img src="image2.png" alt="Graph" /></td>
</tr>
<tr>
<td>N = 2</td>
<td><img src="image3.png" alt="Graph" /></td>
<td><img src="image4.png" alt="Graph" /></td>
</tr>
<tr>
<td>N = 4</td>
<td><img src="image5.png" alt="Graph" /></td>
<td><img src="image6.png" alt="Graph" /></td>
</tr>
<tr>
<td>N = 6</td>
<td><img src="image7.png" alt="Graph" /></td>
<td><img src="image8.png" alt="Graph" /></td>
</tr>
</tbody>
</table>

Figure 41.2. Making predictions. (a) The function performed by an optimized neuron \( w_{MP} \) (shown by three of its contours) trained with weight decay, \( \alpha = 0.01 \) (from figure 39.6). The contours shown are those corresponding to \( \alpha = 0, \pm 1 \), namely \( y = 0.5, 0.27 \) and \( 0.73 \). (b) Are these predictions more reasonable? (Contours shown are for \( y = 0.5, 0.27, 0.73, 0.12 \) and \( 0.88 \).) (c) The posterior probability of \( w \) (schematic); the Bayesian predictions shown in (b) were obtained by averaging together the predictions made by each possible value of the weights \( w \), with each value of \( w \) receiving a vote proportional to its probability under the posterior ensemble. The method used to create (b) is described in section 41.4.
When we last played with it, we trained it by minimizing the objective function
\[ M(w) = G(w) + \alpha E(w). \tag{41.13} \]

The resulting optimized function for the case \( \alpha = 0.01 \) is reproduced in figure 41.2a.

We now consider the task of predicting the class \( t^{(N+1)} \) corresponding to a new input \( \mathbf{x}^{(N+1)} \). It is common practice, when making predictions, simply to use a neural network with its weights fixed to their optimized value \( w_{\text{MP}} \), but this is not optimal, as can be seen intuitively by considering the predictions shown in figure 41.2a. Are these reasonable predictions? Consider new data arriving at points A and B. The best fit model assigns both of these examples probability 0.2 of being in class 1, because they have the same value of \( w_{\text{MP}} \cdot \mathbf{x} \). If we really knew that \( w \) was equal to \( w_{\text{MP}} \), then these predictions would be correct. But we do not know \( w \). The parameters are uncertain. Intuitively we might be inclined to assign a less confident probability (closer to 0.5) at B than at A, as shown in figure 41.2b, since point B is far from the training data. The best fit parameters \( w_{\text{MP}} \) often give over-confident predictions.

A non-Bayesian approach to this problem is to downweight all predictions uniformly, by an empirically determined factor (Copas, 1983). This is not ideal, since intuition suggests the strength of the predictions at B should be downweighted more than those at A. A Bayesian viewpoint helps us to understand the cause of the problem, and provides a straightforward solution. In a nutshell, we obtain Bayesian predictions by taking into account the whole posterior ensemble, shown schematically in figure 41.2c.

The Bayesian prediction of a new datum \( t^{(N+1)} \) involves marginalizing over the parameters (and over anything else about which we are uncertain). For simplicity, let us assume that the weights \( w \) are the only uncertain quantities – the weight decay rate \( \alpha \) and the model \( H \) itself are assumed to be fixed. Then by the sum rule, the predictive probability of a new target \( t^{(N+1)} \) at a location \( \mathbf{x}^{(N+1)} \) is:
\[ P(t^{(N+1)} | \mathbf{x}^{(N+1)}, D, \alpha) = \int d^K w P(t^{(N+1)} | \mathbf{x}^{(N+1)}, w, \alpha) P(w | D, \alpha), \tag{41.14} \]
where \( K \) is the dimensionality of \( w \), three in the toy problem. Thus the predictions are obtained by weighting the prediction for each possible \( w \),
\[ P(t^{(N+1)} = 1 | \mathbf{x}^{(N+1)}, w, \alpha) = y(\mathbf{x}^{(N+1)}; w), \]
\[ P(t^{(N+1)} = 0 | \mathbf{x}^{(N+1)}, w, \alpha) = 1 - y(\mathbf{x}^{(N+1)}; w), \tag{41.15} \]
with a weight given by the posterior probability of \( w \), \( P(w | D, \alpha) \), which we most recently wrote down in equation (41.10). This posterior probability is
\[ P(w | D, \alpha) = \frac{1}{Z_M} \exp(-M(w)), \tag{41.16} \]
where
\[ Z_M = \int d^K w \exp(-M(w)). \tag{41.17} \]

In summary, we can get the Bayesian predictions if we can find a way of computing the integral
\[ P(t^{(N+1)} = 1 | \mathbf{x}^{(N+1)}, D, \alpha) = \int d^K w \ y(\mathbf{x}^{(N+1)}; w) \frac{1}{Z_M} \exp(-M(w)), \tag{41.18} \]
which is the average of the output of the neuron at \( \mathbf{x}^{(N+1)} \) under the posterior distribution of \( w \).
Implementation

How shall we compute the integral (41.18)? For our toy problem, the weight space is three dimensional; for a realistic neural network the dimensionality $K$ might be in the thousands.

Bayesian inference for general data modelling problems may be implemented by exact methods (Chapter 25), by Monte Carlo sampling (Chapter 29), or by deterministic approximate methods, for example, methods that make Gaussian approximations to $P(w|D, \alpha)$ using Laplace’s method (Chapter 27) or variational methods (Chapter 33). For neural networks there are few exact methods. The two main approaches to implementing Bayesian inference for neural networks are the Monte Carlo methods developed by Neal (1996) and the Gaussian approximation methods developed by MacKay (1991).

41.4 Monte Carlo implementation of a single neuron

First we will use a Monte Carlo approach in which the task of evaluating the integral (41.18) is solved by treating $y(x^{(N+1)}; w)$ as a function $f$ of $w$ whose mean we compute using

$$\langle f(w) \rangle \simeq \frac{1}{R} \sum_r f(w^{(r)})$$

(41.19)

where \{w^{(r)}\} are samples from the posterior distribution $\frac{1}{Z_M}\exp(-M(w))$ (c.f. equation (29.6)). We obtain the samples using a Metropolis method (section 29.4). As an aside, a possible disadvantage of this Monte Carlo approach is that it is a poor way of estimating the probability of an improbable event, i.e., a $P(t|D,H)$ that is very close to zero, if the improbable event is most likely to occur in conjunction with improbable parameter values.

How to generate the samples $\{w^{(r)}\}$? Radford Neal introduced the Hamiltonian Monte Carlo method to neural networks. We met this sophisticated Metropolis method, which makes use of gradient information, in Chapter 30. The method we now demonstrate is a simple version of Hamiltonian Monte Carlo called the Langevin Monte Carlo method.

The Langevin Monte Carlo method

The Langevin method (algorithm 41.4) may be summarized as ‘gradient descent with added noise’, as shown pictorially in figure 41.3. A noise vector $p$ is generated from a Gaussian with unit variance. The gradient $g$ is computed,
41.4: Monte Carlo implementation of a single neuron

Algorithm 41.4. Octave source code for the Langevin Monte Carlo method. To obtain the Hamiltonian Monte Carlo method, we repeat the four lines marked * multiple times (algorithm 41.8).

\[
g = \text{gradM}(w); \quad # \text{set gradient using initial } w \\
M = \text{findM}(w); \quad # \text{set objective function too} \\
\text{for } l = 1:L \quad # \text{loop } L \text{ times} \\
P = \text{randn}(\text{size}(w)); \quad # \text{initial momentum is Normal}(0,1) \\
H = P' * P / 2 + M; \quad # \text{evaluate } H(w,p) \\
* \quad P = P - \epsilon * g / 2; \quad # \text{make half-step in } p \\
* \quad w' = w + \epsilon * P; \quad # \text{make step in } w \\
* \quad g = \text{gradM}(w'); \quad # \text{find new gradient} \\
* \quad P = P - \epsilon * g / 2; \quad # \text{make half-step in } p \\
M' = \text{findM}(w'); \quad # \text{find new objective function} \\
H' = P' * P / 2 + M'; \quad # \text{evaluate new value of } H \\
\text{if} ( H' < H ) \quad # \text{decide whether to accept} \\
\quad \text{accept} = 1; \\
\quad \text{elseif} ( \text{rand()} < \exp(-H')) \quad \text{accept} = 1; \quad # \text{compare with a uniform variate} \\
\quad \text{else} \quad \text{accept} = 0; \quad \text{endif} \\
\text{if} ( \text{accept} ) \quad g = g'; \quad w = w'; \quad M = M'; \quad \text{endif} \\
\text{endfor} \\
\text{function } g = \text{gradM}(w) \quad # \text{gradient of objective function} \\
\quad a = x * w; \quad # \text{compute activations} \\
\quad y = \text{sigmoid}(a); \quad # \text{compute outputs} \\
\quad e = t - y; \quad # \text{compute errors} \\
\quad g = -x' * e; \quad # \text{compute the gradient of } G(w) \\
\quad g = alpha * v + g; \quad \text{endfunction} \\
\text{function } M = \text{findM}(w) \quad # \text{objective function} \\
\quad G = -(t' * \log(y) + (1-t') * \log(1-y)); \\
\quad EW = u' * u / 2; \\
\quad M = G + alpha * EW; \quad \text{endfunction}
Figure 41.5. A single neuron learning under the Langevin Monte Carlo method. (a) Evolution of weights $w_0$, $w_1$ and $w_2$ as a function of number of iterations. (b) Evolution of weights $w_1$ and $w_2$ in weight space. Also shown by a line is the evolution of the weights using the optimizer of figure 39.6. (c) The error function $G(w)$ as a function of number of iterations. Also shown is the error function during the optimization of figure 39.6. (d) The objective function $M(x)$ as a function of number of iterations. See also figures 41.6 and 41.7.

and a step in $w$ is made, given by

$$\Delta w = -\frac{1}{\eta} g + \epsilon p. \quad (41.20)$$

Notice that if the $\epsilon p$ term were omitted this would simply be gradient descent with learning rate $\eta = \frac{1}{\epsilon} c^2$. This step in $w$ is accepted or rejected depending on the change in the value of the objective function $M(w)$ and on the change in gradient, with a probability of acceptance such that detailed balance holds.

The Langevin method has one free parameter, $\epsilon$, which controls the typical step size. If $\epsilon$ is set to too large a value, moves may be rejected. If it is set to a very small value, progress around the state space will be slow.

Demonstration of Langevin method

The Langevin method is demonstrated in figures 41.5, 41.6 and 41.7. Here, the objective function is $M(w) = G(w) + \alpha E_W(w)$, with $\alpha = 0.01$. These figures include, for comparison, the results of the previous optimization method using gradient descent on the same objective function (figure 39.6). It can be seen that the mean evolution of $w$ is similar to the evolution of the parameters under gradient descent. The Monte Carlo method appears to have converged to the posterior distribution after about 10000 iterations.

The average acceptance rate during this simulation was 93%; only 7% of the proposed moves were rejected. Probably, faster progress around the state space would have been made if a larger step size $\epsilon$ had been used, but the value was chosen so that the ‘descent rate’ $\eta = \frac{1}{2} c^2$ matched the step size of the earlier simulations.

Making Bayesian predictions

From iteration 10,000 to 40,000, the weights were sampled every 1000 iterations and the corresponding functions of $x$ are plotted in figure 41.6. There is a considerable variety of plausible functions. We obtain a Monte Carlo approximation to the Bayesian predictions by averaging these thirty functions of $x$ together. The result is shown in figure 41.7 and contrasted with the predictions given by the optimized parameters. The Bayesian predictions become satisfyingly moderate as we move away from the region of highest data density.
41.4 Monte Carlo implementation of a single neuron

Figure 41.6. Samples obtained by the Langevin Monte Carlo method. The learning rate was set to $\eta = 0.01$ and the weight decay rate to $\alpha = 0.01$. The step size is given by $\epsilon = \sqrt{2\eta}$. The function performed by the neuron is shown by three of its contours every 1000 iterations from iteration 10,000 to 40,000. The contours shown are those corresponding to $a = 0, \pm 1, \pm 2$, namely $y = 0.5, 0.27, 0.73, 0.12$ and 0.88. Also shown is a vector proportional to $(w_1, w_2)$.

Figure 41.7. Bayesian predictions found by the Langevin Monte Carlo method compared with the predictions using the optimized parameters. (a) The predictive function obtained by averaging the predictions for 30 samples uniformly spaced between iterations 10,000 and 40,000, shown in figure 41.6. The contours shown are those corresponding to $a = 0, \pm 1, \pm 2$, namely $y = 0.5, 0.27, 0.73, 0.12$ and 0.88. (b) For contrast, the predictions given by the ‘most probable’ setting of the neuron’s parameters, as given by optimization of $M(w)$.
wnew = w ;
gnew = g ;
for tau = 1:Tau
    p = p - epsilon * gnew / 2 ; # make half-step in p
    wnew = wnew + epsilon * p ; # make step in w
    gnew = gradM ( wnew ) ; # find new gradient
    p = p - epsilon * gnew / 2 ; # make half-step in p
endfor

Algorithm 41.8. Octave source code for the Hamiltonian Monte Carlo method. The algorithm is identical to the Langevin method in algorithm 41.4, except for the replacement of the four lines marked * in that algorithm by the fragment shown here.

The Bayesian classifier is better able to identify the points where the classification is uncertain. This pleasing behaviour results simply from a mechanical application of the rules of probability.

Optimization and typicality

A final observation concerns the behaviour of the functions \( G(w) \) and \( M(w) \) during the Monte Carlo sampling process, compared with the values of \( G \) and \( M \) at the optimum \( w_{MP} \) (figure 41.5). The function \( G(w) \) fluctuates around the value of \( G(w_{MP}) \), though not in a symmetrical way. The function \( M(w) \) also fluctuates, but it does not fluctuate around \( M(w_{MP}) \) – obviously it cannot, because \( M \) is minimized at \( w_{MP} \), so \( M \) could not go any smaller – furthermore, \( M \) only rarely drops close to \( M(w_{MP}) \). In the language of information theory, the typical set of \( w \) has different properties from the most probable state \( w_{MP} \).

A general message therefore emerges – applicable to all data models, not just neural networks: one should be cautious about making use of optimized parameters, as the properties of optimized parameters may be unrepresentative of the properties of typical, plausible parameters; and the predictions obtained using optimized parameters alone will often be unreasonably over-confident.

Reducing random walk behaviour using Hamiltonian Monte Carlo

As a final study of Monte Carlo methods, we now compare the Langevin Monte Carlo method with its big brother, the Hamiltonian Monte Carlo method. The change to Hamiltonian Monte Carlo is simple to implement, as shown in algorithm 41.8. Each single proposal makes use of multiple gradient evaluations.
along a dynamical trajectory in \( w, p \) space, where \( p \) are the extra ‘momentum’ variables of the Langevin and Hamiltonian Monte Carlo methods. The number of steps ‘\( \text{Tau} \)' was set at random to a number between 100 and 200 for each trajectory. The step size \( \epsilon \) was kept fixed so as to retain comparability with the simulations that have gone before; it is recommended that one randomize the step size in practical applications, however.

Figure 41.9 compares the sampling properties of the Langevin and Hamiltonian Monte Carlo methods. The autocorrelation of the state of the Hamiltonian Monte Carlo simulation falls much more rapidly with simulation time than that of the Langevin method. For this toy problem, Hamiltonian Monte Carlo is at least ten times more efficient in its use of computer time.

41.5 Implementing inference with Gaussian approximations

Physicists love to take nonlinearities and locally linearize them, and they love to approximate probability distributions by Gaussians. Such approximations oer an alternative strategy for dealing with the integral

\[
P(t^{(N+1)} | x^{(N+1)}, D, \alpha) = \int d^K w \, y(x^{(N+1)}, w) \frac{1}{Z_M} \exp(-M(w)), \tag{41.21}
\]

which we just evaluated using Monte Carlo methods.

We start by making a Gaussian approximation to the posterior probability. We go to the minimum of \( M(w) \) (using a gradient-based optimizer) and Taylor-expand \( M \) there:

\[
M(w) \approx M(w_{MP}) + \frac{1}{2} (w - w_{MP})^T A (w - w_{MP}) + \cdots, \tag{41.22}
\]

where \( A \) is the matrix of second derivatives, also known as the Hessian, defined by

\[
A_{ij} \equiv \left. \frac{\partial^2}{\partial w_i \partial w_j} M(w) \right|_{w=w_{MP}}. \tag{41.23}
\]

We thus define our Gaussian approximation:

\[
Q(w; w_{MP}, A) = \left[ \det(A/2\pi) \right]^{1/2} \exp\left[ -\frac{1}{2} (w - w_{MP})^T A (w - w_{MP}) \right]. \tag{41.24}
\]

We can think of the matrix \( A \) as defining error bars on \( w \). To be precise, \( Q \) is a normal distribution whose variance–covariance matrix is \( A^{-1} \).

Exercise 41.1.[2] Show that the second derivative of \( M(w) \) with respect to \( w \) is given by

\[
\frac{\partial^2}{\partial w_i \partial w_j} M(w) = \sum_{n=1}^{N} f'(a^{(n)}) x_i^{(n)} x_j^{(n)} + \alpha \delta_{ij}, \tag{41.25}
\]

where \( f'(a) \) is the first derivative of \( f(a) \equiv 1/(1 + e^{-a}) \), which is

\[
f'(a) = \frac{d}{da} f(a) = f(a)(1 - f(a)), \tag{41.26}
\]

and

\[
a^{(n)} = \sum_j w_j x_j^{(n)}. \tag{41.27}
\]

Having computed the Hessian, our task is then to perform the integral (41.21) using our Gaussian approximation.
Calculating the marginalized probability

The output \( y(\mathbf{x}; \mathbf{w}) \) only depends on \( \mathbf{w} \) through the scalar \( a(\mathbf{x}; \mathbf{w}) \), so we can reduce the dimensionality of the integral by finding the probability density of \( a \). We are assuming a locally Gaussian posterior probability distribution over \( \mathbf{w} = \mathbf{w}_{\text{MP}} + \Delta \mathbf{w} \), \( P(\mathbf{w} | D, \alpha) \simeq (1/Z_{Q}) \exp(-\frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w}) \). For our single neuron, the activation \( a(\mathbf{x}; \mathbf{w}) \) is a linear function of \( \mathbf{w} \) with \( \partial a / \partial \mathbf{w} = \mathbf{x} \), so for any \( \mathbf{x} \), the activation \( a \) is Gaussian-distributed.

Exercise 41.2. Assuming \( \mathbf{w} \) is Gaussian-distributed with mean \( \mathbf{w}_{\text{MP}} \) and variance–covariance matrix \( \mathbf{A}^{-1} \), show that the probability distribution of \( a(\mathbf{x}) \) is

\[
P(a | \mathbf{x}, D, \alpha) = \text{Normal}(a_{\text{MP}}, s^2) = \frac{1}{\sqrt{2\pi s^2}} \exp\left(-\frac{(a - a_{\text{MP}})^2}{2s^2}\right),
\]

where \( a_{\text{MP}} = a(\mathbf{x}; \mathbf{w}_{\text{MP}}) \) and \( s^2 = \mathbf{x}^T \mathbf{A}^{-1} \mathbf{x} \).

This means that the marginalized output is:

\[
P(t=1 | \mathbf{x}, D, \alpha) = \psi(a_{\text{MP}}, s^2) = \int da f(a) \text{ Normal}(a_{\text{MP}}, s^2).
\]

This is to be contrasted with \( y(\mathbf{x}; \mathbf{w}_{\text{MP}}) = f(a_{\text{MP}}) \), the output of the most probable network. The integral of a sigmoid times a Gaussian can be approximated by:

\[
\psi(a_{\text{MP}}, s^2) \simeq \phi(a_{\text{MP}}, s^2) = f(\kappa(s)a_{\text{MP}})
\]

with \( \kappa = 1/\sqrt{1 + \pi s^2/8} \) (figure 41.10).
41.5: Implementing inference with Gaussian approximations

Demonstration

Figure 41.11 shows the result of fitting a Gaussian approximation at the optimum $w_{\text{MP}}$, and the results of using that Gaussian approximation and equation (41.30) to make predictions. Comparing these predictions with those of the Langevin Monte Carlo method (figure 41.7) we observe that, whilst qualitatively the same, the two are clearly numerically different. So at least one of the two methods is not completely accurate.

**Exercise 41.3.** Is the Gaussian approximation to $P(w | D, \alpha)$ too heavy-tailed or too light-tailed, or both? It may help to consider $P(w | D, \alpha)$ as a function of one parameter $w_i$ and to think of the two distributions on a logarithmic scale. Discuss the conditions under which the Gaussian approximation is most accurate.

**Why marginalize?**

If the output is immediately used to make a (0/1) decision and the costs associated with error are symmetrical, then the use of marginalized outputs under this Gaussian approximation will make no difference to the performance of the classifier, compared with using the outputs given by the most probable parameters, since both functions pass through 0.5 at $a_{\text{MP}} = 0$. But these Bayesian outputs will make a difference if, for example, there is an option of saying ‘I don’t know’, in addition to saying ‘I guess 0’ and ‘I guess 1’. And even if there are just the two choices ‘0’ and ‘1’, if the costs associated with error are unequal, then the decision boundary will be some contour other than the 0.5 contour, and the boundary will be affected by marginalization.
Postscript on Supervised Neural Networks

One of my students, Robert, asked:

Maybe I’m missing something fundamental, but supervised neural networks seem equivalent to fitting a pre-defined function to some given data, then extrapolating – what’s the difference?

I agree with Robert. The supervised neural networks we have studied so far are simply parameterized nonlinear functions which can be fitted to data. Hopefully you will agree with another comment that Robert made:

Unsupervised networks seem much more interesting than their supervised counterparts. I’m amazed that it works!
Hopfield Networks

We have now spent three chapters studying the single neuron. The time has come to connect multiple neurons together, making the output of one neuron be the input to another, so as to make neural networks.

Neural networks can be divided into two classes on the basis of their connectivity.

Feedforward networks. In a feedforward network, all the connections are directed such that the network forms a directed acyclic graph.

Feedback networks. Any network that is not a feedforward network will be called a feedback network.

In this chapter we will discuss a fully connected feedback network called the Hopfield network. The weights in the Hopfield network are constrained to be symmetric, i.e., the weight from neuron \( i \) to neuron \( j \) is equal to the weight from neuron \( j \) to neuron \( i \).

Hopfield networks have two applications. First, they can act as associative memories. Second, they can be used to solve optimization problems. We will first discuss the idea of associative memory, also known as content-addressable memory.

42.1 Hebbian learning

In Chapter 38, we discussed the contrast between traditional digital memories and biological memories. Perhaps the most striking difference is the associative nature of biological memory.

A simple model due to Donald Hebb (1949) captures the idea of associative memory. Imagine that the weights between neurons whose activities are positively correlated are increased:

\[
\frac{dw_{ij}}{dt} \sim \text{Correlation}(x_i, x_j). \tag{42.1}
\]

Now imagine that when stimulus \( m \) is present (for example, the smell of a banana), the activity of neuron \( m \) increases; and that neuron \( n \) is associated
with another stimulus, \( n \) (for example, the sight of a yellow object). If these two stimuli — a yellow sight and a banana smell — co-occur in the environment, then the Hebbian learning rule (42.1) will increase the weights \( w_{nm} \) and \( w_{mn} \). This means that when, on a later occasion, stimulus \( n \) occurs in isolation, making the activity \( x_n \) large, the positive weight from \( n \) to \( m \) will cause neuron \( m \) also to be activated. Thus the response to the sight of a yellow object is an automatic association with the smell of a banana. We could call this ‘pattern completion’. No teacher is required for this associative memory to work. No signal is needed to indicate that a correlation has been detected or that an association should be made. The unsupervised, local learning algorithm and the unsupervised, local activity rule spontaneously produce associative memory.

This idea seems so simple and so effective that it must be relevant to how memories work in the brain.

42.2 Definition of the binary Hopfield network

**Convention for weights.** Our convention in general will be that \( w_{ij} \) denotes the connection from neuron \( j \) to neuron \( i \).

**Architecture.** A Hopfield network consists of \( I \) neurons. They are fully connected through symmetric, bidirectional connections with weights \( w_{ij} = w_{ji} \). There are no self-connections, so \( w_{ii} = 0 \) for all \( i \). Biases \( w_{i0} \) may be included (these may be viewed as weights from a neuron ‘0’ whose activity is permanently \( x_0 = 1 \)). We will denote the activity of neuron \( i \) (its output) by \( x_i \).

**Activity rule.** Roughly, a Hopfield network’s activity rule is for each neuron to update its state as if it were a single neuron with the threshold activation function

\[
x(a) = \Theta(a) = \begin{cases} 
1 & a \geq 0 \\
-1 & a < 0.
\end{cases} \tag{42.2}
\]

Since there is feedback in a Hopfield network (every neuron’s output is an input to all the other neurons) we will have to specify an order for the updates to occur. The updates may be synchronous or asynchronous.

**Synchronous updates** — all neurons compute their activations

\[
a_i = \sum_j w_{ij} x_j \tag{42.3}
\]

then update their states simultaneously to

\[
x_i = \Theta(a_i). \tag{42.4}
\]

**Asynchronous updates** — one neuron at a time computes its activation and updates its state. The sequence of selected neurons may be a fixed sequence or a random sequence.

The properties of a Hopfield network may be sensitive to the above choices.

**Learning rule.** The learning rule is intended to make a set of desired memories \( \{x^{(n)}\} \) be stable states of the Hopfield network’s activity rule. Each memory is a binary pattern, with \( x_i \in \{-1, 1\} \).
42.3 Definition of the continuous Hopfield network

The weights are set using the sum of outer products or Hebb rule,

\[ w_{ij} = \eta \sum_n x_i^{(n)} x_j^{(n)}, \]  

(42.5)

where \( \eta \) is an unimportant constant. To prevent the largest possible weight from growing with \( N \) we might choose to set \( \eta = 1/N \).

Exercise 42.1. Explain why the value of \( \eta \) is not important for the Hopfield network defined above.

42.3 Definition of the continuous Hopfield network

Using the identical architecture and learning rule we can define a Hopfield network whose activities are real numbers between \(-1\) and \(1\).

Activity rule. A Hopfield network’s activity rule is for each neuron to update its state as if it were a single neuron with a sigmoid activation function. The updates may be synchronous or asynchronous, and involve the equations

\[ a_i = \sum_j w_{ij} x_j \]  

(42.6)

and

\[ x_i = \tanh(a_i). \]  

(42.7)

The learning rule is the same as in the binary Hopfield network, but the value of \( \eta \) becomes relevant. Alternatively, we may fix \( \eta \) and introduce a gain \( \beta \in (0, \infty) \) into the activation function:

\[ x_i = \tanh(\beta a_i). \]  

(42.8)

Exercise 42.2. Where have we encountered equations 42.6, 42.7, and 42.8 before?

42.4 Convergence of the Hopfield network

The hope is that the Hopfield networks we have defined will perform associative memory recall, as shown schematically in figure 42.2. We hope that the activity rule of a Hopfield network will take a partial memory or a corrupted memory, and perform pattern completion or error correction to restore the original memory.

But why should we expect any pattern to be stable under the activity rule, let alone the desired memories?

We address the continuous Hopfield network, since the binary network is a special case of it. We have already encountered the activity rule (42.6, 42.8)
when we discussed variational methods (section 33.2): when we approximated
the spin system whose energy function was
\[
E(x; J) = -\frac{1}{2} \sum_{m,n} J_{mn} x_m x_n - \sum_n h_n x_n
\]
(42.9)
with a separable distribution
\[
Q(x; a) = \frac{1}{Z_Q} \exp \left( \sum_n a_n x_n \right)
\]
(42.10)
and optimized the latter so as to minimize the variational free energy
\[
\beta \tilde{F}(a) = \beta \sum_x Q(x; a) E(x; J) - \sum_x Q(x; a) \ln \frac{1}{Q(x; a)},
\]
(42.11)
we found that the pair of iterative equations
\[
a_m = \beta \left( \sum_n J_{mn} \bar{x}_n + h_m \right)
\]
(42.12)
and
\[
\bar{x}_n = \tanh(a_n)
\]
(42.13)
were guaranteed to decrease the variational free energy
\[
\beta \tilde{F}(a) = \beta \left( -\frac{1}{2} \sum_{m,n} J_{mn} \bar{x}_m \bar{x}_n - \sum_n h_n \bar{x}_n \right) - \sum_n H_2^{(e)}(q_n).
\]
(42.14)
If we simply replace $J$ by $w$, $\bar{x}$ by $x$, and $h_n$ by $w_{0n}$, we see that the equations of the Hopfield network are identical to a set of mean field equations that minimize
\[
\beta \tilde{F}(x) = -\beta \frac{1}{2} x^T W x - \sum_i H_2^{(e)}((1 + x_i)/2).
\]
(42.15)
There is a general name for a function that decreases under the dynamical
evolution of a system and that is bounded below: such a function is a Lyapunov
function for the system. It is useful to be able to prove the existence of
Lyapunov functions: if a system has a Lyapunov function then its dynamics
are bound to settle down to a fixed point, which is a local minimum of the
Lyapunov function, or a limit cycle, along which the Lyapunov function is a
constant. Chaotic behaviour is not possible for a system with a Lyapunov
function. If a system has a Lyapunov function then its state space can be
divided into basins of attraction, one basin associated with each attractor.
So, the continuous Hopfield network’s activity rules (if implemented asyn-
chronously) have a Lyapunov function. This Lyapunov function is a convex
function of each parameter $a_i$ so a Hopfield network’s dynamics will always
converge to a stable fixed point.
This convergence proof depends crucially on the fact that the Hopfield
network’s connections are symmetric. It also depends on the updates being
made asynchronously.

Exercise 42.3. [2, p.520] Show by constructing an example that if a feedback
network does not have symmetric connections then its dynamics may
fail to converge to a fixed point.

Exercise 42.4. [2, p.521] Show by constructing an example that if a Hopfield
network is updated synchronously that, from some initial conditions, it
may fail to converge to a fixed point.
Convergence of the Hopfield network

42.4: Convergence of the Hopfield network

Figure 42.3. Binary Hopfield network storing four memories.

(a) The four memories, and the weight matrix. (b–h) Initial states that differ by one, two, three, four, or even five bits from a desired memory are restored to that memory in one or two iterations. (i–m) Some initial conditions that are far from the memories lead to stable states other than the four memories; in (i), the stable state looks like a mixture of two memories, ‘D’ and ‘J’; stable state (j) is like a mixture of ‘J’ and ‘C’; in (k), we find a corrupted version of the ‘M’ memory (two bits distant); in (l) a corrupted version of ‘J’ (four bits distant) and in (m), a state which looks spurious until we recognize that it is the inverse of the stable state (l).
42.5 The associative memory in action

Figure 42.3 shows the dynamics of a 25-unit binary Hopfield network that has learnt four patterns by Hebbian learning. The four patterns are displayed as five by five binary images in figure 42.3a. For twelve initial conditions, panels (b–m) show the state of the network, iteration by iteration, all 25 units being updated asynchronously in each iteration. For an initial condition randomly perturbed from a memory, it often only takes one iteration for all the errors to be corrected. The network has more stable states in addition to the four desired memories: the inverse of any stable state is also a stable state; and there are several stable states that can be interpreted as mixtures of the memories.

Brain damage

The network can be severely damaged and still work fine as an associative memory. If we take the 300 weights of the network shown in figure 42.3 and randomly set 50 or 100 of them to zero, we still find that the desired memories are attracting stable states. Imagine a digital computer that still works fine even when 20% of its components are destroyed!

Exercise 42.5. Implement a Hopfield network and confirm this amazing robust error-correcting capability.

More memories

We can squash more memories into the network too. Figure 42.4a shows a set of five memories. When we train the network with Hebbian learning, all five memories are stable states, even when 26 of the weights are randomly deleted (as shown by the ‘x’s in the figure). However, the basins of attraction are smaller than before: figure 42.4(b–f) shows the dynamics resulting from randomly chosen starting states close to each of the memories (3 bits flipped). Only three of the memories are recovered correctly.

If we try to store too many patterns, the associative memory fails catastrophically. When we add a sixth pattern, as shown in figure 42.5, only one of the patterns is stable; the others all flow into one of two spurious stable states.

42.6 The continuous-time continuous Hopfield network

The fact that the Hopfield network’s properties are not robust to the minor change from asynchronous to synchronous updates might be a cause for concern; can this model be a useful model of biological networks? It turns out that once we move to a continuous-time version of the Hopfield networks, this issue melts away.

We assume that each neuron’s activity \( x_i(t) \) is a continuous function of time \( x_i(t) \) and that the activations \( a_i(t) \) are computed instantaneously in accordance with

\[
 a_i(t) = \sum_j w_{ij} x_j(t),
\]

(42.16)

The neuron’s response to its activation is assumed to be mediated by the differential equation:

\[
 \frac{d}{dt} x_i(t) = -\frac{1}{\tau} (x_i(t) - f(a_i)),
\]

(42.17)
42.6: The continuous-time continuous Hopfield network

(a) Desired memories:

(b) Initial states that differ by three random bits from a memory: some are restored, but some converge to other states.

Figure 42.4. Hopfield network storing five memories, and suffering deletion of 26 of its 300 weights. (a) The five memories, and the weights of the network, with deleted weights shown by ‘x’.

(b–f) Initial states that differ by three random bits from a memory: some are restored, but some converge to other states.

Figure 42.5. An overloaded Hopfield network trained on six memories, most of which are not stable.
512

42 — Hopfield Networks

<table>
<thead>
<tr>
<th>Desired memories</th>
<th>Attracting stable states</th>
</tr>
</thead>
<tbody>
<tr>
<td>moscow-----------russia</td>
<td>moscow-----------russia</td>
</tr>
<tr>
<td>lima-------------peru</td>
<td>lima-------------peru</td>
</tr>
<tr>
<td>london----------england</td>
<td>london----------england</td>
</tr>
<tr>
<td>tokyo-----------japan</td>
<td>tokyo-----------japan</td>
</tr>
<tr>
<td>edinburgh---scotland</td>
<td>edinburgh---scotland</td>
</tr>
<tr>
<td>ottawa--------canada</td>
<td>oslo--------norway</td>
</tr>
<tr>
<td>oslo-----------norway</td>
<td>stockholm---sweden</td>
</tr>
<tr>
<td>stockholm---sweden</td>
<td>paris--------france</td>
</tr>
<tr>
<td>paris--------france</td>
<td>wzmhewn--xqwwpq</td>
</tr>
<tr>
<td>wzmhewn--xqwwpq</td>
<td>paris--------france</td>
</tr>
<tr>
<td>paris--------france</td>
<td>ecnarf-------sirap</td>
</tr>
</tbody>
</table>

Figure 42.6. Failure modes of a Hopfield network (highly schematic). A list of desired memories, and the resulting list of attracting stable states. Notice (1) some memories that are retained with a small number of errors; (2) desired memories that are completely lost (there is no attracting stable state at the desired memory or near it); (3) spurious stable states unrelated to the original list; (4) spurious stable states that are confabulations of desired memories.

where \( f(a) \) is the activation function, for example \( f(a) = \tanh(a) \). For a steady activation \( a_i \), the activity \( x_i(t) \) relaxes exponentially to \( f(a_i) \) with time-constant \( \tau \).

Now, here is the nice result: as long as the weight matrix is symmetric, this system has the variational free energy (42.15) as its Lyapunov function.

**Exercise 42.6.**\(^1\) By computing \( \frac{d}{dt} \tilde{F} \), prove that the variational free energy \( \tilde{F}(x) \) is a Lyapunov function for the continuous-time Hopfield network.

It is particularly easy to prove that a function \( L \) is a Lyapunov functions if the system’s dynamics perform steepest descent on \( L \), with \( \frac{d}{dt} x_i(t) \propto \frac{\partial}{\partial x_i} L \). In the case of the continuous-time continuous Hopfield network, it is not quite so simple, but every component of \( \frac{d}{dt} x_i(t) \) does have the same sign as \( \frac{\partial}{\partial x_i} \tilde{F} \), which means that with an appropriately defined metric, the Hopfield network dynamics do perform steepest descents on \( \tilde{F}(x) \).

### 42.7 The capacity of the Hopfield network

One way in which we viewed learning in the single neuron was as communication – communication of the labels of the training data set from one point in time to a later point in time. We found that the capacity of a linear threshold neuron was 2 bits per weight.

Similarly, we might view the Hopfield associative memory as a communication channel (figure 42.6). A list of desired memories is encoded into a set of weights \( W \) using the Hebb rule of equation (42.5), or perhaps some other learning rule. The receiver, receiving the weights \( W \) only, finds the stable states of the Hopfield network, which he interprets as the original memories. This communication system can fail in various ways, as illustrated in the figure.

1. Individual bits in some memories might be corrupted, that is, a stable state of the Hopfield network is displaced a little from the desired memory.

2. Entire memories might be absent from the list of attractors of the network; or a stable state might be present but have such a small basin of attraction that it is of no use for pattern completion and error correction.

3. Spurious additional memories unrelated to the desired memories might be present.

4. Spurious additional memories derived from the desired memories by operations such as mixing and inversion may also be present.
42.7: The capacity of the Hopfield network

Of these failure modes, modes 1 and 2 are clearly undesirable, mode 2 especially so. Mode 3 might not matter so much as long as each of the desired memories has a large basin of attraction. The fourth failure mode might in some contexts actually be viewed as beneficial. For example, if a network is required to memorize examples of valid sentences such as ‘John loves Mary’ and ‘John gets cake’, we might be happy to find that ‘John loves cake’ was also a stable state of the network. We might call this behaviour ‘generalization’.

The capacity of a Hopfield network with $I$ neurons might be defined to be the number of random patterns $N$ that can be stored without failure-mode 2 having substantial probability. If we also require failure-mode 1 to have tiny probability then the resulting capacity is much smaller. We now study these alternative definitions of the capacity.

The capacity of the Hopfield network – stringent definition

We will first explore the information storage capabilities of a binary Hopfield network that learns using the Hebb rule by considering the stability of just one bit of one of the desired patterns, assuming that the state of the network is set to that desired pattern $x^{(n)}$. We will assume that the patterns to be stored are randomly selected binary patterns.

The activation of a particular neuron $i$ is

$$a_i = \sum_j w_{ij} x_j^{(n)}, \quad (42.18)$$

where the weights are, for $i \neq j$,

$$w_{ij} = x_i^{(n)} x_j^{(n)} + \sum_{m \neq n} x_i^{(m)} x_j^{(m)}. \quad (42.19)$$

Here we have split $W$ into two terms, the first of which will contribute ‘signal’, reinforcing the desired memory, and the second ‘noise’. Substituting for $w_{ij}$, the activation is

$$a_i = \sum_{j \neq i} x_i^{(n)} x_j^{(n)} + \sum_{j \neq i \ m \neq n} x_i^{(m)} x_j^{(m)} x_j^{(n)} \quad (42.20)$$

$$= (I-1)x_i^{(n)} + \sum_{j \neq i \ m \neq n} x_i^{(m)} x_j^{(m)} x_j^{(n)}. \quad (42.21)$$

The first term is $(I-1)$ times the desired state $x_i^{(n)}$. If this were the only term, it would keep the neuron firmly clamped in the desired state. The second term is a sum of $(I-1)(N-1)$ random quantities $x_i^{(m)} x_j^{(m)} x_j^{(n)}$. A moment’s reflection confirms that these quantities are independent random binary variables with mean 0 and variance 1.

Thus, considering the statistics of $a_i$ under the ensemble of random patterns, we conclude that $a_i$ has mean $(I-1)x_i^{(n)}$ and variance $(I-1)(N-1)$.

For brevity, we will now assume $I$ and $N$ are large enough that we can neglect the distinction between $I$ and $I-1$, and between $N$ and $N-1$. Then we can restate our conclusion: $a_i$ is Gaussian-distributed with mean $Ix_i^{(n)}$ and variance $IN$.

What then is the probability that the selected bit is stable, if we put the network into the state $x^{(n)}$? The probability that bit $i$ will flip on the first iteration of the Hopfield network’s dynamics is

$$P(i \text{ unstable}) = \Phi \left( -\frac{I}{\sqrt{IN}} \right) = \Phi \left( -\frac{1}{\sqrt{N/I}} \right), \quad (42.22)$$

![Figure 42.7](image-url)
where
\[ \Phi(z) \equiv \int_{-\infty}^{\infty} dq \frac{1}{\sqrt{2\pi}} e^{-q^2/2}. \] (42.23)

The important quantity \( N/I \) is the ratio of the number of patterns stored to the number of neurons. If, for example, we try to store \( N \approx 0.18I \) patterns in the Hopfield network then there is a chance of 1% that a specified bit in a specified pattern will be unstable on the first iteration.

We are now in a position to derive our first capacity result, for the case where no corruption of the desired memories is permitted.

**Exercise 42.7.** Assume that we wish all the desired patterns to be completely stable – we don’t want any of the bits to flip when the network is put into any desired pattern state – and the total probability of any error at all is required to be less than a small number \( \epsilon \). Using the approximation to the error function for large \( z \),

\[ \Phi(-z) \approx \frac{1}{\sqrt{2\pi}} \frac{e^{-z^2/2}}{z}, \] (42.24)

show that the maximum number of patterns that can be stored, \( N_{\text{max}} \), is

\[ N_{\text{max}} \approx \frac{I}{4\ln I + 2\ln(1/\epsilon)}. \] (42.25)

If, however, we allow a small amount of corruption of memories to occur, the number of patterns that can be stored increases.

**The statistical physicists’ capacity**

The analysis that led to equation (42.22) tells us that if we try to store \( N \approx 0.18I \) patterns in the Hopfield network then, starting from a desired memory, about 1% of the bits will be unstable on the first iteration. Our analysis does not shed light on what is expected to happen on subsequent iterations. The flipping of these bits might make some of the other bits unstable too, causing an increasing number of bits to be flipped. This process might lead to an avalanche in which the network’s state ends up a long way from the desired memory.

In fact, when \( N/I \) is large, such avalanches do happen. When \( N/I \) is small, they tend not to – there is a stable state near to each desired memory. For the limit of large \( I \), Amit et al. (1985) have used methods from statistical physics to find numerically the transition between these two behaviours. There is a sharp discontinuity at

\[ N_{\text{crit}} = 0.138I. \] (42.26)

**Figure 42.8.** Overlap between a desired memory and the stable state nearest to it as a function of the loading fraction \( N/I \). The overlap is defined to be the scaled inner product \( \sum x_i x_i^{(n)}/I \), which is 1 when recall is perfect and zero when the stable state has 50% of the bits flipped. There is an abrupt transition at \( N/I = 0.138 \), where the overlap drops from 0.97 to zero.
Below this critical value, there is likely to be a stable state near every desired memory, in which a small fraction of the bits are flipped. When \( N/I \) exceeds 0.138, the system has only spurious stable states, known as spin glass states, none of which is correlated with any of the desired memories. Just below the critical value, the fraction of bits that are flipped when a desired memory has evolved to its associated stable state is 1.6%. Figure 42.8 shows the overlap between the desired memory and the nearest stable state as a function of \( N/I \).

Some other transitions in properties of the model occur at some additional values of \( N/I \), as summarized below.

For all \( N/I \), stable spin glass states exist, uncorrelated with the desired memories.

For \( N/I > 0.138 \), these spin glass states are the only stable states.

For \( N/I \in (0, 0.138) \), there are stable states close to the desired memories.

For \( N/I \in (0, 0.05) \), the stable states associated with the desired memories have lower energy than the spurious spin glass states.

For \( N/I \in (0.05, 0.138) \), the spin glass states dominate – there are spin glass states that have lower energy than the stable states associated with the desired memories.

For \( N/I \in (0, 0.03) \), there are additional mixture states, which are combinations of several desired memories. These stable states do not have as low energy as the stable states associated with the desired memories.

In conclusion, the capacity of the Hopfield network with \( I \) neurons, if we define the capacity in terms of the abrupt discontinuity discussed above, is 0.138\( I \) random binary patterns, each of length \( I \), each of which is received with 1.6% of its bits flipped. In bits, this capacity is

\[
0.138 I^2 \times (1 - H_2(0.016)) = 0.122 I^2 \text{ bits.} \quad (42.27)
\]

Since there are \( I^2/2 \) weights in the network, we can also express the capacity as 0.24 bits per weight.

42.8 Improving on the capacity of the Hebb rule

The capacities discussed in the previous section are the capacities of the Hopfield network whose weights are set using the Hebbian learning rule. We can do better than the Hebb rule by defining an objective function that measures how well the network stores all the memories, and minimizing it.

For an associative memory to be useful, it must be able to correct at least one flipped bit. Let’s make an objective function that measures whether flipped bits tend to be restored correctly. Our intention is that, for every neuron \( i \) in the network, the weights to that neuron should satisfy this rule:

for every pattern \( x^{(n)} \), if the neurons other than \( i \) are set correctly to \( x_j^{(n)} \), then the activation of neuron \( i \) should be such that its preferred output is \( x_i = x_i^{(n)} \).

Is this rule a familiar idea? Yes, it is precisely what we wanted the single neuron of Chapter 39 to do. Each pattern \( x^{(n)} \) defines an input, target pair for the single neuron \( i \). And it defines an input, target pair for all the other neurons too.
Algorithm 42.9. Octave source code for optimizing the weights of a Hopfield network, so that it works as an associative memory. c.f. algorithm 39.5. The data matrix $x$ has $I$ columns and $N$ rows. The matrix $t$ is identical to $x$ except that $-1$s are replaced by $0$s.

\[
\begin{align*}
    w &= x' * x ; \quad \# \text{initialize the weights using Hebb rule} \\
    \text{for } l = 1:L \# \text{loop } L \text{ times} \\
    \quad &\quad \text{for } i=1:I \# \text{ensure the self-weights are zero.} \\
    \quad &\quad \quad w(i,i) = 0 ; \\
    \quad &\quad \text{end} \# \\
    \quad \text{a} &= x * w ; \quad \# \text{compute all activations} \\
    \quad \text{y} &= \text{sigmoid}(a) ; \quad \# \text{compute all outputs} \\
    \quad \text{e} &= t - y ; \quad \# \text{compute all errors} \\
    \quad \text{gw} &= x' * e ; \quad \# \text{compute the gradients} \\
    \quad \text{gw} &= \text{gw} + \text{gw}' ; \quad \# \text{symmetrize gradients} \\
    \quad w &= w + \eta \cdot ( \text{gw} - \alpha \cdot w ) ; \quad \# \text{make step} \\
    \end{align*}
\]

So, just as we defined an objective function (39.11) for the training of a single neuron as a classifier, we can define

\[
G(W) = - \sum_i \sum_n t_i^{(n)} \ln y_i^{(n)} + (1 - t_i^{(n)}) \ln(1 - y_i^{(n)}) \quad (42.28)
\]

where

\[
t_i^{(n)} = \begin{cases} 
1 & x_i^{(n)} = 1 \\
0 & x_i^{(n)} = -1 
\end{cases}
\]

and

\[
y_i^{(n)} = \frac{1}{1 + \exp(-a_i^{(n)})}, \quad \text{where } a_i^{(n)} = \sum w_{ij} x_j^{(n)}. \quad (42.30)
\]

We can then steal the algorithm (algorithm 39.5, p.478) which we wrote for the single neuron, to write an algorithm for optimizing a Hopfield network, algorithm 42.9. The convenient syntax of Octave requires very few changes; the extra lines enforce the constraints that the self-weights $w_{ii}$ should all be zero and that the weight matrix should be symmetrical ($w_{ij} = w_{ji}$).

As expected, this learning algorithm does a better job than the one-shot Hebbian learning rule. When the six patterns of figure 42.5, which cannot be memorized by the Hebb rule, are learned using algorithm 42.9, all six patterns become stable states.

**Exercise 42.8.** Implement this learning rule and investigate empirically its capacity for memorizing random patterns; also compare its avalanche properties with those of the Hebb rule.

### 42.9 Hopfield networks for optimization problems

Since a Hopfield network’s dynamics minimize an energy function, it is natural to ask whether we can map interesting optimization problems onto Hopfield networks. Biological data processing problems often involve an element of *constraint satisfaction* – in scene interpretation, for example, one might wish to infer the spatial location, orientation, brightness and texture of each visible element, and which visible elements are connected together in objects. These inferences are constrained by the given data and by prior knowledge about continuity of objects.
Hopfield and Tank (1985) suggested that one might take an interesting constraint satisfaction problem and design the weights of a binary or continuous Hopfield network such that the settling process of the network would minimize the objective function of the problem.

The travelling salesman problem

A classic constraint satisfaction problem to which Hopfield networks have been applied is the travelling salesman problem.

A set of $K$ cities is given, and a matrix of the $K(K-1)/2$ distances between those cities. The task is to find a closed tour of the cities, visiting each city once, that has the smallest total distance. The travelling salesman problem is equivalent in difficulty to an NP-complete problem.

The method suggested by Hopfield and Tank is to represent a tentative solution to the problem by the state of a network with $I = K^2$ neurons arranged in a square, with each neuron representing the hypothesis that a particular city comes at a particular point in the tour. It will be convenient to consider the states of the neurons as being between 0 and 1 rather than $-1$ and 1. Two solution states for a four-city travelling salesman problem are shown in figure 42.10a.

The weights in the Hopfield network play two roles. First, they must define an energy function which is minimized only when the state of the network represents a valid tour. A valid state is one that looks like a permutation matrix, having exactly one ‘1’ in every row and one ‘1’ in every column. This rule can be enforced by putting large negative weights between any pair of neurons that are in the same row or the same column, and setting a positive bias for all neurons to ensure that $K$ neurons do turn on. Figure 42.10b shows the negative weights that are connected to one neuron, ‘B2’, which represents the statement ‘city B comes second in the tour’.

Second, the weights must encode the objective function that we want to minimize – the total distance. This can be done by putting negative weights proportional to the appropriate distances between the nodes in adjacent columns. For example, between the $B$ and $D$ nodes in adjacent columns, the weight would be $-d_{BD}$. The negative weights that are connected to neuron $B2$ are shown in figure 42.10c. The result is that when the network is in a valid state, its total energy will be the total distance of the corresponding tour.
tour, plus a constant given by the energy associated with the biases.

Now, since a Hopfield network minimizes its energy, it is hoped that the binary or continuous Hopfield network’s dynamics will take the state to a minimum that is a valid tour and which might be an optimal tour. This hope is not fulfilled for large travelling salesman problems, however, without some careful modifications. We have not specified the size of the weights that enforce the tour’s validity, relative to the size of the distance weights, and setting this scale factor poses difficulties. If ‘large’ validity-enforcing weights are used, the network’s dynamics will rattle into a valid state with little regard for the distances. If ‘small’ validity-enforcing weights are used, it is possible that the distance weights will cause the network to adopt an invalid state that has lower energy than any valid state. Our original formulation of the energy function puts the objective function and the solution’s validity in potential conflict with each other. This difficulty has been resolved by the work of Sree Aiyer (1991), who showed how to modify the distance weights so that they would not interfere with the solution’s validity, and how to define a continuous Hopfield network whose dynamics are at all times confined to a ‘valid subspace’. Aiyer used a graduated non-convexity or deterministic annealing approach to find good solutions using these Hopfield networks. The deterministic annealing approach involves gradually increasing the gain $\beta$ of the neurons in the network from 0 to $\infty$, at which point the state of the network corresponds to a valid tour. A sequence of trajectories generated by applying this method to a thirty-city travelling salesman problem is shown in figure 42.11a.

A solution to the ‘travelling scholar problem’ found by Aiyer using a continuous Hopfield network is shown in figure 42.11b.
42.10 Further exercises

Exercise 42.9. Storing two memories.

Two binary memories \( m \) and \( n \) \((m_i, n_i \in \{-1, +1\})\) are stored by Hebbian learning in a Hopfield network using

\[
    w_{ij} = \begin{cases} 
    m_i m_j + n_i n_j & \text{for } i \neq j \\
    0 & \text{for } i = j.
    \end{cases} \tag{42.31}
\]

The biases \( b_i \) are set to zero.

The network is put in the state \( x = m \). Evaluate the activation \( a_i \) of neuron \( i \) and show that in can be written in the form

\[
    a_i = \mu m_i + v n_i. \tag{42.32}
\]

By comparing the signal strength, \( \mu \), with the magnitude of the noise strength, \( |v| \), show that \( x = m \) is a stable state of the dynamics of the network.

The network is put in a state \( x \) differing in \( D \) places from \( m \),

\[
    x = m + 2d, \tag{42.33}
\]

where the perturbation \( d \) satisfies \( d_i \in \{-1, 0, +1\} \). \( D \) is the number of components of \( d \) that are non-zero, and for each \( d_i \) that is non-zero, \( d_i = -m_i \). Defining the overlap between \( m \) and \( n \) to be

\[
    o_{mn} = \sum_{i=1}^{I} m_i n_i, \tag{42.34}
\]

evaluate the activation \( a_i \) of neuron \( i \) again and show that the dynamics of the network will restore \( x \) to \( m \) if the number of flipped bits satisfies

\[
    D < \frac{1}{4} (I - |o_{mn}| - 2). \tag{42.35}
\]

How does this number compare with the maximum number of flipped bits that can be corrected by the optimal decoder, assuming the vector \( x \) is either a noisy version of \( m \) or of \( n \)?

Exercise 42.10. Hopfield network as a collection of binary classifiers. This exercise explores the link between unsupervised networks and supervised networks. If a Hopfield network’s desired memories are all attracting stable states, then every neuron in the network has weights going to it that solve a classification problem personal to that neuron. Take the set of memories and write them in the form \( x_i^{(n)} \), \( x_i^{(m)} \), where \( x' \) denotes all the components \( x_i' \) for all \( i' \neq i \), and let \( w' \) denote the vector of weights \( w_{i'j} \), for \( i' \neq i \).

Using what we know about the capacity of the single neuron, show that it is almost certainly impossible to store more than \( 2I \) random memories in a Hopfield network of \( I \) neurons.
Lyapunov functions

Exercise 42.11.\[9\] Erik’s puzzle. In a stripped-down version of Conway’s game of life, cells are arranged on a square grid. Each cell is either alive or dead. Live cells do not die. Dead cells become alive if two or more of their immediate neighbours are alive. (Neighbours to north, south, east and west.) What is the smallest number of live cells needed in order that these rules lead to an entire $N \times N$ square being alive?

In a $d$-dimensional version of the same game, the rule is that if $d$ neighbours are alive then you come to life. What is the smallest number of live cells needed in order that an entire $N \times N \times \cdots \times N$ hypercube becomes alive? (And how should those live cells be arranged?)

The southeast puzzle

The southeast puzzle is played on a semi-infinite chess board, starting at its northwest (top left) corner. There are three rules:

1. In the starting position, one piece is placed in the northwest-most square (figure 42.13a).

2. It is not permitted for more than one piece to be on any given square.

3. At each step, you remove one piece from the board, and replace it with two pieces, one in the square immediately to the east, and one in the the square immediately to the south, as illustrated in figure 42.13b. Every such step increases the number of pieces on the board by one.

After move (b) has been made, either piece may be selected for the next move. Figure 42.13c shows the outcome of moving the lower piece. At the next move, either the lowest piece or the middle piece of the three may be selected; the uppermost piece may not be selected, since that would violate rule 2. At move (d) we have selected the middle piece. Now any of the pieces may be moved, except for the leftmost piece.

Now, here is the puzzle:

Exercise 42.12.\[4, p.521\] Is it possible to obtain a position in which all the ten squares closest to the northwest corner, marked in figure 42.13z, are empty?

[Hint: this puzzle has a connection to data compression.]

42.11 Solutions

Solution to exercise 42.3 (p.508). Take a binary feedback network with 2 neurons and let $w_{12} = 1$ and $w_{21} = -1$. Then whenever neuron 1 is updated, it will match neuron 2, and whenever neuron 2 is updated, it will flip to the opposite state from neuron 1. There is no stable state.
Solution to exercise 42.4 (p.508).  Take a binary Hopfield network with 2 neurons and let \( w_{12} = w_{21} = 1 \), and let the initial condition be \( x_1 = 1, x_2 = -1 \). Then if the dynamics are synchronous, on every iteration both neurons will flip their state. The dynamics do not converge to a fixed point.

Solution to exercise 42.12 (p.520). The key to this problem is to notice its similarity to the construction of a binary symbol code. Starting from the empty string, we can build a binary tree by repeatedly splitting a codeword into two. Every codeword has an implicit probability \( 2^{-l} \), where \( l \) is the depth of the codeword in the binary tree. Whenever we split a codeword in two and create two new codewords whose length is increased by one, the two new codewords each have implicit probability equal to half that of the old codeword. For a complete binary code, the Kraft equality affirms that the sum of these implicit probabilities is 1.

Similarly, in \texttt{southeast}, we can associate a ‘weight’ with each piece on the board. If we assign a weight of 1 to any piece sitting on the top left square; a weight of 1/2 to any piece on a square whose distance from the top left is one; a weight of 1/4 to any piece whose distance from the top left is two; and so forth, with ‘distance’ being the city-block distance; then every legal move in \texttt{southeast} leaves unchanged the total weight of all pieces on the board. Lyapunov functions come in two flavours: the function may be a function of state whose value is known to stay constant; or it may be a function of state that is bounded below, and whose value always decreases or stays constant. The total weight is a Lyapunov function of the second type.

The starting weight is 1, so now we have a powerful tool: a conserved function of the state. Is it possible to find a position in which the ten highest-weight squares are vacant, and the total weight is 1? What is the total weight if all the other squares on the board are occupied (figure 42.14)? The total weight would be \( \sum_{l=4}^{\infty} (l+1)2^{-l} \). Which is equal to 3/4. So it is impossible to empty all ten of those squares.

Figure 42.14. A possible position for the \texttt{southeast} puzzle?
Boltzmann Machines

43.1 From Hopfield networks to Boltzmann machines

We have noticed that the binary Hopfield network minimizes an energy function

\[ E(x) = -\frac{1}{2} x^T W x \]  

(43.1)

and that the continuous Hopfield network with activation function \( x_n = \tanh(a_n) \) can be viewed as approximating the probability distribution associated with that energy function,

\[ P(x|W) = \frac{1}{Z(W)} \exp[-E(x)] = \frac{1}{Z(W)} \exp \left[ \frac{1}{2} x^T W x \right]. \]  

(43.2)

These observations motivate the idea of working with a neural network model that actually implements the above probability distribution.

The stochastic Hopfield network or Boltzmann machine (Hinton and Sejnowski, 1986) has the following activity rule:

<table>
<thead>
<tr>
<th>Activity rule of Boltzmann machine: after computing the activation ( a_n ),</th>
</tr>
</thead>
<tbody>
<tr>
<td>set ( x_n = +1 ) with probability ( \frac{1}{1 + e^{-2a}} )</td>
</tr>
<tr>
<td>else set ( x_n = -1 ).</td>
</tr>
</tbody>
</table>

(43.3)

This rule implements Gibbs sampling for the probability distribution (43.2).

Boltzmann machine learning

Given a set of examples \( \{x^{(n)}\}_1^N \) from the real world, we might be interested in adjusting the weights \( W \) such that the generative model

\[ P(x|W) = \frac{1}{Z(W)} \exp \left[ \frac{1}{2} x^T W x \right] \]  

(43.4)

is well matched to those examples. We can derive a learning algorithm by writing down Bayes’ theorem to obtain the posterior probability of the weights given the data:

\[ P(W|\{x^{(n)}\}_1^N) = \frac{\prod_{n=1}^N P(x^{(n)}|W) P(W)}{P(\{x^{(n)}\}_1^N)}. \]  

(43.5)

We concentrate on the first term in the numerator, the likelihood, and derive a maximum likelihood algorithm (though there might be advantages in pursuing
43.1: From Hopfield networks to Boltzmann machines

a full Bayesian approach as we did in the case of the single neuron). We differentiate the logarithm of the likelihood,

\[ \ln \left[ \prod_{n=1}^{N} P(x^{(n)}|W) \right] = \sum_{n=1}^{N} \left[ \frac{1}{2} x^{(n)}^T W x^{(n)} - \ln Z(W) \right], \tag{43.6} \]

with respect to \( w_{ij} \), bearing in mind that \( W \) is defined to be symmetric with \( w_{ji} = w_{ij} \).

**Exercise 43.1.** Show that the derivative of \( \ln Z(W) \) with respect to \( w_{ij} \) is

\[ \frac{\partial}{\partial w_{ij}} \ln Z(W) = \sum_{x} x_i x_j P(x|W) = \langle x_i x_j \rangle_{P(x|W)}. \tag{43.7} \]

[This exercise is similar to exercise 22.12 (p.307).]

The derivative of the log likelihood is therefore:

\[ \frac{\partial}{\partial w_{ij}} \ln P(\{x^{(n)}\}_1^N|W) = \sum_{n=1}^{N} \left[ x_{i}^{(n)} x_{j}^{(n)} - \langle x_{i} x_{j} \rangle_{P(x|W)} \right] \tag{43.8} \]

\[ = N \left[ \langle x_{i} x_{j} \rangle_{\text{Data}} - \langle x_{i} x_{j} \rangle_{P(x|W)} \right]. \tag{43.9} \]

This gradient is proportional to the difference of two terms. The first term is the *empirical* correlation between \( x_i \) and \( x_j \),

\[ \langle x_{i} x_{j} \rangle_{\text{Data}} = \frac{1}{N} \sum_{n=1}^{N} \left[ x_{i}^{(n)} x_{j}^{(n)} \right], \tag{43.10} \]

and the second term is the correlation between \( x_i \) and \( x_j \) under the current model,

\[ \langle x_{i} x_{j} \rangle_{P(x|W)} = \sum_{x} x_{i} x_{j} P(x|W). \tag{43.11} \]

The first correlation \( \langle x_{i} x_{j} \rangle_{\text{Data}} \) is readily evaluated – it is just the empirical correlation between the activities in the real world. The second correlation, \( \langle x_{i} x_{j} \rangle_{P(x|W)} \), is not so easy to evaluate, but it can be estimated by Monte Carlo methods, that is, by observing the average value of \( x_{i} x_{j} \) while the activity rule of the Boltzmann machine, equation (43.3), is iterated.

In the special case \( W = 0 \), we can evaluate the gradient exactly because, by symmetry, the correlation \( \langle x_{i} x_{j} \rangle_{P(x|W)} \) must be zero. If the weights are adjusted by gradient descent with learning rate \( \eta \), then, after one iteration, the weights will be

\[ w_{ij} = \eta \sum_{n=1}^{N} \left[ x_{i}^{(n)} x_{j}^{(n)} \right], \tag{43.12} \]

precisely the value of the weights given by the Hebb rule, equation (16.5), with which we trained the Hopfield network.

**Interpretation of Boltzmann machine learning**

One way of viewing the two terms in the gradient (43.9) is as ‘waking’ and ‘sleeping’ rules. While the network is ‘awake’, it measures the correlation between \( x_i \) and \( x_j \) in the real world, and weights are *increased* in proportion. While the network is ‘asleep’, it ‘dreams’ about the world using the generative model (43.4), and measures the correlations between \( x_i \) and \( x_j \) in the model world; these correlations determine a proportional *decrease* in the weights. If the second-order correlations in the dream world match the correlations in the real world, then the two terms balance and the weights do not change.

---

**Figure 43.1.** The ‘shifter’ ensembles. (a) Four samples from the plain shifter ensemble. (b) Four corresponding samples from the labelled shifter ensemble.
Criticism of Hopfield networks and simple Boltzmann machines

Up to this point we have discussed Hopfield networks and Boltzmann machines in which all of the neurons correspond to visible variables \(x_i\). The result is a probabilistic model that, when optimized, can capture the second-order statistics of the environment. [The second-order statistics of an ensemble \(P(x)\) are the expected values \((x_i x_j)\) of all the pairwise products \(x_i x_j\).] The real world, however, often has higher-order correlations that must be included if our description of it is to be effective. Often the second-order correlations in themselves may carry little or no useful information.

Consider, for example, the ensemble of binary images of chairs. We can imagine images of chairs with various designs – four-legged chairs, comfy chairs, chairs with five legs and wheels, wooden chairs, cushioned chairs, chairs with rockers instead of legs. A child can easily learn to distinguish these images from images of carrots and parrots. But I expect the second-order statistics of the raw data are useless for describing the ensemble. Second-order statistics only capture whether two pixels are likely to be in the same state as each other. Higher-order concepts are needed to make a good generative model of images of chairs.

A simpler ensemble of images in which high-order statistics are important is the ‘shifter ensemble’, which comes in two flavours. Figure 43.1a shows a few samples from the ‘plain shifter ensemble’. In each image, the bottom eight pixels are a copy of the top eight pixels, either shifted one pixel to the left, or unshifted, or shifted one pixel to the right. (The top eight pixels are set at random.) This ensemble is a simple model of the visual signals from the two eyes arriving at early levels of the brain. The signals from the two eyes are similar to each other but may differ by small translations because of the varying depth of the visual world. This ensemble is simple to describe, but its second-order statistics convey no useful information. The correlation between one pixel and any of the three pixels above it is \(1/3\). The correlation between any other two pixels is zero.

Figure 43.1b shows a few samples from the ‘labelled shifter ensemble’. Here, the problem has been made easier by including an extra three neurons that label the visual image as being an instance of either the ‘shift left’, ‘no shift’, or ‘shift right’ sub-ensemble. But with this extra information, the ensemble is still not learnable using second-order statistics alone. The second-order correlation between any label neuron and any image neuron is zero. We need models that can capture higher-order statistics of an environment.

So, how can we develop such models? One idea might be to create models that directly capture higher-order correlations, such as:

\[
P'(x|W, V, \ldots) = \frac{1}{Z'} \exp \left( \frac{1}{2} \sum_{ij} w_{ij} x_i x_j + \frac{1}{6} \sum_{ijk} v_{ijk} x_i x_j x_k + \cdots \right). \tag{43.13}
\]

Such higher-order Boltzmann machines are equally easy to simulate using stochastic updates, and the learning rule for the higher-order parameters \(v_{ijk}\) is equivalent to the learning rule for \(w_{ij}\).

Exercise 43.2.[2] Derive the gradient of the log likelihood with respect to \(v_{ijk}\).

It is possible that the spines found on biological neurons are responsible for detecting correlations between small numbers of incoming signals. However, to capture statistics of high enough order to describe the ensemble of images of chairs well would require an unimaginable number of terms. To capture
merely the fourth-order statistics in a 128 \times 128 pixel image, we need more than 10^7 parameters.

So measuring moments of images is not a good way to describe their underlying structure. Perhaps what we need instead or in addition are hidden variables, also known to statisticians as latent variables. This is the important innovation introduced by Hinton and Sejnowski (1986). The idea is that the high-order correlations among the visible variables are described by including extra hidden variables and sticking to a model that has only second-order interactions between its variables; the hidden variables induce higher-order correlations between the visible variables.

43.2 Boltzmann machine with hidden units

We now add hidden neurons to our stochastic model. These are neurons that do not correspond to observed variables; they are free to play any role in the probabilistic model defined by equation (43.4). They might actually take on interpretable roles, effectively performing `feature extraction'.

Learning in Boltzmann machines with hidden units

The activity rule of a Boltzmann machine with hidden units is identical to that of the original Boltzmann machine. The learning rule can again be derived by maximum likelihood, but now we need to take into account the fact that the states of the hidden units are unknown. We will denote the states of the visible units by \( x \), the states of the hidden units by \( h \), and the generic state of a neuron (either visible or hidden) by \( y \), with \( y \equiv (x, h) \). The state of the network when the visible neurons are clamped in state \( x^{(n)} \) is \( y^{(n)} \equiv (x^{(n)}, h) \).

The likelihood of \( \mathbf{W} \) given a single data example \( x^{(n)} \) is

\[
P(x^{(n)} | \mathbf{W}) = \sum_h P(x^{(n)}, h | \mathbf{W}) = \sum_h \frac{1}{Z(\mathbf{W})} \exp \left[ \frac{1}{2} y^{(n)} \mathbf{W} y^{(n)} \right],
\]

where

\[
Z(\mathbf{W}) = \sum_{x,h} \exp \left[ \frac{1}{2} y^T \mathbf{W} y \right].
\]

Equation (43.14) may also be written

\[
P(x^{(n)} | \mathbf{W}) = \frac{Z_{x^{(n)}}(\mathbf{W})}{Z(\mathbf{W})}
\]

where

\[
Z_{x^{(n)}}(\mathbf{W}) = \sum_h \exp \left[ \frac{1}{2} y^{(n)} \mathbf{W} y^{(n)} \right].
\]

Differentiating the likelihood as before, we find that the derivative with respect to any weight \( w_{ij} \) is again the difference between a `waking' term and a `sleeping' term,

\[
\frac{\partial}{\partial w_{ij}} \ln P((x^{(n)})_i | \mathbf{W}) = \sum_n \left\{ \langle y_i y_j \rangle_{P(h|x^{(n)}, \mathbf{W})} - \langle y_i y_j \rangle_{P(x,h|\mathbf{W})} \right\}.
\]

The first term \( \langle y_i y_j \rangle_{P(h|x^{(n)}, \mathbf{W})} \) is the correlation between \( y_i \) and \( y_j \) if the Boltzmann machine is simulated with the visible variables clamped to \( x^{(n)} \) and the hidden variables freely sampling from their conditional distribution.

The second term \( \langle y_i y_j \rangle_{P(x,h|\mathbf{W})} \) is the correlation between \( y_i \) and \( y_j \) when the Boltzmann machine generates samples from its model distribution.
Hinton and Sejnowski demonstrated that non-trivial ensembles such as the labelled shifter ensemble can be learned using a Boltzmann machine with hidden units. The hidden units take on the role of feature detectors that spot patterns likely to be associated with one of the three shifts.

The Boltzmann machine is time-consuming to simulate because the computation of the gradient of the log likelihood depends on taking the difference of two gradients, both found by Monte Carlo methods. So Boltzmann machines are not in widespread use. It is an area of active research to create models that embody the same capabilities using more efficient computations (Hinton et al., 1995; Dayan et al., 1995; Hinton and Ghahramani, 1997; Hinton, 2000; Hinton and Teh, 2001).

43.3 Exercise

Exercise 43.3. Can the ‘bars and stripes’ ensemble (figure 43.2) be learned by a Boltzmann machine with no hidden units? [You may be surprised!]

Figure 43.2. Four samples from the ‘bars and stripes’ ensemble. Each sample is generated by first picking an orientation, horizontal or vertical; then, for each row of spins in that orientation (each bar or stripe respectively), switching all spins on with probability $1/2$. 
44

Supervised Learning in Multilayer Networks

44.1 Multilayer perceptrons

No course on neural networks could be complete without a discussion of supervised multilayer networks, also known as backpropagation networks.

The multilayer perceptron is a feedforward network. It has input neurons, hidden neurons and output neurons. The hidden neurons may be arranged in a sequence of layers. The most common multilayer perceptrons have a single hidden layer, and are known as ‘two-layer’ networks, the number ‘two’ counting the number of layers of neurons not including the inputs.

Such a feedforward network defines a nonlinear parameterized mapping from an input \( x \) to an output \( y = y(x; w, A) \). The output is a continuous function of the input and of the parameters \( w \); the architecture of the net, i.e., the functional form of the mapping, is denoted by \( A \). Feedforward networks can be ‘trained’ to perform regression and classification tasks.

Regression networks

In the case of a regression problem, the mapping for a network with one hidden layer may have the form:

\[
\begin{align*}
\text{Hidden layer:} & \quad a_j^{(1)} = \sum_l w_{jl}^{(1)} x_l + \theta_j^{(1)}; \quad h_j = f^{(1)}(a_j^{(1)}) \\
\text{Output layer:} & \quad a_i^{(2)} = \sum_j w_{ij}^{(2)} h_j + \theta_i^{(2)}; \quad y_i = f^{(2)}(a_i^{(2)})
\end{align*}
\] (44.1) (44.2)

where, for example, \( f^{(1)}(a) = \tanh(a) \), and \( f^{(2)}(a) = a \). Here \( l \) runs over the inputs \( x_1, \ldots, x_L \), \( j \) runs over the hidden units, and \( i \) runs over the outputs. The ‘weights’ \( w \) and ‘biases’ \( \theta \) together make up the parameter vector \( w \). The nonlinear sigmoid function \( f^{(1)} \) at the hidden layer gives the neural network greater computational flexibility than a standard linear regression model. Graphically, we can represent the neural network as a set of layers of connected neurons (figure 44.1).

What sorts of functions can these networks implement?

Just as we explored the weight space of the single neuron in Chapter 39, examining the functions it could produce, let us explore the weight space of a multilayer network. In figure 44.2 I take a network with one input and one output and a large number \( H \) of hidden units, set the biases and weights \( \theta_j^{(1)}, \ldots, \theta_j^{(H)} \), \( w_{ij}^{(H+1)} \) and \( w_{ij}^{(2)} \), and choose \( f^{(1)} \) and \( f^{(2)} \) at random. The result is a random function. In figure 44.3 I take a network with one input and one output and a large number \( H \) of hidden units, set the biases and weights \( \theta_j^{(1)} \), \( \theta_j^{(H)} \), \( \theta_j^{(H+1)} \) and \( \theta_j^{(2)} \), and choose \( f^{(1)} \) and \( f^{(2)} \) at random. The result is a random function.
44.2 How a regression network is traditionally trained

This network is trained using a data set $D = \{x^{(n)}, t^{(n)}\}$ by adjusting $w$ so as to minimize an error function, e.g.,

$$E_D(w) = \frac{1}{2} \sum_n \sum_i \left( t_i^{(n)} - y_i(x^{(n)}; w) \right)^2. \quad (44.3)$$

This objective function is a sum of terms, one for each input/target pair $\{x, t\}$, measuring how close the output $y(x; w)$ is to the target $t$.

This minimization is based on repeated evaluation of the gradient of $E_D$. This gradient can be efficiently computed using the backpropagation algorithm (Rumelhart et al., 1986), which uses the chain rule to find the derivatives.
Often, regularization (also known as weight decay) is included, modifying the objective function to:

\[ M(w) = \beta E_D + \alpha E_W \]  

(44.4)

where, for example, \( E_W = \frac{1}{2} \sum w_i^2 \). This additional term favours small values of \( w \) and decreases the tendency of a model to overfit noise in the training data.

Rumelhart et al. (1986) showed that multilayer perceptrons can be trained, by gradient descent on \( M(w) \), to discover solutions to non-trivial problems such as deciding whether an image is symmetric or not. These networks have been successfully applied to real-world tasks as varied as pronouncing English textreading aloud (Sejnowski and Rosenberg, 1987) and focussing multiple-mirror telescopes (Angel et al., 1990).

### 44.3 Neural network learning as inference

The neural network learning process above can be given the following probabilistic interpretation. [Here we repeat and generalize the discussion of Chapter 41.]

The error function is interpreted as defining a noise model. \( \beta E_D \) is the negative log likelihood:

\[ P(D \mid w, \beta, \mathcal{H}) = \frac{1}{Z_D(\beta)} \exp(-\beta E_D). \]  

(44.5)

Thus, the use of the sum-squared error \( E_D \) (44.3) corresponds to an assumption of Gaussian noise on the target variables, and the parameter \( \beta \) defines a noise level \( \sigma_D^2 = 1/\beta \).

Similarly the regularizer is interpreted in terms of a log prior probability distribution over the parameters:

\[ P(w \mid \alpha, \mathcal{H}) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W). \]  

(44.6)

If \( E_W \) is quadratic as defined above, then the corresponding prior distribution is a Gaussian with variance \( \sigma_W^2 = 1/\alpha \). The probabilistic model \( \mathcal{H} \) specifies the architecture \( \mathcal{A} \) of the network, the likelihood (44.5), and the prior (44.6).

The objective function \( M(w) \) then corresponds to the inference of the parameters \( w \), given the data:

\[ P(w \mid D, \alpha, \beta, \mathcal{H}) = \frac{P(D \mid w, \beta, \mathcal{H})P(w \mid \alpha, \mathcal{H})}{P(D \mid \alpha, \beta, \mathcal{H})} \]  

(44.7)

\[ = \frac{1}{Z_M} \exp(-M(w)). \]  

(44.8)

The \( w \) found by (locally) minimizing \( M(w) \) is then interpreted as the (locally) most probable parameter vector, \( w_{\text{MP}} \).

The interpretation of \( M(w) \) as a log probability adds little new at this stage. But new tools will emerge when we proceed to other inferences. First, though, let us establish the probabilistic interpretation of classification networks, to which the same tools apply.
Binary classification networks

If the targets \( t \) in a data set are binary classification labels (0, 1), it is natural to use a neural network whose output \( y(x; w, A) \) is bounded between 0 and 1, and is interpreted as a probability \( P(t=1|x; w, A) \). For example, a network with one hidden layer could be described by the feedforward equations (44.1) and (44.2), with \( f^2(a) = 1/(1+e^{-a}) \). The error function \( \beta E_D \) is replaced by the negative log likelihood:

\[
G(w) = - \left[ \sum_n t^{(n)} \ln y(x^{(n)}; w) + (1 - t^{(n)}) \ln(1 - y(x^{(n)}; w)) \right].
\]  

(44.9)

The total objective function is then \( M = G + \alpha E_W \). Note that this includes no parameter \( \beta \) (because there is no Gaussian noise).

Multi-class classification networks

For a multi-class classification problem, we can represent the targets by a vector, \( t \), in which a single element is set to 1, indicating the correct class, and all other elements are set to 0. In this case it is appropriate to use a ‘softmax’ network having coupled outputs which sum to one and are interpreted as class probabilities \( y_i = P(t_i=1|x, w, A) \). The last part of equation (44.2) is replaced by:

\[
y_i = \frac{e^{a_i}}{\sum_p e^{a_p}}.
\]  

(44.10)

The negative log likelihood in this case is

\[
G(w) = - \sum_n \sum_i t^{(n)}_i \ln y_i(x^{(n)}; w).
\]  

(44.11)

As in the case of the regression network, the minimization of the objective function \( M(w) = G + \alpha E_W \) corresponds to an inference of the form (44.8). A variety of useful results can be built on this interpretation.

44.4 Benefits of the Bayesian approach to supervised feedforward neural networks

From the statistical perspective, supervised neural networks are nothing more than nonlinear curve-fitting devices. Curve fitting is not a trivial task however. The effective complexity of an interpolating model is of crucial importance, as illustrated in figure 44.5. Consider a control parameter that influences the complexity of a model, for example a regularization constant \( \alpha \) (weight decay parameter). As the control parameter is varied to increase the complexity of the model (descending from figure 44.5a–c and going from left to right across figure 44.5d), the best fit to the training data that the model can achieve becomes increasingly good. However, the empirical performance of the model, the test error, first decreases then increases again. An over-complex model overfits the data and generalizes poorly. This problem may also complicate the choice of architecture in a multilayer perceptron, the radius of the basis functions in a radial basis function network, and the choice of the input variables themselves in any multidimensional regression problem. Finding values for model control parameters that are appropriate for the data is therefore an important and non-trivial problem.

The overfitting problem can be solved by using a Bayesian approach to control model complexity.
44.4: Benefits of the Bayesian approach to supervised feedforward neural networks

If we give a probabilistic interpretation to the model, then we can evaluate the evidence for alternative values of the control parameters. As was explained in Chapter 28, over-complex models turn out to be less probable, and the evidence $P(\text{Data} \mid \text{Control Parameters})$ can be used as an objective function for optimization of model control parameters (figure 44.5e). The setting of $\alpha$ that maximizes the evidence is displayed in figure 44.5b.

Bayesian optimization of model control parameters has four important advantages. (1) No ‘test set’ or ‘validation set’ is involved, so all available training data can be devoted to both model fitting and model comparison. (2) Regularization constants can be optimized on-line, i.e., simultaneously with the optimization of ordinary model parameters. (3) The Bayesian objective function is not noisy, in contrast to a cross-validation measure. (4) The gradient of the evidence with respect to the control parameters can be evaluated, making it possible to simultaneously optimize a large number of control parameters.

Probabilistic modelling also handles uncertainty in a natural manner. It offers a unique prescription, marginalization, for incorporating uncertainty about parameters into predictions; this procedure yields better predictions, as we saw in Chapter 41. Figure 44.6 shows error bars on the predictions of a trained neural network.

**Implementation of Bayesian inference**

As was mentioned in Chapter 41, Bayesian inference for multilayer networks may be implemented by Monte Carlo sampling, or by deterministic methods employing Gaussian approximations (Neal, 1996; MacKay, 1992c).

Within the Bayesian framework for data modelling, it is easy to improve our probabilistic models. For example, if we believe that some input variables...
in a problem may be irrelevant to the predicted quantity, but we don’t know which, we can define a new model with multiple hyperparameters that captures the idea of uncertain input variable relevance (MacKay, 1994b; Neal, 1996; MacKay, 1995b); these models then infer automatically from the data which are the relevant input variables for a problem.

44.5 Exercises

Exercise 44.1. How to measure a classifier’s quality. You’ve just written a new classification algorithm and want to measure how well it performs on a test set, and compare it with other classifiers. What performance measure should you use? There are several standard answers. Let’s assume the classifier gives an output $y(x)$, where $x$ is the input, which we won’t discuss further, and that the true target value is $t$. In the simplest discussions of classifiers, both $y$ and $t$ are binary variables, but you might care to consider cases where $y$ and $t$ are more general objects also.

The most widely used measure of performance on a test set is the error rate – the fraction of misclassifications made by the classifier. This measure forces the classifier to give a 0/1 output and ignores any additional information that the classifier might be able to offer – for example, an indication of the firmness of a prediction. Unfortunately, the error rate does not necessarily measure how informative a classifier’s output is. Consider frequency tables showing the joint frequency of the 0/1 output of a classifier (horizontal axis), and the true 0/1 variable (vertical axis). The numbers that we’ll show are percentages. The error rate $e$ is the sum of the two off-diagonal numbers, which we could call the false positive rate $e_+$ and the false negative rate $e_-$. Of the following three classifiers, A and B have the same error rate of 10% and C has a greater error rate of 12%.

<table>
<thead>
<tr>
<th>Classifier A</th>
<th>Classifier B</th>
<th>Classifier C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$t$</td>
<td>$y$</td>
</tr>
<tr>
<td>0</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

But clearly classifier A, which simply guesses that the outcome is 0 for all cases, is conveying no information at all about $t$; whereas classifier B has an informative output: if $y = 0$ then we are sure that $t$ really is zero; and if $y = 1$ then there is a 50% chance that $t = 1$, as compared to the prior probability $P(t = 1) = 0.1$. Classifier C is slightly less informative than B, but it is still much more useful than the information-free classifier A.

One way to improve on the error rate as a performance measure is to report the pair $(e_+, e_-)$, the false positive error rate and the false negative error rate, which are $(0, 0.1)$ and $(0.1, 0)$ for classifiers A and B. It is especially important to distinguish between these two error probabilities in applications where the two sorts of error have different associated costs. However, there are a couple of problems with the ‘error rate pair’:

- First, if I simply told you that classifier A has error rates $(0, 0.1)$ and B has error rates $(0.1, 0)$, it would not be immediately evident that classifier A is actually utterly worthless. Surely we should have a performance measure that gives the worst possible score to A!
- Second, if we turn to a multiple-class classification problem such as digit recognition, then the number of types of error increases from two to

How common sense ranks the classifiers:
(best) $B > C > A$ (worst).

How error rate ranks the classifiers:
(best) $A = B > C$ (worst).
10 × 9 = 90 – one for each possible confusion of class \( t \) with \( t' \). It would be nice to have some sensible way of collapsing these 90 numbers into a single rankable number that makes more sense than the error rate.

Another reason for not liking the error rate is that it doesn’t give a classifier credit for accurately specifying its uncertainty. Consider classifiers that have three outputs available, ‘0’, ‘1’ and a rejection class, ‘?’ , which indicates that the classifier is not sure. Consider classifiers D and E with the following frequency tables, in percentages:

<table>
<thead>
<tr>
<th>Classifier D</th>
<th>y</th>
<th>0</th>
<th>?</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td></td>
<td>74</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>78</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classifier E</th>
<th>y</th>
<th>0</th>
<th>?</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td></td>
<td>74</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>78</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Both of these classifiers have \((e_+, e_-, r) = (6\%, 0\%, 11\%)\). But are they equally good classifiers? Compare classifier E with C. The two classifiers are equivalent. E is just C in disguise – we could make E by taking the output of C and tossing a coin when C says ‘1’ in order to decide whether to give output ‘1’ or ‘?’. So E is equal to C and thus inferior to B. Now compare D with B. Can you justify the suggestion that D is a more informative classifier than B, and thus is superior to E? Yet D and E have the same \((e_+, e_-, r)\) scores.

People often plot error-reject curves (also known as ROC curves; ROC stands for ‘receiver operating characteristic’) which show the total \( e = (e_+ + e_-) \) versus \( r \) as \( r \) is allowed to vary from 0 to 1, and use these curves to compare classifiers (figure 44.7). [In the special case of binary classification problems, \( e_+ \) may be plotted versus \( e_- \) instead.] But as we have seen, error rates can be undiscerning performance measures. Does plotting one error rate as a function of another make this weakness of error rates go away?

For this exercise, either construct an explicit example demonstrating that the error-reject curve, and the area under it, are not necessarily good ways to compare classifiers; or prove that they are.

As a suggested alternative method for comparing classifiers, consider the mutual information between the output and the target,

\[
I(T; Y) \equiv H(T) - H(T|Y) = \sum_{y,t} P(y) P(t|y) \log \frac{P(t)}{P(t|y)},
\]  

(44.12)

which measures how many bits the classifier’s output conveys about the target.

Evaluate the mutual information for classifiers A–E above.

Investigate this performance measure and discuss whether it is a useful one. Does it have practical drawbacks?
About Chapter 45

Feedforward neural networks such as multilayer perceptrons are popular tools for nonlinear regression and classification problems. From a Bayesian perspective, a choice of a neural network model can be viewed as defining a prior probability distribution over nonlinear functions, and the neural network's learning process can be interpreted in terms of the posterior probability distribution over the unknown function. (Some learning algorithms search for the function with maximum posterior probability and other Monte Carlo methods draw samples from this posterior probability.)

In the limit of large but otherwise standard networks, Neal (1996) has shown that the prior distribution over nonlinear functions implied by the Bayesian neural network falls in a class of probability distributions known as Gaussian processes. The hyperparameters of the neural network model determine the characteristic lengthscales of the Gaussian process. Neal's observation motivates the idea of discarding parameterized networks and working directly with Gaussian processes. Computations in which the parameters of the network are optimized are then replaced by simple matrix operations using the covariance matrix of the Gaussian process.

In this chapter I will review work on this idea by Williams and Rasmussen (1996), Neal (1997b), Barber and Williams (1997) and Gibbs and MacKay (2000), and will assess whether, for supervised regression and classification tasks, the feedforward network has been superceded.

Exercise 45.1. I regret that this chapter is rather dry. There's no simple explanatory examples in it, and few pictures. This exercise asks you to create interesting pictures to explain to yourself this chapter's ideas.

Source code for computer demonstrations written in the free language octave is available at:
Radford Neal's software for Gaussian processes is available at:
Gaussian Processes

After the publication of Rumelhart, Hinton and Williams's (1986) paper on supervised learning in neural networks there was a surge of interest in the empirical modelling of relationships in high-dimensional data using nonlinear parametric models such as multilayer perceptrons and radial basis functions. In the Bayesian interpretation of these modelling methods, a nonlinear function \( y(x) \) parameterized by parameters \( \mathbf{w} \) is assumed to underlie the data \( \{ \mathbf{x}^{(n)}, t_n \}_{n=1}^{N} \), and the adaptation of the model to the data corresponds to an inference of the function given the data. We will denote the set of input vectors by \( \mathbf{X}_N \equiv \{ \mathbf{x}^{(n)} \}_{n=1}^{N} \) and the set of corresponding target values by the vector \( \mathbf{t}_N \equiv \{ t_n \}_{n=1}^{N} \). The inference of \( y(x) \) is described by the posterior probability distribution

\[
P(y(x) \mid \mathbf{t}_N, \mathbf{X}_N) = \frac{P(t_N \mid y(x), \mathbf{X}_N)P(y(x))}{P(t_N \mid \mathbf{X}_N)}, \quad (45.1)
\]

Of the two terms on the right-hand side, the first, \( P(t_N \mid y(x), \mathbf{X}_N) \), is the probability of the target values given the function \( y(x) \), which in the case of regression problems is often assumed to be a separable Gaussian distribution; and the second term, \( P(y(x)) \), is the prior distribution on functions assumed by the model. This prior is implicit in the choice of parametric model and the choice of regularizers used during the model fitting. The prior typically specifies that the function \( y(x) \) is expected to be continuous and smooth, and has less high frequency power than low frequency power, but the precise meaning of the prior is somewhat obscured by the use of the parametric model.

Now, from the point of view of prediction of future values of \( t \), all that matters is the assumed prior \( P(y(x)) \) and the assumed noise model \( P(t_N \mid y(x), \mathbf{X}_N) \) – the parameterization of the function \( y(x; \mathbf{w}) \) is irrelevant.

The idea of Gaussian process modelling is to place a prior \( P(y(x)) \) directly on the space of functions, without parameterizing \( y(x) \). The simplest type of prior over functions is called a Gaussian process. It can be thought of as the generalization of a Gaussian distribution over a finite vector space to a function space of infinite dimension. Just as a Gaussian distribution is fully specified by its mean and covariance matrix, a Gaussian process is specified by a mean and a covariance function. Here, the mean is a function of \( x \) (which we will often take to be the zero function), and the covariance is a function \( C(x, x') \) that expresses the expected covariance between the values of the function \( y \) at the points \( x \) and \( x' \). The function \( y(x) \) in any one data modelling problem is assumed to be a single sample from this Gaussian distribution. Gaussian processes are already well established models for various spatial and temporal problems – for example, Brownian motion, Langevin processes and Wiener processes are all examples of Gaussian processes; Kalman filters, widely used
to model speech waveforms, also correspond to Gaussian process models; the method of ‘kriging’ in geostatistics is a Gaussian process regression method.

Reservations about Gaussian processes

It might be thought that it is not possible to reproduce the interesting properties of neural network interpolation methods with something so simple as a Gaussian distribution, but as we shall now see, many popular nonlinear interpolation methods are equivalent to particular Gaussian processes. (I use the term ‘interpolation’ to cover both the problem of ‘regression’ – fitting a curve through noisy data – and the task of fitting an interpolant that passes exactly through the given data points.)

It might also be thought that the computational complexity of inference when we work with priors over infinite-dimensional function spaces might be infinitely large. But by concentrating on the joint probability distribution of the observed data and the quantities we wish to predict, it is possible to make predictions with resources that scale as polynomial functions of \( N \), the number of data points.

45.1 Standard methods for nonlinear regression

The problem

We are given \( N \) data points \( \mathbf{X}_N, \mathbf{t}_N = \{x^{(n)}, t^{(n)}\}_{n=1}^N \). The inputs \( x \) are vectors of some fixed input dimension \( I \). The targets \( t \) are either real numbers, in which case the task will be a regression or interpolation task, or they are categorical variables, for example \( t \in \{0, 1\} \), in which case the task is a classification task. We will concentrate on the case of regression for the time being.

Assuming that a function \( y(x) \) underlies the observed data, the task is to infer the function from the given data, and predict the function’s value – or the value of the observation \( t_{N+1} \) – at a new point \( x^{(N+1)} \).

Parametric approaches to the problem

In a parametric approach to regression we express the unknown function \( y(x) \) in terms of a nonlinear function \( y(x; \mathbf{w}) \) parameterized by parameters \( \mathbf{w} \).

Example 45.2. Fixed basis functions. Using a set of basis functions \( \{\phi_h(x)\}_{h=1}^H \), we can write

\[
y(x; \mathbf{w}) = \sum_{h=1}^{H} w_h \phi_h(x).
\]

(45.2)

If the basis functions are nonlinear functions of \( x \) such as radial basis functions centred at fixed points \( \{c_h\}_{h=1}^H \),

\[
\phi_h(x) = \exp \left[ -\frac{(x - c_h)^2}{2r^2} \right],
\]

(45.3)

then \( y(x; \mathbf{w}) \) is a nonlinear function of \( x \); however, since the dependence of \( y \) on the parameters \( \mathbf{w} \) is linear, we might sometimes refer to this as a ‘linear’ model. In neural network terms, this model is like a multilayer network whose connections from the input layer to the nonlinear hidden layer are fixed; only the output weights \( \mathbf{w} \) are adaptive.

Other possible sets of fixed basis functions include polynomials such as \( \phi_h(x) = x^p x^q \) where \( p \) and \( q \) are integer powers that depend on \( h \).
45.1: Standard methods for nonlinear regression

Example 45.3. Adaptive basis functions. Alternatively, we might make a function \( y(x) \) from basis functions that depend on additional parameters included in the vector \( w \). In a two-layer feedforward neural network with nonlinear hidden units and a linear output, the function can be written

\[
y(x; w) = \sum_{h=1}^{H} w_h^{(2)} \tanh \left( \sum_{i=1}^{I} w_{hi} x_i + w_{h0}^{(1)} \right) + w_0^{(2)}
\]

(45.4)

where \( I \) is the dimensionality of the input space and the weight vector \( w \) consists of the input weights \( \{w_{hi}^{(1)}\} \), the hidden unit biases \( \{w_{h0}^{(1)}\} \), the output weights \( \{w_h^{(2)}\} \) and the output bias \( w_0^{(2)} \). In this model, the dependence of \( y \) on \( w \) is nonlinear.

Having chosen the parameterization, we then infer the function \( y(x; w) \) by inferring the parameters \( w \). The posterior probability of the parameters is

\[
P(w | t_N, X_N) = \frac{P(t_N | w, X_N)P(w)}{P(t_N | X_N)}.
\]

(45.5)

The factor \( P(t_N | w, X_N) \) states the probability of the observed data points when the parameters \( w \) (and hence, the function \( y \)) are known. This probability distribution is often taken to be a separable Gaussian, each data point \( t_n \) differing from the underlying value \( y(x^{(n)}; w) \) by additive noise. The factor \( P(w) \) specifies the prior probability distribution of the parameters. This too is often taken to be a separable Gaussian distribution. If the dependence of \( y \) on \( w \) is nonlinear the posterior distribution \( P(w | t_N, X_N) \) is in general not a Gaussian.

The inference can be implemented in various ways. In the Laplace method, we minimize an objective function

\[
M(w) = -\ln \left[ P(t_N | w, X_N)P(w) \right]
\]

(45.6)

with respect to \( w \), locating the locally most probable parameters, then use the curvature of \( M \), \( \partial^2 M(w)/\partial w_i \partial w_j \), to define error bars on \( w \). Alternatively we can use more general Markov chain Monte Carlo techniques to create samples from the posterior distribution \( P(w | t_N, X_N) \).

Having obtained one of these representations of the inference of \( w \) given the data, predictions are then made by marginalizing over the parameters:

\[
P(t_{N+1} | t_N, X_{N+1}) = \int d^R w \, P(t_{N+1} | w, x^{(N+1)})P(w | t_N, X_N).
\]

(45.7)

If we have a Gaussian representation of the posterior distribution \( P(w | t_N, X_N) \), then this integral can typically be evaluated directly. In the alternative Monte Carlo approach, which generates \( R \) samples \( w^{(r)} \) that are intended to be samples from the posterior distribution \( P(w | t_N, X_N) \), we approximate the predictive distribution by

\[
P(t_{N+1} | t_N, X_{N+1}) \simeq \frac{1}{R} \sum_{r=1}^{R} P(t_{N+1} | w^{(r)}, x^{(N+1)}).
\]

(45.8)

Nonparametric approaches.

In nonparametric methods, predictions are obtained without explicitly parameterizing the unknown function \( y(x) \); \( y(x) \) lives in the infinite-dimensional
space of all continuous functions of \( x \). One well known nonparametric approach to the regression problem is the spline smoothing method (Kimeldorf and Wahba, 1970). A spline solution to a one-dimensional regression problem can be described as follows: we define the estimator of \( y(x) \) to be the function \( \hat{y}(x) \) that minimizes the functional

\[
M(y(x)) = \frac{1}{2} \beta \sum_{n=1}^{N} (y(x^{(n)}) - t_n)^2 + \frac{1}{2} \alpha \int dx \mid y^{(p)}(x) \mid^2,
\]

(45.9)

where \( y^{(p)} \) is the \( p \)th derivative of \( y \) and \( p \) is a positive number. If \( p \) is set to \( 2 \) then the resulting function \( \hat{y}(x) \) is a cubic spline, that is, a piecewise cubic function that has ‘knots’ – discontinuities in its second derivative – at the data points \( \{x^{(n)}\} \).

This estimation method can be interpreted as a Bayesian method by identifying the prior for the function \( y(x) \) as:

\[
\ln P(y(x) | \alpha) = -\frac{1}{2} \alpha \int dx \mid y^{(p)}(x) \mid^2 + \text{const},
\]

(45.10)

and the probability of the data measurements \( t_N = \{t_n\}_{n=1}^{N} \) assuming independent Gaussian noise as:

\[
\ln P(t_N | y(x), \beta) = -\frac{1}{2} \beta \sum_{n=1}^{N} (y(x^{(n)}) - t_n)^2 + \text{const}.
\]

(45.11)

[The constants in equations (45.10) and (45.11) are functions of \( \alpha \) and \( \beta \) respectively. Strictly the prior (45.10) is improper since addition of an arbitrary polynomial of degree \( (p - 1) \) to \( y(x) \) is not constrained. This impropriety is easily rectified by the addition of \( (p - 1) \) appropriate terms to (45.10).] Given this interpretation of the functions in equation (45.9), \( M(y(x)) \) is equal to minus the log of the posterior probability \( P(y(x) | t_N, \alpha, \beta) \), within an additive constant, and the splines estimation procedure can be interpreted as yielding a Bayesian MAP estimate. The Bayesian perspective allows us additionally to put error bars on the splines estimate and to draw typical samples from the posterior distribution, and it gives an automatic method for inferring the hyperparameters \( \alpha \) and \( \beta \).

Comments

Splines priors are Gaussian processes

The prior distribution defined in equation (45.10) is our first example of a Gaussian process. Throwing mathematical precision to the winds, a Gaussian process can be defined as a probability distribution on a space of functions \( y(x) \) that can be written in the form

\[
P(y(x) | \mu(x), A) = \frac{1}{Z} \exp \left[ -\frac{1}{2} (y(x) - \mu(x))^T A (y(x) - \mu(x)) \right],
\]

(45.12)

where \( \mu(x) \) is the mean function and \( A \) is a linear operator, and where the inner product of two functions \( y(x)^T z(x) \) is defined by, for example, \( \int dx y(x) z(x) \). Here, if we denote by \( D \) the linear operator that maps \( y(x) \) to the derivative of \( y(x) \), we can write equation (45.10) as

\[
\ln P(y(x) | \alpha) = -\frac{1}{2} \alpha \int dx \mid D^p y(x) \mid^2 + \text{const} = -\frac{1}{2} y(x)^T A y(x) + \text{const},
\]

(45.13)
which has the same form as equation (45.12) with \( \mu(x) = 0 \), and \( A \equiv [D^p]^T D^p \).

In order for the prior in equation (45.12) to be a proper prior, \( A \) must be a positive definite operator, i.e., one satisfying \( y(x)^T A y(x) > 0 \) for all functions \( y(x) \) other than \( y(x) = 0 \).

**Splines can be written as parametric models**

Splines may be written in terms of an infinite set of fixed basis functions, as in equation (45.2), as follows. First rescale the \( x \) axis so that the interval \( (0, 2\pi) \) is much wider than the range of \( x \) values of interest. Let the basis functions be a Fourier set \( \{\cos h x, \sin h x, \ h = 0, 1, 2, \ldots\} \), so the function is

\[
y(x) = \sum_{h=0}^{\infty} w_{h}(\cos) \cos(h x) + \sum_{h=1}^{\infty} w_{h}(\sin) \sin(h x).
\]  

(45.14)

Use the regularizer

\[
E_{W}(w) = \sum_{h=0}^{\infty} \frac{1}{2} h^2 w_{h}(\cos)^2 + \sum_{h=1}^{\infty} \frac{1}{2} h^2 w_{h}(\sin)^2
\]  

(45.15)

to define a Gaussian prior on \( w \),

\[
P(w | \alpha) = \frac{1}{Z_{W}(\alpha)} \exp(-\alpha E_{W}).
\]  

(45.16)

If \( p = 2 \) then we have the cubic splines regularizer \( E_{W}(w) = \int y^{(2)}(x)^2 \, dx \), as in equation (45.9); if \( p = 1 \) we have the regularizer \( E_{W}(w) = \int y^{(1)}(x)^2 \, dx \), etc. (To make the prior proper we must add an extra regularizer on the term \( w_{0}(\cos) \).) Thus in terms of the prior \( P(y(x)) \) there is no fundamental difference between the ‘nonparametric’ splines approach and other parametric approaches.

**Representation is irrelevant for prediction**

From the point of view of prediction at least, there are two objects of interest. The first is the conditional distribution \( P(t_{N+1} | t_N, X_{N+1}) \) defined in equation (45.7). The other object of interest, should we wish to compare one model with others, is the joint probability of all the observed data given the model, the evidence \( P(t_N | X_N) \), which appeared as the normalizing constant in equation (45.5). Neither of these quantities makes any reference to the representation of the unknown function \( y(x) \). So at the end of the day, our choice of representation is irrelevant.

The question we now address is, in the case of popular parametric models, what form do these two quantities take? We will see that for standard models with fixed basis functions and Gaussian distributions on the unknown parameters, the joint probability of all the observed data given the model, \( P(t_N | X_N) \), is a multivariate Gaussian distribution with mean zero and with a covariance matrix determined by the basis functions; this implies that the conditional distribution \( P(t_{N+1} | t_N, X_{N+1}) \) is also a Gaussian distribution, whose mean depends linearly on the values of the targets \( t_N \). Standard parametric models are simple examples of Gaussian processes.

### 45.2 From parametric models to Gaussian processes

**Linear models**

Let us consider a regression problem using \( H \) fixed basis functions, for example one-dimensional radial basis functions as defined in equation (45.3).
Let us assume that a list of \( N \) input points \( \{ x^{(n)} \} \) has been specified and define the \( N \times H \) matrix \( R \) to be the matrix of values of the basis functions \( \{ \phi_h(x) \}_{h=1}^{H} \) at the points \( \{ x_n \} \),

\[
R_{nh} \equiv \phi_h(x^{(n)}). \tag{45.17}
\]

We define the vector \( y_N \) to be the vector of values of \( y(x) \) at the \( N \) points,

\[
y_n \equiv \sum_h R_{nh} w_h. \tag{45.18}
\]

If the prior distribution of \( w \) is Gaussian with zero mean,

\[
P(w) = \text{Normal}(w; 0, \sigma_w^2 I), \tag{45.19}
\]

then \( y \), being a linear function of \( w \), is also Gaussian distributed, with mean zero. The covariance matrix of \( y \) is

\[
Q = \langle yy^\top \rangle = \langle R w w^\top R^\top \rangle = R \langle w w^\top \rangle R^\top = \sigma_w^2 R R^\top. \tag{45.20}
\]

So the prior distribution of \( y \) is:

\[
P(y) = \text{Normal}(y; 0, Q) = \text{Normal}(y; 0, \sigma_w^2 R R^\top). \tag{45.22}
\]

This result, that the vector of \( N \) function values \( y \) has a Gaussian distribution, is true for any selected points \( X_N \). This is the defining property of a Gaussian process. The probability distribution of a function \( y(x) \) is a Gaussian process if for any finite selection of points \( x^{(1)}, x^{(2)}, \ldots, x^{(N)} \), the density \( P(y(x^{(1)}), y(x^{(2)}), \ldots, y(x^{(N)})) \) is a Gaussian.

Now, if the number of basis functions \( H \) is smaller than the number of data points \( N \), then the matrix \( Q \) will not have full rank. In this case the probability distribution of \( y \) might be thought of as a flat elliptical pancake confined to an \( H \)-dimensional subspace in the \( N \)-dimensional space in which \( y \) lives.

What about the target values? If each target \( t_n \) is assumed to differ by additive Gaussian noise of variance \( \sigma_t^2 \) from the corresponding function value \( y_n \) then \( t \) also has a Gaussian prior distribution,

\[
P(t) = \text{Normal}(t; 0, Q + \sigma_t^2 I). \tag{45.23}
\]

We will denote the covariance matrix of \( t \) by \( C \):

\[
C = Q + \sigma_t^2 I = \sigma_w^2 R R^\top + \sigma_t^2 I. \tag{45.24}
\]

Whether or not \( Q \) has full rank, the covariance matrix \( C \) has full rank since \( \sigma_t^2 I \) is full rank.

What does the covariance matrix \( Q \) look like? In general, the \( (n, n') \) entry of \( Q \) is

\[
Q_{nn'} = [\sigma_w^2 R R^\top]_{nn'} = \sigma_w^2 \sum_h \phi_h(x^{(n)}) \phi_h(x^{(n')}) \tag{45.25}
\]

and the \( (n, n') \) entry of \( C \) is

\[
C_{nn'} = \sigma_w^2 \sum_h \phi_h(x^{(n)}) \phi_h(x^{(n')}) + \delta_{nn'} \sigma_t^2, \tag{45.26}
\]

where \( \delta_{nn'} = 1 \) if \( n = n' \) and 0 otherwise.
Example 45.4. Let’s take as an example a one-dimensional case, with radial basis functions. The expression for $Q_{nn'}$ becomes simplest if we assume we have uniformly spaced basis functions with the basis function labelled $h$ being centred on the point $x = h$ and take the limit $H \to \infty$, so that the sum over $h$ becomes an integral; to avoid having a covariance that diverges with $H$, we had better make $\sigma^2_p$ scale as $S = \left(\frac{H}{H}\right)$, where $H$ is the number of basis functions per unit length of the $x$-axis, and $S$ is a constant; then

$$Q_{nn'} = S \int_{h_{\text{min}}}^{h_{\text{max}}} dh \phi_h(x^{(n)}) \phi_h(x^{(n')})$$

$$= S \int_{h_{\text{min}}}^{h_{\text{max}}} dh \exp \left[-\frac{(x^{(n)} - h)^2}{2\sigma^2_p}\right] \exp \left[-\frac{(x^{(n')} - h)^2}{2\sigma^2_p}\right].$$

(45.27)

(45.28)

If we let the limits of integration be $\pm\infty$, we can solve this integral:

$$Q_{nn'} = \sqrt{\pi \sigma^2_p} S \exp \left[-\frac{(x^{(n')} - x^{(n)})^2}{4\sigma^2_p}\right].$$

(45.29)

We are arriving at a new perspective on the interpolation problem. Instead of specifying the prior distribution on functions in terms of basis functions and priors on parameters, the prior can be summarized simply by a covariance function,

$$C(x^{(n)}, x^{(n')}) \equiv \theta_1 \exp \left[-\frac{(x^{(n')} - x^{(n)})^2}{4\sigma^2_p}\right],$$

(45.30)

where we have given a new name, $\theta_1$, to the constant out front.

Generalizing from this particular case, a vista of interpolation methods opens up. Given any valid covariance function $C(x, x')$ – we’ll discuss in a moment what ‘valid’ means – we can define the covariance matrix for $N$ function values at locations $X_N$ to be the matrix $Q$ given by

$$Q_{nn'} = C(x^{(n)}, x^{(n')})$$

(45.31)

and the covariance matrix for $N$ corresponding target values, assuming Gaussian noise, to be the matrix $C$ given by

$$C_{nn'} = C(x^{(n)}, x^{(n')}) + \sigma^2_p \delta_{nn'}.$$ 

(45.32)

In conclusion, the prior probability of the $N$ target values $t$ in the data set is:

$$P(t) = \text{Normal}(t; \mathbf{0}, C) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2} t^T C^{-1} t}.$$ 

(45.33)

Samples from this Gaussian process and a few other simple Gaussian processes are displayed in figure 45.1.

**Multilayer neural networks and Gaussian processes**

Figures 44.2 and 44.3 show some random samples from the prior distribution over functions defined by a selection of standard multilayer perceptrons with large numbers of hidden units. Those samples don’t seem a million miles away from the Gaussian process samples of figure 45.1. And indeed Neal (1996) showed that the properties of a neural network with one hidden layer (as in equation (45.4)) converge to those of a Gaussian process as the number of hidden neurons tends to infinity, if standard ‘weight decay’ priors are assumed. The covariance function of this Gaussian process depends on the details of the priors assumed for the weights in the network and the activation functions of the hidden units.
Figure 45.1. Samples drawn from Gaussian process priors. Each panel shows two functions drawn from a Gaussian process prior. The four corresponding covariance functions are given below each plot. The decrease in length scale from (a) to (b) produces more rapidly fluctuating functions. The periodic properties of the covariance function in (c) can be seen. The covariance function in (d) contains the non-stationary term \(xx'\) corresponding to the covariance of a straight line, so that typical functions include linear trends. From Gibbs (1997).

45.3 Using a given Gaussian process model in regression

We have spent some time talking about priors. We now return to our data and the problem of prediction. How do we make predictions with a Gaussian process?

Having formed the covariance matrix \(C\) defined in equation (45.32) our task is to infer \(t_{N+1}\) given the observed vector \(t_N\). The joint density \(P(t_{N+1}, t_N)\) is a Gaussian; so the conditional distribution

\[
P(t_{N+1} | t_N) = \frac{P(t_{N+1}, t_N)}{P(t_N)} \tag{45.34}
\]

is also a Gaussian. We now distinguish between different sizes of covariance matrix \(C\) with a subscript, such that \(C_{N+1}\) is the \((N+1) \times (N+1)\) covariance matrix for the vector \(t_{N+1} = (t_1, \ldots, t_{N+1})^T\). We define submatrices of \(C_{N+1}\) as follows:

\[
C_{N+1} \equiv \begin{bmatrix} C_N & k \\ k' & \kappa \end{bmatrix} \tag{45.35}
\]

The posterior distribution (45.34) is given by

\[
P(t_{N+1} | t_N) \propto \exp \left[ -\frac{1}{2} \begin{bmatrix} t_N & t_{N+1} \end{bmatrix} C_{N+1}^{-1} \begin{bmatrix} t_N \\ t_{N+1} \end{bmatrix} \right]. \tag{45.36}
\]

We can evaluate the mean and standard deviation of the posterior distribution of \(t_{N+1}\) by brute force inversion of \(C_{N+1}\). There is a more elegant expression

\[
\begin{align*}
&\frac{1}{2} \begin{bmatrix} C_N & k \\ k' & \kappa \end{bmatrix} \begin{bmatrix} C_N & k \\ k' & \kappa \end{bmatrix}^{-1} \begin{bmatrix} C_N & k \\ k' & \kappa \end{bmatrix}^{-1} \\
&= C_N + k \kappa^{-1} k' - k \kappa^{-1} C_N \kappa^{-1} k'.
\end{align*}
\]
45.4: Examples of covariance functions

for the predictive distribution, however, which is useful whenever predictions are to be made at a number of new points on the basis of the data set of size $N$. We can write $C_{N+1}^{-1}$ in terms of $C_N$ and $C_{N+1}^{-1}$ using the partitioned inverse equations (Barnett, 1979)

$$C_{N+1}^{-1} = \begin{bmatrix} M & m \\ m^T & m \end{bmatrix}$$ (45.37)

where

$$m = \left( \kappa - k^T C_N^{-1} k \right)^{-1}$$ (45.38)

$$m = -m C_N^{-1} k$$ (45.39)

$$M = C_N^{-1} + \frac{1}{m} m m^T.$$ (45.40)

When we substitute this matrix into equation (45.36) we find

$$P(t_{N+1} | t_N) = \frac{1}{Z} \exp \left[ -\frac{(t_{N+1} - \hat{t}_{N+1})^2}{2\sigma_{t_{N+1}}^2} \right]$$ (45.41)

where

$$\hat{t}_{N+1} = k^T C_N^{-1} t_N$$ (45.42)

$$\sigma_{t_{N+1}}^2 = \kappa - k^T C_N^{-1} k.$$ (45.43)

The predictive mean at the new point is given by $\hat{t}_{N+1}$ and $\sigma_{t_{N+1}}$ defines the error bars on this prediction. Notice that we do not need to invert $C_{N+1}$ in order to make predictions at $x^{(N+1)}$. Only $C_N$ needs to be inverted. Thus Gaussian processes allow one to implement a model with a number of basis functions $H$ much larger than the number of data points $N$, with the computational requirement being of order $N^3$, independent of $H$. [We’ll discuss ways of reducing this cost later.]

The predictions produced by a Gaussian process depend entirely on the covariance matrix $C$. We now discuss the sorts of covariance functions one might choose to define $C$, and how we can automate the selection of the covariance function in response to data.

45.4 Examples of covariance functions

The only constraint on our choice of covariance function is that it must generate a non-negative-definite covariance matrix for any set of points $(x_n)_{n=1}^N$. We will denote the parameters of a covariance function by $\theta$. The covariance matrix of $t$ has entries given by

$$C_{mn} = C(x^{(m)}, x^{(n)}; \theta) + \delta_{mn} \mathcal{N}(x^{(n)}; \theta)$$ (45.44)

where $C$ is the covariance function and $\mathcal{N}$ is a noise model which might be stationary or spatially varying, for example,

$$\mathcal{N}(x; \theta) = \left\{ \begin{array}{ll} \theta_3 & \text{for input-independent noise} \\
\exp \left( \sum_{j=1}^J \beta_j \phi_j(x) \right) & \text{for input-dependent noise.} \end{array} \right.$$ (45.45)

The continuity properties of $C$ determine the continuity properties of typical samples from the Gaussian process prior. An encyclopaedic paper on Gaussian processes giving many valid covariance functions has been written by Abrahamsen (1997).
Stationary covariance functions

A stationary covariance function is one that is translation invariant in that it satisfies

$$C(x, x'; \theta) = D(x - x'; \theta)$$

(45.46)

for some function $D$, i.e., the covariance is a function of separation only, also known as the autocovariance function. If additionally $C$ depends only on the magnitude of the distance between $x$ and $x'$, then the covariance function is said to be homogenous. Stationary covariance functions may also be described in terms of the Fourier transform of the function $D$, which is known as the power spectrum of the Gaussian process. This Fourier transform is necessarily a positive function of frequency. One way of constructing a valid stationary covariance function is to invent a positive function of frequency and define $D$ to be its inverse Fourier transform.

Example 45.5. Let the power spectrum be a Gaussian function of frequency.

Since the Fourier transform of a Gaussian is a Gaussian, the autocovariance function corresponding to this power spectrum is a Gaussian function of separation. This argument rederives the covariance function we derived at equation (45.30).

Generalizing slightly, a popular form for $C$ with hyperparameters $\theta = (\theta_1, \theta_2, \{r_i\})$ is

$$C(x, x'; \theta) = \theta_1 \exp \left[ -\frac{1}{2} \sum_{i=1}^{I} \frac{(x_i - x'_i)^2}{r_i^2} \right] + \theta_2.$$  

(45.47)

$x$ is an $I$-dimensional vector and $r_i$ is a length scale associated with input $x_i$, the lengthscale in that direction on which $y$ is expected to vary significantly. A very large length scale means that $y$ is expected to be essentially a constant function of that input. Such an input could be said to be irrelevant, as in the automatic relevance determination method for neural networks (MacKay, 1994a; Neal, 1996). The $\theta_1$ hyperparameter defines the vertical scale of variations of a typical function. The $\theta_2$ hyperparameter allows the whole function to be offset away from zero by some unknown constant — to understand this term, examine equation (45.25) and consider the basis function $\phi(x) = 1$.

Another stationary covariance function is

$$C(x, x') = \exp(-|x - x'|^{\nu}) \quad 0 < \nu \leq 2.$$  

(45.48)

For $\nu = 2$, this is a special case of the previous covariance function. For $\nu \in (1, 2)$, the typical functions from this prior are smooth but not analytic functions. For $\nu \leq 1$ typical functions are continuous but not smooth.

A covariance function that models a function that is periodic with known period $\lambda_i$ in the $i^{th}$ input direction is

$$C(x, x'; \theta) = \theta_1 \exp \left[ -\frac{1}{2} \sum_{i} \left( \sin \left( \frac{\pi (x_i - x'_i)}{\lambda_i} \right) \right)^2 \right].$$  

(45.49)

Figure 45.1 shows some random samples drawn from Gaussian processes with a variety of different covariance functions.
45.5: Adaptation of Gaussian process models

Nonstationary covariance functions

The simplest nonstationary covariance function is the one corresponding to a linear trend. Consider the plane \( y(x) = \sum_i w_i x_i + c \). If the \( \{w_i\} \) and \( c \) have Gaussian distributions with zero mean and variances \( \sigma^2_w \) and \( \sigma^2_c \) respectively then the plane has a covariance function

\[
C_{\text{lin}}(x, x'; \{\sigma_w, \sigma_c\}) = \sum_{i=1}^{I} \sigma^2_w x_i x_i' + \sigma^2_c.
\] (45.50)

An example of random sample functions incorporating the linear term can be seen in figure 45.1d.

45.5 Adaptation of Gaussian process models

Let us assume that a form of covariance function has been chosen, but that it depends on undetermined hyperparameters \( \theta \). We would like to ‘learn’ these hyperparameters from the data. This learning process is equivalent to the inference of the hyperparameters of a neural network, for example, weight decay hyperparameters. It is a complexity control problem, one that is solved nicely by the Bayesian Occam’s razor.

Ideally we would like to define a prior distribution on the hyperparameters and integrate over them in order to make our predictions, i.e., we would like to find

\[
P(t_{N+1} | x_{N+1}, D) = \int P(t_{N+1} | x_{N+1}, \theta, D) P(\theta | D) d\theta.
\] (45.51)

But this integral is usually intractable. There are two approaches we can take.
1. We can approximate the integral by using the most probable values of hyperparameters.

\[ P(t_{N+1} | \mathbf{x}_{N+1}, D) \simeq P(t_{N+1} | \mathbf{x}_{N+1}, D, \theta_{MP}) \]  

(45.52)

2. Or we can perform the integration over \( \theta \) numerically using Monte Carlo methods (Williams and Rasmussen, 1996; Neal, 1997b).

Either of these approaches is implemented most efficiently if the gradient of the posterior probability of \( \theta \) can be evaluated.

Gradient

The posterior probability of \( \theta \) is

\[ P(\theta | D) \propto P(t_N | X_N, \theta) P(\theta). \]  

(45.53)

The log of the first term (the evidence for the hyperparameters) is

\[ \ln P(t_N | X_N, \theta) = -\frac{1}{2} \ln \det \mathbf{C}_N - \frac{1}{2} \mathbf{t}_N^T \mathbf{C}_N^{-1} \mathbf{t}_N - \frac{N}{2} \ln 2\pi, \]  

(45.54)

and its derivative with respect to a hyperparameter \( \theta \) is

\[ \frac{\partial}{\partial \theta} \ln P(t_N | X_N, \theta) = -\frac{1}{2} \text{Trace} \left( \mathbf{C}_N^{-1} \frac{\partial \mathbf{C}_N}{\partial \theta} \right) + \frac{1}{2} \mathbf{t}_N^T \mathbf{C}_N^{-1} \frac{\partial \mathbf{C}_N}{\partial \theta} \mathbf{C}_N^{-1} \mathbf{t}_N. \]  

(45.55)

Comments

Assuming that finding the derivatives of the priors is straightforward, we can now search for \( \theta_{MP} \). However there are two problems that we need to be aware of. Firstly, as illustrated in figure 45.2, the evidence may be multimodal. Suitable priors and sensible optimization strategies often eliminate poor optima. Secondly and perhaps most importantly the evaluation of the gradient of the log likelihood requires the evaluation of \( \mathbf{C}_N^{-1} \). Any exact inversion method (such as Cholesky decomposition, LU decomposition or Gauss–Jordan elimination) has an associated computational cost that is of order \( N^3 \) and so calculating gradients becomes time consuming for large training data sets. Approximate methods for implementing the predictions (equations (45.42) and (45.43)) and gradient computation (equation (45.55)) are an active research area. One approach based on the ideas of Skilling (1993) makes approximations to \( \mathbf{C}^{-1} \mathbf{t} \) and \( \text{Trace} \mathbf{C}^{-1} \) using iterative methods with cost \( \mathcal{O}(N^2) \) (Gibbs and MacKay, 1996; Gibbs, 1997). Further references on this topic are given at the end of the chapter.

45.6 Classification

Gaussian processes can be integrated into classification modelling once we identify a variable that can sensibly be given a Gaussian process prior.

In a binary classification problem, we can define a quantity \( a_n \equiv a(x^{(n)}) \) such that the probability that the class is 1 rather than 0 is

\[ P(t_n = 1 | a_n) = \frac{1}{1 + e^{-a_n}}. \]  

(45.56)

Large positive values of \( a \) correspond to probabilities close to one; large negative values of \( a \) define probabilities that are close to zero. In a classification problem, we typically intend that the probability \( P(t_n = 1) \) should be a smoothly varying function of \( x \). We can embody this prior belief by defining \( a(x) \) to have a Gaussian process prior.
Implementation

It is not so easy to perform inferences and adapt the Gaussian process model to data in a classification model as in regression problems because the likelihood function (45.56) is not a Gaussian function of $a_n$. So the posterior distribution of $a$ given some observations $t$ is not Gaussian and the normalization constant $P(t_X \mid X_N)$ cannot be written down analytically. Barber and Williams (1997) have implemented classifiers based on Gaussian process priors using Laplace approximations (Chapter 27). Neal (1997b) has implemented a Monte Carlo approach to implementing a Gaussian process classifier. Gibbs and MacKay (2000) have implemented another cheap and cheerful approach based on the methods of Jaakkola and Jordan (section 33.8). In this variational Gaussian process classifier, we obtain tractable upper and lower bounds for the unnormalized posterior density over $a$, $P(t_X \mid a)P(a)$. These bounds are parameterized by variational parameters which are adjusted in order to obtain the tightest possible fit. Using normalized versions of the optimized bounds we then compute approximations to the predictive distributions.

Multi-class classification problems can also be solved with Monte Carlo methods (Neal, 1997b) and variational methods (Gibbs, 1997).

45.7 Discussion

Gaussian processes are moderately simple to implement and use. Because very few parameters of the model need to be determined by hand (generally only the priors on the hyperparameters), Gaussian processes are useful tools for automated tasks where fine tuning for each problem is not possible. We do not appear to sacrifice any performance for this simplicity.

It is easy to construct Gaussian processes that have particular desired properties; for example we can make a straightforward automatic relevance determination model.

One obvious problem with Gaussian processes is the computational cost associated with inverting an $N \times N$ matrix. The cost of direct methods of inversion becomes prohibitive when the number of data points $N$ is greater than about 1000.

Have we thrown the baby out with the bath water?

According to the hype of 1987, neural networks were meant to be intelligent models that discovered features and patterns in data. Gaussian processes in contrast are simply smoothing devices. How can Gaussian processes possibly replace neural networks? Were neural networks over-hyped, or have we underestimated the power of smoothing methods?

I think both these propositions are true. The success of Gaussian processes shows that many real-world data modelling problems are perfectly well solved by sensible smoothing methods. The most interesting problems, the task of feature discovery for example, are not ones that Gaussian processes will solve. But maybe multilayer perceptrons can’t solve them either. Perhaps a fresh start is needed, approaching the problem of machine learning from a paradigm different from the supervised feedforward mapping.

Further reading

The study of Gaussian processes for regression is far from new. Time series analysis was being performed by the astronomer T.N. Thiele using Gaussian
processes in 1880 (Lauritzen, 1981). In the 1940s, Wiener–Kolmogorov prediction theory was introduced for prediction of trajectories of military targets (Wiener, 1948). Within the geostatistics field, Matheron (1963) proposed a framework for regression using optimal linear estimators which he called ‘kriging’ after D.G. Krige, a South African mining engineer. This framework is identical to the Gaussian process approach to regression. Kriging has been developed considerably in the last thirty years (see Cressie (1993) for a review) including several Bayesian treatments (Omre, 1987; Kitanidis, 1986). However the geostatistics approach to the Gaussian process model has concentrated mainly on low-dimensional problems and has largely ignored any probabilistic interpretation of the model. Kalman filters are widely used to implement inferences for stationary one-dimensional Gaussian processes, and are popular models for speech and music modelling (Bar-Shalom and Fortman, 1988). Generalized radial basis functions (Poggio and Girosi, 1989), ARMA models (Wahba, 1990) and variable metric kernel methods (Lowe, 1995) are all closely related to Gaussian processes. See also O’Hagan (1978).

The idea of replacing supervised neural networks by Gaussian processes was first explored by Williams and Rasmussen (1996) and Neal (1997b). A thorough comparison of Gaussian processes with other methods such as neural networks and MARS was made by Rasmussen (1996). Methods for reducing the complexity of data modelling with Gaussian processes remain an active research area (Poggio and Girosi, 1990; Luo and Wahba, 1997; Tresp, 2000; Williams and Seeger, 2001; Smola and Bartlett, 2001; Rasmussen, 2002; Seeger et al., 2003; Opper and Winther, 2000).

A longer review of Gaussian processes is in (MacKay, 1998b). A review paper on regression with complexity control using hierarchical Bayesian models is (MacKay, 1992a).

Gaussian processes and support vector learning machines (Scholkopf et al., 1995; Vapnik, 1995) have a lot in common. Both are kernel-based predictors, the kernel being another name for the covariance function. A Bayesian version of support vectors, exploiting this connection, can be found in (Chu et al., 2001; Chu et al., 2002; Chu et al., 2003b; Chu et al., 2003a).
46

Deconvolution

46.1 Traditional image reconstruction methods

Optimal linear filters

In many imaging problems, the data measurements \( \{d_n\} \) are linearly related to the underlying image \( \mathbf{f} \):

\[
d_n = \sum_k R_{nk} f_k + n_n. \tag{46.1}
\]

The vector \( \mathbf{n} \) denotes the inevitable noise that corrupts real data. In the case of a camera which produces a blurred picture, the vector \( \mathbf{f} \) denotes the true image, \( \mathbf{d} \) denotes the blurred and noisy picture, and the linear operator \( \mathbf{R} \) is a convolution defined by the point spread function of the camera. In this special case, the true image and the data vector reside in the same space; but it is important to maintain a distinction between them. We will use the subscript \( n = 1, \ldots, N \) to run over data measurements, and the subscripts \( k, k' = 1, \ldots, K \) to run over image pixels.

One might speculate that since the blur was created by a linear operation, then perhaps it might be deblurred by another linear operation. We can derive the optimal linear filter in two ways.

Bayesian derivation

We assume that the linear operator \( \mathbf{R} \) is known, and that the noise \( \mathbf{n} \) is Gaussian and independent, with a known standard deviation \( \sigma_n \).

\[
P(\mathbf{d} | \mathbf{f}, \sigma_n, \mathcal{H}) = \frac{1}{(2\pi\sigma_n^2)^{N/2}} \exp \left( -\sum_n (d_n - \sum_k R_{nk} f_k)^2 / (2\sigma_n^2) \right). \tag{46.2}
\]

We assume that the prior probability of the image is also Gaussian, with a scale parameter \( \sigma_f \).

\[
P(\mathbf{f} | \sigma_f, \mathcal{H}) = \frac{\det^{-1/2} \mathbf{C}}{(2\pi\sigma_f^2)^{K/2}} \exp \left( -\sum_{k,k'} f_k C_{kk'} f_{k'} / (2\sigma_f^2) \right). \tag{46.3}
\]

If we assume no correlations among the pixels then the symmetric, full rank matrix \( \mathbf{C} \) is equal to the identity matrix \( \mathbf{I} \). The more sophisticated ‘intrinsic correlation function’ model uses \( \mathbf{C} = [\mathbf{G}\mathbf{G}^T]^{-1} \), where \( \mathbf{G} \) is a convolution that takes us from an imaginary ‘hidden’ image, which is uncorrelated, to the real correlated image. The intrinsic correlation function should not be confused with the point spread function \( \mathbf{R} \) which defines the image-to-data mapping.
A zero-mean Gaussian prior is clearly a poor assumption if it is known that all elements of the image \( f \) are positive, but let us proceed. We can now write down the posterior probability of an image \( f \) given the data \( d \).

\[
P(f \mid d, \sigma_\nu, \sigma_f, \mathcal{H}) = \frac{P(d \mid f, \sigma_\nu, \sigma_f, \mathcal{H}) P(f \mid \sigma_f, \mathcal{H})}{P(d \mid \sigma_\nu, \sigma_f, \mathcal{H})}.
\]  
(46.4)

In words,

\[
\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}.
\]  
(46.5)

The ‘evidence’ \( P(d \mid \sigma_\nu, \sigma_f, \mathcal{H}) \) is the normalizing constant for this posterior distribution. Here it is unimportant, but it is used in a more sophisticated analysis to compare, for example, different values of \( \sigma_\nu \) and \( \sigma_f \), or different point spread functions \( R \).

Since the posterior distribution is the product of two Gaussian functions of \( f \), it is also a Gaussian, and can therefore be summarized by its mean, which is also the most probable image, \( f_{\text{MP}} \), and its covariance matrix:

\[
\Sigma_{f_{\text{MP}}} \equiv \left[ -\nabla^2 \log P(f \mid d, \sigma_\nu, \sigma_f, \mathcal{H}) \right]^{-1},
\]  
(46.6)

which defines the joint error bars on \( f \). In this equation, the symbol \( \nabla \) denotes differentiation with respect to the image parameters \( f \). We can find \( f_{\text{MP}} \) by differentiating the log of the posterior, and solving for the derivative being zero. We obtain:

\[
f_{\text{MP}} = \left[ R^T R + \frac{\sigma_\nu^2}{\sigma_f^2} C \right]^{-1} R^T d.
\]  
(46.7)

The operator \( \left[ R^T R + \frac{\sigma_\nu^2}{\sigma_f^2} C \right]^{-1} R^T \) is called the optimal linear filter. When the term \( \frac{\sigma_\nu^2}{\sigma_f^2} C \) can be neglected, the optimal linear filter is the pseudoinverse \( [R^T R]^{-1} R^T \). The term \( \frac{\sigma_\nu^2}{\sigma_f^2} C \) regularizes this ill-conditioned inverse.

The optimal linear filter can also be manipulated into the form:

\[
\text{Optimal linear filter} = C^{-1} R^T \left[ R C^{-1} R^T + \frac{\sigma_\nu^2}{\sigma_f^2} I \right]^{-1}.
\]  
(46.8)

**Minimum square error derivation**

The non-Bayesian derivation of the optimal linear filter starts by assuming that we will ‘estimate’ the true image \( f \) by a linear function of the data:

\[
\hat{f} = W d.
\]  
(46.9)

The linear operator \( W \) is then ‘optimized’ by minimizing the expected sum-of-squared error between \( f \) and the unknown true image \( f \). In the following equations, summations over repeated indices \( k, k', n \) are implicit. The expectation \( \langle \cdot \rangle \) is over both the statistics of the random variables \( \{ u_n \} \), and the ensemble of images \( f \) which we expect to bump into. We assume that the noise is zero mean and uncorrelated to second order with itself and everything else, with \( \langle u_n u_{n'} \rangle = \sigma_u^2 \delta_{nn'} \).

\[
\langle E \rangle = \frac{1}{2} \left( (W_{kn} u_n - f_k)^2 \right)
\]  
(46.10)

\[
= \frac{1}{2} \left( (W_{kn} R_{nj} f_j - f_k)^2 \right) + \frac{1}{2} W_{kn} W_{kn} \sigma_u^2.
\]  
(46.11)
46.1: Traditional image reconstruction methods

Differentiating with respect to \( W \), and introducing \( F \equiv \langle f_j f_j \rangle \) (c.f. \( \sigma_f^2 C^{-1} \) in the Bayesian derivation above), we find that the optimal linear filter is:

\[
W_{\text{opt}} = FR^T \left[ RFR^T + \sigma_f^2 I \right]^{-1}.
\]

(46.12)

If we identify \( F = \sigma_f^2 C^{-1} \), we obtain the optimal linear filter (46.8) of the Bayesian derivation. The ad hoc assumptions made in this derivation were the choice of a quadratic error measure, and the decision to use a linear estimator. It is interesting that without explicit assumptions of Gaussian distributions, this derivation has reproduced the same estimator as the Bayesian posterior mode, \( f_{\text{MP}} \).

The advantage of a Bayesian approach is that we can criticize these assumptions and modify them in order to make better reconstructions.

**Other image models**

The better matched our model of images \( P(f \mid \mathcal{H}) \) is to the real world, the better our image reconstructions will be, and the less data we will need to answer any given question. The Gaussian models which lead to the optimal linear filter are spectacularly poorly matched to the real world. For example, the Gaussian prior (46.3) fails to specify that all pixel intensities in an image are positive. This omission leads to the most pronounced artefacts where the image under observation has high contrast or large black patches. Optimal linear filters applied to astronomical data give reconstructions with negative areas in them, corresponding to patches of sky that suck energy out of telescopes! The maximum entropy model for image deconvolution (Gull and Daniell, 1978) was a great success principally because this model forced the reconstructed image to be positive. The spurious negative areas and complementary spurious positive areas are eliminated, and the quality of the reconstruction is greatly enhanced.

The 'classic maximum entropy' model assigns an entropic prior

\[
P(f \mid \alpha, m, \mathcal{H}_{\text{Classic}}) = \exp(\alpha S(f, m)) / Z,
\]

where

\[
S(f, m) = \sum_i (f_i \ln(m_i / f_i) + f_i - m_i)
\]

(46.13, 46.14) (Skilling, 1989). This model enforces positivity; the parameter \( \alpha \) defines a characteristic dynamic range by which the pixel values are expected to differ from the default image \( m \).

The 'intrinsic-correlation-function maximum-entropy' model (Gull, 1989) introduces an expectation of spatial correlations into the prior on \( f \) by writing \( f = G h \), where \( G \) is a convolution with an intrinsic correlation function, and putting a classic maxent prior on the underlying hidden image \( h \).

**Probabilistic movies**

Having found not only the most probable image \( f_{\text{MP}} \) but also error bars on it, \( \Sigma_{f_{\text{MP}}} \), one task is to visualize those error bars. Whether or not we use Monte Carlo methods to infer \( f \), a correlated random walk around the posterior distribution can be used to visualize the uncertainties and correlations. For a Gaussian posterior distribution, we can create a correlated sequence of unit normal random vectors \( n \) using

\[
n^{(t+1)} = cn^{(t)} + sz,
\]

(46.15)
where $z$ is a unit normal random vector and $c^2 + s^2 = 1$ ($c$ controls how persistent the memory of the sequence is). We then render the image sequence defined by
\[ f^{(t)} = f_{MIP} + \Sigma_{f_{id}}^{1/2} b^{(t)} \]
where $\Sigma_{f_{id}}^{1/2}$ is the Cholesky decomposition of $\Sigma_{f_{id}}$.

### 46.2 Supervised neural networks for image deconvolution

Neural network researchers often exploit the following strategy. Given a problem currently solved with a standard algorithm: interpret the computations performed by the algorithm as a parameterized mapping from an input to an output, and call this mapping a neural network; then adapt the parameters to data so as to produce another mapping that solves the task better. By construction, the neural network can reproduce the standard algorithm, so this data-driven adaptation can only make the performance better.

There are several reasons why standard algorithms can be bettered in this way.

1. Algorithms are often not designed to optimize the real objective function. For example, in speech recognition, a hidden Markov model is designed to model the speech signal, and is fitted so as to maximize the generative probability given the known string of words in the training data; but the real objective is to discriminate between different words. If an inadequate model is being used, the neural-net-style training of the model will focus the limited resources of the model on the aspects relevant to the discrimination task. Discriminative training of hidden Markov models for speech recognition does improve their performance.

2. The neural network can be more flexible than the standard model; some of the adaptive parameters might have been viewed as fixed features by the original designers. A flexible network can find properties in the data that were not included in the original model.

### 46.3 Deconvolution in humans

A huge fraction of our brain is devoted to vision. One of the neglected features of our visual system is that the raw image falling on the retina is severely blurred: while most people can see with a resolution of about 1 arcminute (one sixtieth of a degree) under any daylight conditions, bright or dim, the image on our retina is blurred through a point spread function of width as large as 5 arcminutes (Wald and Griffin, 1947; Howarth and Bradley, 1986). It is amazing that we are able to resolve pixels that are twenty-five times smaller in area than the blob produced on our retina by any point source.

Isaac Newton was aware of this conundrum. It’s hard to make a lens that does not have chromatic aberration, and our cornea and lens, like a lens made of ordinary glass, refract blue light more strongly than red. Typically our eyes focus correctly for the middle of the visible spectrum (green), so if we look at a single white dot made of red, green, and blue light, the image on our retina consists of a sharply focussed green dot surrounded by a broader red blob superposed on an even broader blue blob. The width of the red and blue blobs is proportional to the diameter of the pupil, which is largest under dim lighting conditions. [The blobs are roughly concentric, though most people have a slight bias, such that in one eye the red blob is centred a tiny distance
to the left and the blue is centred a tiny distance to the right, and in the other eye it’s the other way round. This slight bias explains why when we look at blue and red writing on a dark background most people perceive the blue writing to be at a slightly greater depth than the red. In a minority of people, this small bias is the other way round and the red/blue depth perception is reversed. But this effect (which many people are aware of, having noticed it in cinemas, for example) is tiny compared with the chromatic aberration we are discussing.]

You can vividly demonstrate to yourself how enormous the chromatic aberration in your eye is with the help of a sheet of card and a colour computer screen.

For the most impressive results – I guarantee you will be amazed – use a dim room with no light apart from the computer screen; a pretty strong effect will still be seen even if the room has daylight coming into it, as long as it is not bright sunshine. Cut a slit about 1.5 mm wide in the card. On the screen, display a few small coloured objects on a black background. I especially recommend thin vertical objects coloured pure red, pure blue, magenta (i.e., red plus blue), and white (red plus blue plus green). Include a little black-and-white text on the screen too. Stand or sit sufficiently far away that you can only just read the text – perhaps a distance of four metres or so, if you have normal vision. Now, hold the slit vertically in front of one of your eyes, and close the other eye. Hold the slit near to your eye – brushing your eyelashes – and look through it. Waggle the slit slowly to the left and to the right, so that the slit is alternately in front of the left and right sides of your pupil. What do you see? I see the red objects wagging to and fro, and the blue objects wagging to and fro, through huge distances and in opposite directions, while white objects appear to stay still and are negligibly distorted. Thin magenta objects can be seen splitting into their constituent red and blue parts. Measure how large the motion of the red and blue objects is – it’s more than 5 minutes of arc for me, in a dim room. Then check how sharply you can see under these conditions – look at the text on the screen, for example: is it not the case that you can see (through your whole pupil) features far smaller than the distance through which the red and blue components were wagging? Yet when you are using the whole pupil, what is falling on your retina must be an image blurred with a blurring diameter equal to the wagging amplitude.

One of the main functions of early visual processing must be to deconvolve this chromatic aberration. Neuroscientists sometimes conjecture that the reason why retinal ganglion cells and cells in the lateral geniculate nucleus (the main brain area to which retinal ganglion cells project) have centre-surround receptive fields with colour opponency (long wavelength in the centre and medium wavelength in the surround, for example) is in order to perform ‘feature extraction’ or ‘edge detection’, but I think this view is mistaken. The reason we have centre-surround filters at the first stage of visual processing (in the fovea at least) is for the huge task of deconvolution of chromatic aberration.

I speculate that the McCollough effect, an extremely long-lasting association of colours with orientation (McCollough, 1965; MacKay and MacKay, 1974), is produced by the adaptation mechanism that tunes our chromatic-aberration-deconvolution circuits. Our deconvolution circuits must be rapidly tuneable, because the point spread function of our eye changes with our pupil diameter, which can change within seconds; and indeed the McCollough effect can be induced within 30 seconds. At the same time, the effect is long-lasting when an eye is covered, because it’s in our interests that our deconvolution
circuits should stay well-tuned while we sleep, so that we can see sharply the instant we wake up.

I also wonder whether the main reason that we evolved colour vision was not ‘in order to see fruit better’ but ‘so as to be able to see black and white sharper’—deconvolving chromatic aberration is easier, even in an entirely black and white world, if one has access to chromatic information in the image.

And a final speculation: why do our eyes make micro-saccades when we look at things? These miniature eye-movements are of an angular size bigger than the spacing between the cones in the fovea (which are spaced at roughly 1 minute of arc, the perceived resolution of the eye). The typical size of a microsaccade is 5–10 minutes of arc (Ratliff and L.A., 1950). Is it a coincidence that this is the same as the size of chromatic aberration? Surely micro-saccades must play an essential role in the deconvolution mechanism that delivers our high-resolution vision.

46.4 Exercises

Exercise 46.1. Blur an image with a circular (top hat) point spread function and add noise. Then deconvolve the blurry noisy image using the optimal linear filter. Find error bars and visualize them by making a probabilistic movie.