39

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*.

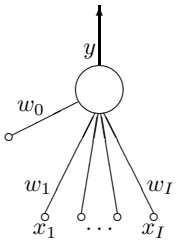


Figure 39.1. A single neuron

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.

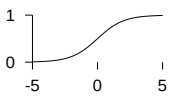
Architecture. A single neuron has a number I of *inputs* x_i and one *output* which we will here call y. (See figure 39.1.) Associated with each input is a *weight* w_i ($i = 1, \ldots, I$). There may be an additional parameter w_0 of the neuron called a *bias* which we may view as being the weight associated with an input x_0 that 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,
$$a = \sum_i w_i x_i, \tag{39.1}$$
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.

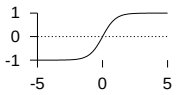
activation → activity
$a \quad \rightarrow \quad y(a)$

   (a) Deterministic activation functions:
      i. Linear.
$$y(a) = a. \tag{39.2}$$
      ii. Sigmoid (logistic function).
$$y(a) = \frac{1}{1 + e^{-a}} \qquad (y \in (0, 1)). \tag{39.3}$$



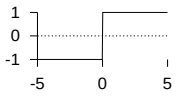
iii. Sigmoid (tanh).

$$y(a) = \tanh(a) \qquad (y \in (-1, 1)). \tag{39.4}$$



iv. Threshold function.

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



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

i. Heat bath.

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

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 $y(\mathbf{x}; \mathbf{w})$; the 'output' of the network, y, is a nonlinear function of the 'inputs' $\mathbf{x}$; this function is parameterized by 'weights' $\mathbf{w}$.

We will study a single neuron which produces an output between 0 and 1 as the following function of $\mathbf{x}$:

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

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

Motivations for the linear logistic function

In section 11.2 we studied 'the best detection of pulses', assuming that one of two signals $\mathbf{x}_0$ and $\mathbf{x}_1$ had been transmitted over a Gaussian channel with variance–covariance matrix $\mathbf{A}^{-1}$. We found that the probability that the source signal was $s = 1$ rather than $s = 0$, given the received signal $\mathbf{y}$, was

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

where $a(\mathbf{y})$ was a linear function of the received vector,

$$a(\mathbf{y}) = \mathbf{w}^{\mathsf{T}}\mathbf{y} + \theta, \tag{39.9}$$

with $\mathbf{w} \equiv \mathbf{A}(\mathbf{x}_1 - \mathbf{x}_0)$.

The linear logistic function can be motivated in several other ways – see the exercises.

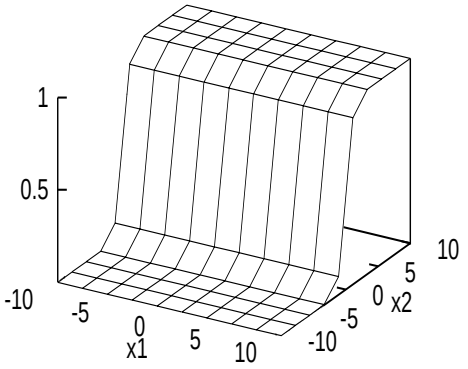


Figure 39.2. Output of a simple neural network as a function of its input.

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:

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2)}}. \tag{39.10}$$

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 $\mathbf{w}$.

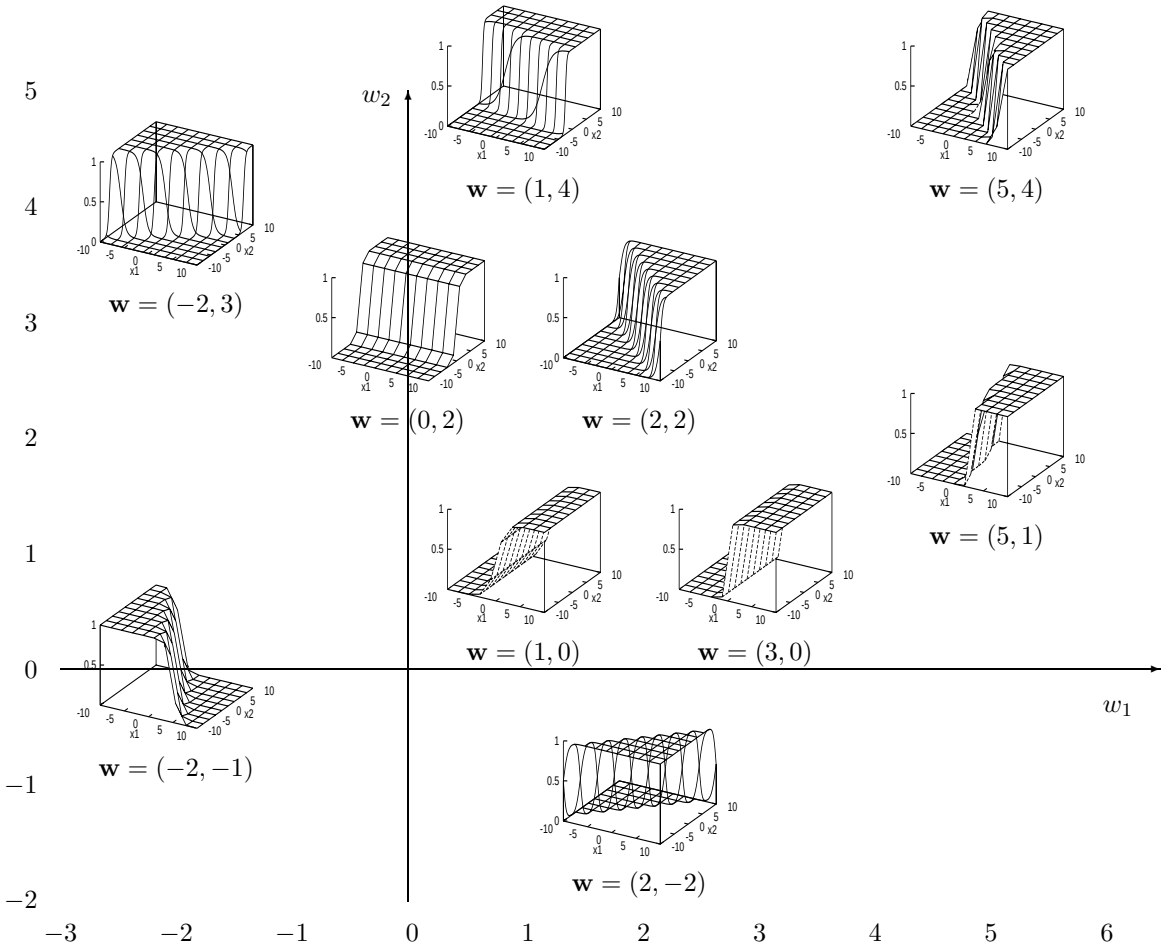


Figure 39.3. Weight space.

▶ 39.3 Training the single neuron as a binary classifier

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

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

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 $y(\mathbf{x}^{(n)};\mathbf{w}) = t^{(n)}$ for all n.

We now differentiate this objective function with respect to $\mathbf{w}$.

Exercise 39.2.[2] The backpropagation algorithm. Show that the derivative $\mathbf{g} = \partial G/\partial \mathbf{w}$ is given by:

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

Notice that the quantity $e^{(n)} \equiv t^{(n)} - y^{(n)}$ 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/parameter], whereas a change in a parameter has dimensions [parameter]). Since the derivative $\partial G/\partial \mathbf{w}$ is a sum of terms $\mathbf{g}^{(n)}$ defined by

$$g_j^{(n)} \equiv -(t^{(n)} - y^{(n)}) x_j^{(n)} \quad (39.13)$$

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 $\mathbf{w}$ a little in a direction opposite to $\mathbf{g}^{(n)}$.

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 $\mathbf{x}$ (which may be arbitrary real numbers), we compute the *activation* of the neuron,

$$a = \sum_i w_i x_i, \quad (39.14)$$

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.

$$y(a) = \frac{1}{1+e^{-a}}. \quad (39.15)$$

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*

$$e = t - y \tag{39.16}$$

then adjust the weights $\mathbf{w}$ in a direction that would reduce the magnitude of this error:

$$\Delta w_i = \eta e x_i, \tag{39.17}$$

where η is the 'learning rate'. Commonly η is set by trial and error to a constant value or to a decreasing function of simulation time τ such as η_0/τ.

The activity rule and learning rule are repeated for each input/target pair $(\mathbf{x}, t)$ 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 $(\mathbf{x}^{(n)}, t^{(n)})$ $(n = 1, \ldots, N)$, compute $y^{(n)} = y(\mathbf{x}^{(n)}; \mathbf{w})$, where

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + \exp(-\sum_i w_i x_i)}, \tag{39.18}$$

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

$$g_i^{(n)} = -e^{(n)} x_i^{(n)}. \tag{39.19}$$

Then let

$$\Delta w_i = -\eta \sum_n g_i^{(n)}. \tag{39.20}$$

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

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}. \tag{39.21}$$

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.

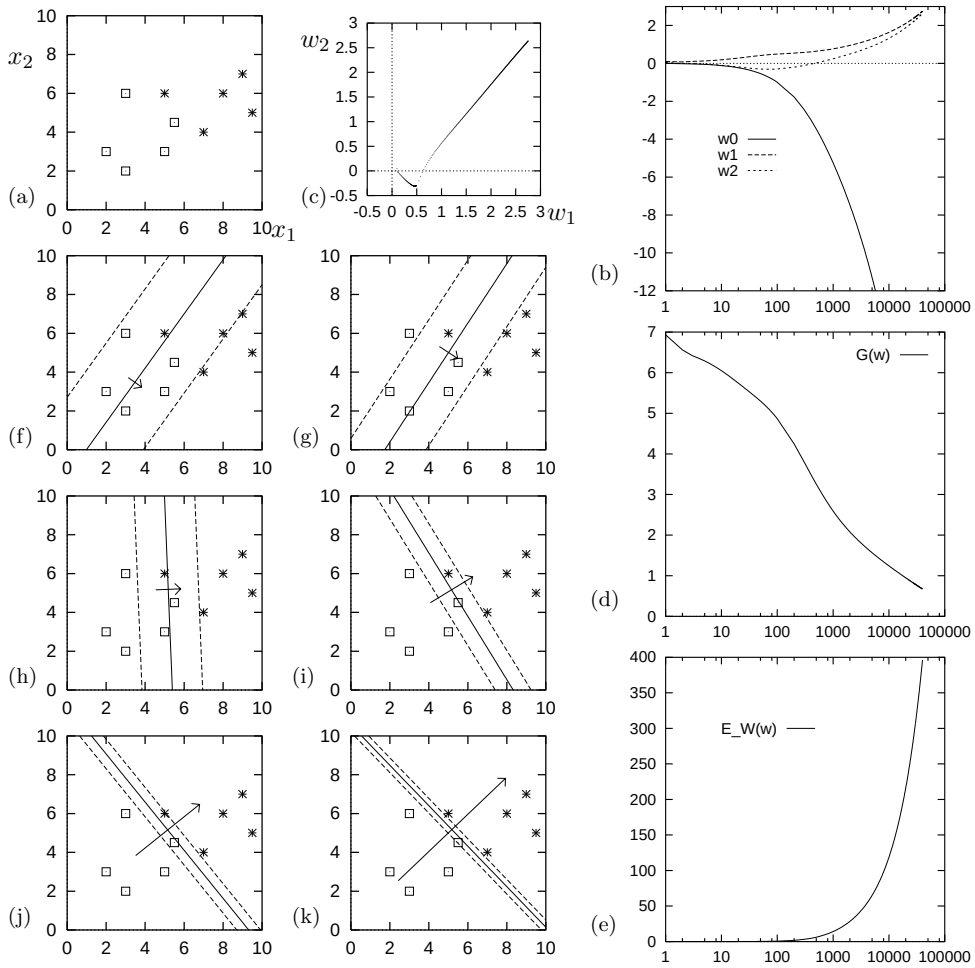


Figure 39.4. A single neuron learning to classify by gradient descent. The neuron has two weights w_1 and w_2 and a bias w_0. The learning rate was set to $\eta = 0.01$ and batch-mode gradient descent was performed using the code displayed in algorithm 39.5. (a) The training data. (b) Evolution of weights 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 $G(\mathbf{w})$ as a function of number of iterations. (e) The magnitude of the weights $E_W(\mathbf{w})$ as a function of time. (f–k) The function performed by the neuron (shown by three of its contours) after 30, 80, 500, 3000, 10 000 and 40 000 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.

```
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
```

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 * 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 = sigmoid(a)` computes the sigmoid function of all elements of the vector `a`.

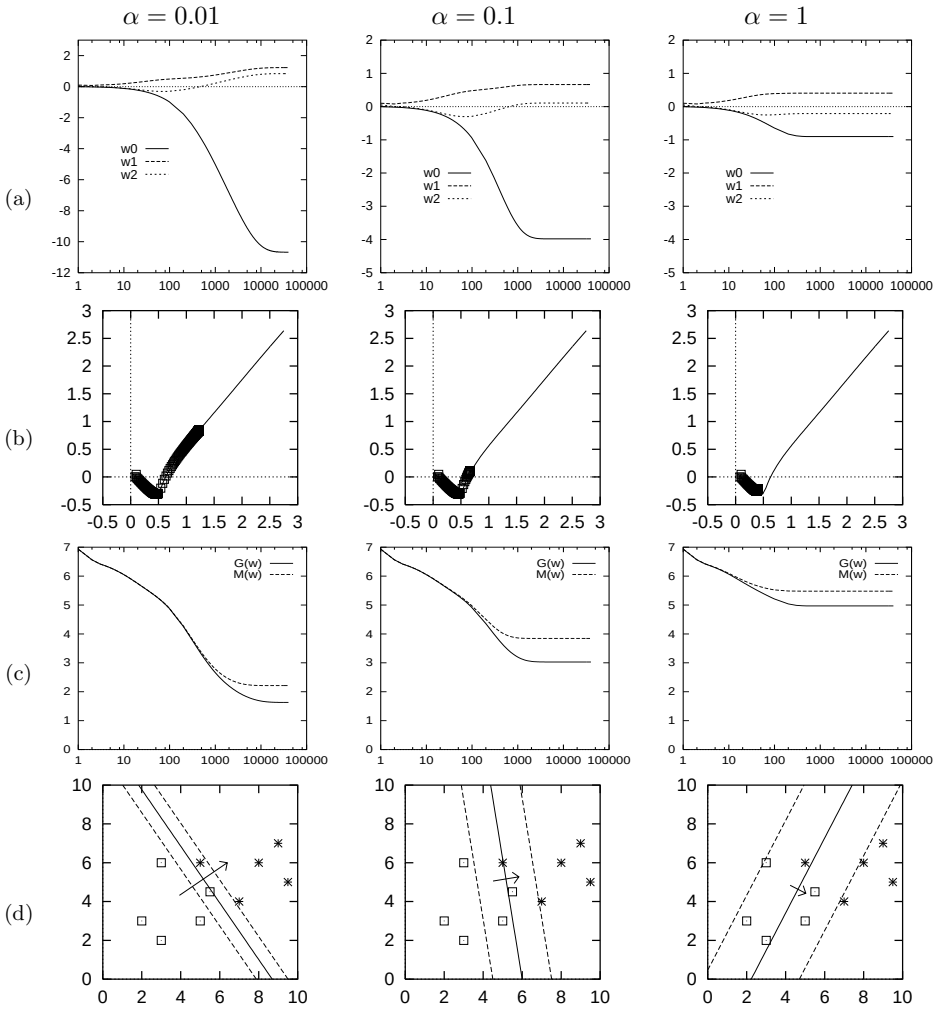


Figure 39.6. The influence of weight decay on a single neuron's learning. The objective function is $M(\mathbf{w}) = G(\mathbf{w}) + \alpha E_W(\mathbf{w})$. The learning method was as in figure 39.4. (a) Evolution of weights w_0, w_1 and w_2. (b) Evolution of weights w_1 and w_2 in weight space shown 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 $M(\mathbf{w})$ and the error function $G(\mathbf{w})$ as a function of number of iterations. (d) The function performed by the neuron after 40 000 iterations.

▶ 39.4 Beyond descent on the error function: regularization

If the parameter η is set to an appropriate value, this algorithm works: the algorithm finds a setting of $\mathbf{w}$ that 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 $G(\mathbf{w})$, 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 $\mathbf{w}$ 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:

$$M(\mathbf{w}) = G(\mathbf{w}) + \alpha E_W(\mathbf{w}) \tag{39.22}$$

where the simplest choice of regularizer is the *weight decay* regularizer

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

The *regularization constant* α is called the weight decay rate. This additional term favours small values of $\mathbf{w}$ and decreases the tendency of a model to overfit fine details of the training data. The quantity α 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.[1] Compute the derivative of $M(\mathbf{w})$ 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 η 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 α, with weight decay.]

▶ 39.5 Further exercises

More motivations for the linear neuron

▷ Exercise 39.4.[2] Consider the task of recognizing which of two Gaussian distributions a vector $\mathbf{z}$ 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 $\mathbf{z}$ given s ($s \in \{0,1\}$) be

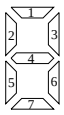
$$P(\mathbf{z} \mid s) = \prod_{i=1}^{I} \text{Normal}(z_i; 0, \sigma_{si}^2), \tag{39.24}$$

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

$$P(s=1 \mid \mathbf{z}) = \frac{1}{1+\exp(-\mathbf{w}^{\mathsf{T}}\mathbf{x}+\theta)}, \tag{39.25}$$

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

Exercise 39.5.[2] **The noisy LED**.



$\mathbf{c}(2) =$  $\mathbf{c}(3) =$  $\mathbf{c}(8) =$ 

Consider an LED display with 7 elements numbered as shown above. The state of the display is a vector $\mathbf{x}$. 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 $\mathbf{x}$? Show that $P(s=2 \mid \mathbf{x})$ can be written in the form

$$P(s=2 \mid \mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^{\mathsf{T}}\mathbf{x}+\theta)}, \tag{39.26}$$

and compute the values of the weights $\mathbf{w}$ 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 $\mathbf{x}$? Put your answer in the form

$$P(s \mid \mathbf{x}) = \frac{e^{a_s}}{\sum_{s'} e^{a_{s'}}}, \tag{39.27}$$

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

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

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

Table 39.7. An alternative 15-character alphabet for the 7-element LED display.

▷ Exercise 39.6.[2] A $(3,1)$ error-correcting code consists of the two codewords $\mathbf{x}^{(1)} = (1,0,0)$ and $\mathbf{x}^{(2)} = (0,0,1)$. 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

received vector is $\mathbf{r}$. Show that the posterior probability of s given $\mathbf{r}$ can be written in the form

$$P(s=1 \,|\, \mathbf{r}) = \frac{1}{1 + \exp\left(-w_0 - \sum_{n=1}^{3} w_n r_n\right)},$$

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

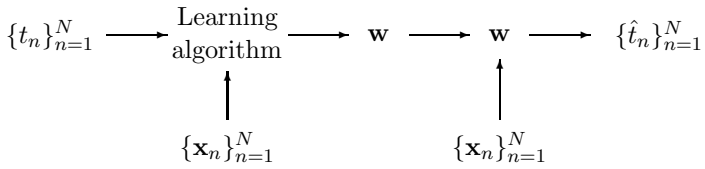


Figure 40.1. Neural network learning viewed as communication.

▶ 40.1 Neural network learning as communication

Many neural network models involve the adaptation of a set of weights $\mathbf{w}$ 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 $\{\mathbf{x}_n\}_{n=1}^N$. 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 $\mathbf{w}$ that depends on those data. The receiver then uses $\mathbf{w}$; 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 $\mathbf{x}_{N+1}$.] Just as a disk drive is a communication channel, the adapted network weights $\mathbf{w}$ 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 $2K$ 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 $\{\mathbf{x}_n\}$ that were in the training set. What matters now is how many different distinguishable functions our neuron can produce, given that we can observe the function only 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 $\mathbf{w}$ that can reproduce this labelling exactly. We will assume the algorithm finds such a $\mathbf{w}$ 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 $\{\mathbf{x}_n\}$ are we considering? Our answer might depend on this choice. We will assume that the points are in *general position*.

Definition 40.1 *A set of points $\{\mathbf{x}_n\}$ in K-dimensional space are in* general position *if any subset of size $\leq K$ is linearly independent, and no $K+1$ of them lie in a $(K-1)$-dimensional plane.*

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

$$y = f\left(\sum_{k=1}^{K} w_k x_k\right) \tag{40.1}$$

where

$$f(a) = \begin{cases} 1 & a > 0 \\ 0 & a \leq 0. \end{cases} \tag{40.2}$$

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.)

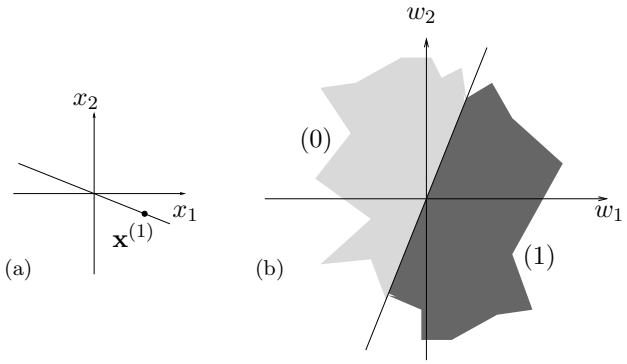


Figure 40.2. One data point in a two-dimensional input space, and the two regions of weight space that give the two alternative labellings of that point.

▶ 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 $\mathbf{x}^{(1)}$ then we can realize both possible labellings by setting $\mathbf{w} = \pm\mathbf{x}^{(1)}$. 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 $\mathbf{x} = (\pm 1, \pm 1)$. [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

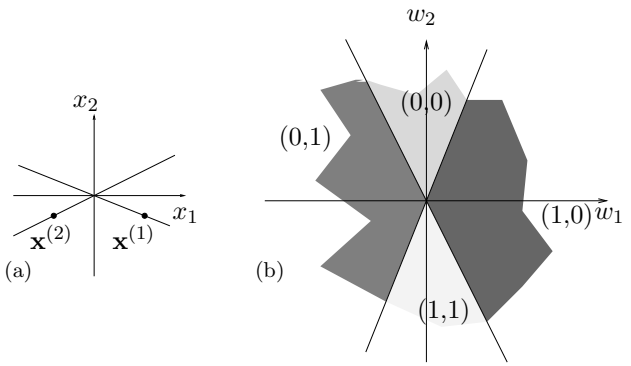


Figure 40.3. Two data points in a two-dimensional input space, and the four regions of weight space that give the four alternative labellings.

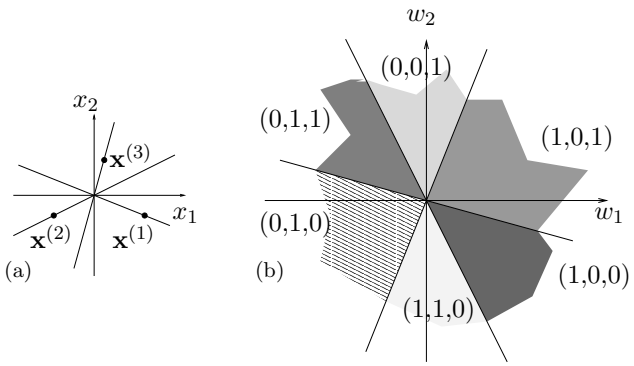


Figure 40.4. Three data points in a two-dimensional input space, and the six regions of weight space that give alternative labellings of those points. In this case, the labellings $(0,0,0)$ and $(1,1,1)$ cannot be realized. For any three points in general position there are always two labellings that cannot be realized.

space that classify a particular example $\mathbf{x}^{(n)}$ as a 1. For example, figure 40.2a shows a single point in our two-dimensional $\mathbf{x}$-space, and figure 40.2b shows the two corresponding sets of points in $\mathbf{w}$-space. One set of weight vectors occupy the half space

$$\mathbf{x}^{(n)} \cdot \mathbf{w} > 0, \tag{40.3}$$

and the others occupy $\mathbf{x}^{(n)} \cdot \mathbf{w} < 0$. 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 $\mathbf{x}^{(1)} \cdot \mathbf{w} = 0$ and $\mathbf{x}^{(2)} \cdot \mathbf{w} = 0$ 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 $\mathbf{x}^{(1)} \cdot \mathbf{w} = 0$ and $\mathbf{x}^{(2)} \cdot \mathbf{w} = 0$ into four regions; in any one region all vectors $\mathbf{w}$ 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 $\mathbf{w}$ 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 $\mathbf{w}$ 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

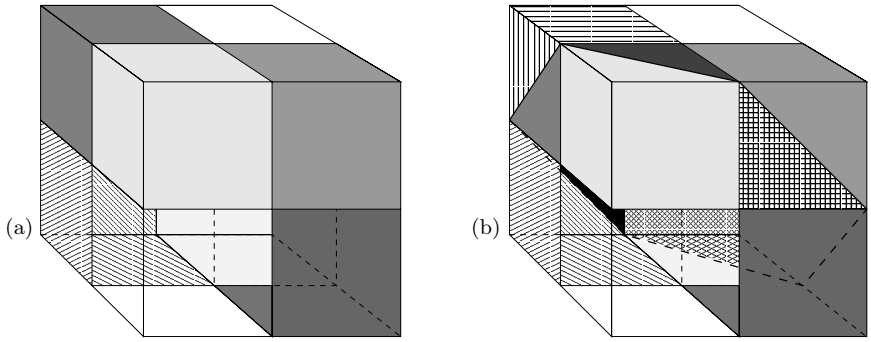


Figure 40.5. Weight space illustrations for $T(3,3)$ and $T(4,3)$. (a) $T(3,3) = 8$. Three hyperplanes (corresponding to three points in general position) divide 3-space into 8 regions, shown here by colouring the relevant part of the surface of a hollow, semi-transparent cube centred on the origin. (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.

	K							
N	1	2	3	4	5	6	7	8
1	2	2	2	2	2	2	2	2
2	2	4	4					
3	2	6	8					
4	2	8	14					
5	2	10						
6	2	12						

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

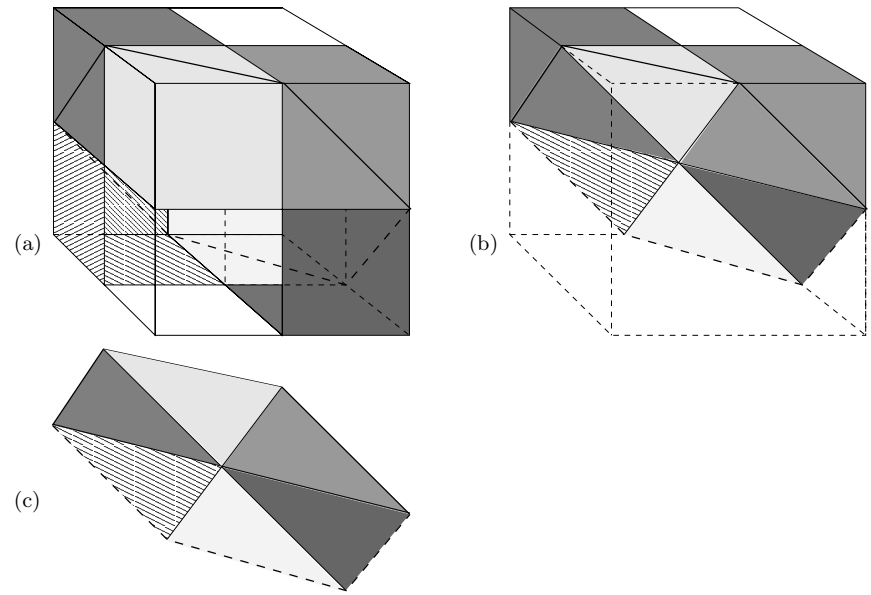


Figure 40.7. Illustration of the cutting process going from $T(3,3)$ to $T(4,3)$. (a) The eight regions of figure 40.5a with one added hyperplane. All of the regions that are not coloured white have been cut into two. (b) Here, the hollow cube has been made solid, so we can see which regions are cut 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 Nth 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

$$T(4, 3) = T(3, 3) + T(3, 2). \tag{40.4}$$

Recurrence relation for any N, K

Generalizing this picture, we see that when we add an Nth 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 Nth 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 Nth hyperplane, which gives the following equation for $T(N, K)$:

$$T(N, K) = T(N-1, K) + T(N-1, K-1). \tag{40.5}$$

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

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

Table 40.8. Pascal's triangle.

	K							
N	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		

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

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

[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.

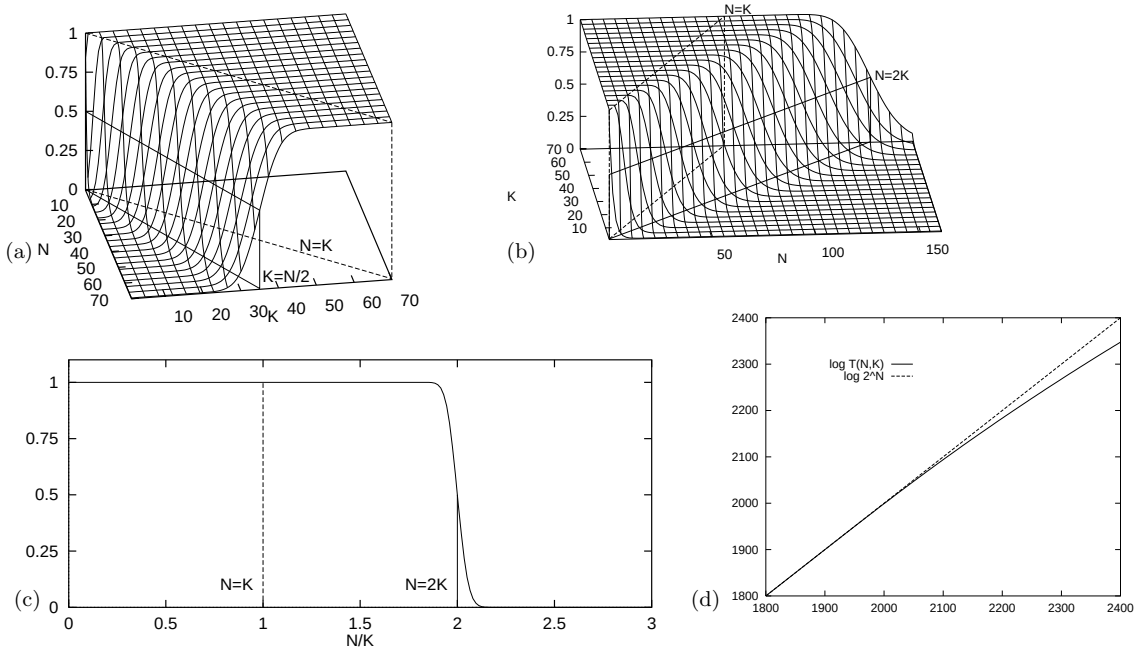


Figure 40.9. The fraction of functions on N points in K dimensions that are linear threshold functions, $T(N,K)/2^N$, shown from various viewpoints. In (a) we see the dependence on K, which is approximately an error function passing through 0.5 at $K = N/2$; the fraction reaches 1 at $K = N$. In (b) 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 $C_{\alpha,\beta}(N,K) \equiv \binom{N+\alpha}{K+\beta}$. 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:

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

Using the fact that the Nth 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 $K-1 \geq N-1$.

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

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,

$$\sum_{0}^{K} \binom{N}{k} \simeq 2^N \, \Phi\left(\frac{K-(N/2)}{\sqrt{N}/2}\right), \tag{40.10}$$

where $\Phi(z) \equiv \int_{-\infty}^{z} \exp(-z^2/2)/\sqrt{2\pi}$. Figure 40.9 shows the realizable 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.[2] 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.[2, p.491] 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.[2] Consider the binary threshold neuron in $K = 3$ dimensions, and the set of points $\{\mathbf{x}\} = \{(1,0,0),(0,1,0),(0,0,1),(1,1,1)\}$. Find a parameter vector $\mathbf{w}$ 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.[3] 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?

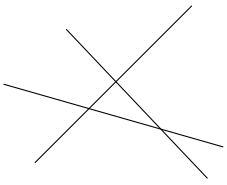


Figure 40.10. Three lines in a plane create seven regions.

Exercise 40.8.[2] 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.[3] 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

$$T(4,3)/2^4 = 14/16 = 7/8. \tag{40.11}$$

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

$$M(\mathbf{w}) = G(\mathbf{w}) + \alpha E_W(\mathbf{w}) \tag{41.1}$$

made up of an error function

$$G(\mathbf{w}) = -\sum_n \left[t^{(n)} \ln y(\mathbf{x}^{(n)}; \mathbf{w}) + (1 - t^{(n)}) \ln(1 - y(\mathbf{x}^{(n)}; \mathbf{w})) \right] \tag{41.2}$$

and a regularizer

$$E_W(\mathbf{w}) = \frac{1}{2} \sum_i w_i^2. \tag{41.3}$$

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

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

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

$$\begin{array}{rcl} P(t = 1 \,|\, \mathbf{w}, \mathbf{x}) & = & y \\ P(t = 0 \,|\, \mathbf{w}, \mathbf{x}) & = & 1 - y \end{array} \tag{41.4}$$

can be rewritten as the single equation

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

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

$$P(D \,|\, \mathbf{w}) = \exp[-G(\mathbf{w})]. \tag{41.6}$$

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

$$P(\mathbf{w} \,|\, \alpha) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W). \tag{41.7}$$

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 $\mathbf{w}$.

The objective function $M(\mathbf{w})$ then corresponds to the *inference* of the parameters $\mathbf{w}$, given the data:

$$P(\mathbf{w} \mid D, \alpha) = \frac{P(D \mid \mathbf{w})P(\mathbf{w} \mid \alpha)}{P(D \mid \alpha)} \tag{41.8}$$

$$= \frac{e^{-G(\mathbf{w})}\, e^{-\alpha E_W(\mathbf{w})}/Z_W(\alpha)}{P(D \mid \alpha)} \tag{41.9}$$

$$= \frac{1}{Z_M} \exp(-M(\mathbf{w})). \tag{41.10}$$

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

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 $M(\mathbf{w})$ 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,

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2)}}, \tag{41.11}$$

we can plot the posterior probability of $\mathbf{w}$, $P(\mathbf{w} \mid D, \alpha) \propto \exp(-M(\mathbf{w}))$. 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 $\mathbf{x}$ and a t value indicated by $\times$ $(t = 1)$ or $\Box$ $(t = 0)$. The likelihood function $\exp(-G(\mathbf{w}))$ is shown as a function of $\mathbf{w}$ in the second column. It is a product of functions of the form (41.11).

The product of traditional learning is a point in $\mathbf{w}$-space, the estimator $\mathbf{w}^*$, 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 $\mathbf{w}$; rather we evaluate their posterior probabilities. The posterior distribution is obtained by multiplying the likelihood by a prior distribution over $\mathbf{w}$ 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 $\mathbf{w}^*$.

▶ 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.

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}. \tag{41.12}$$

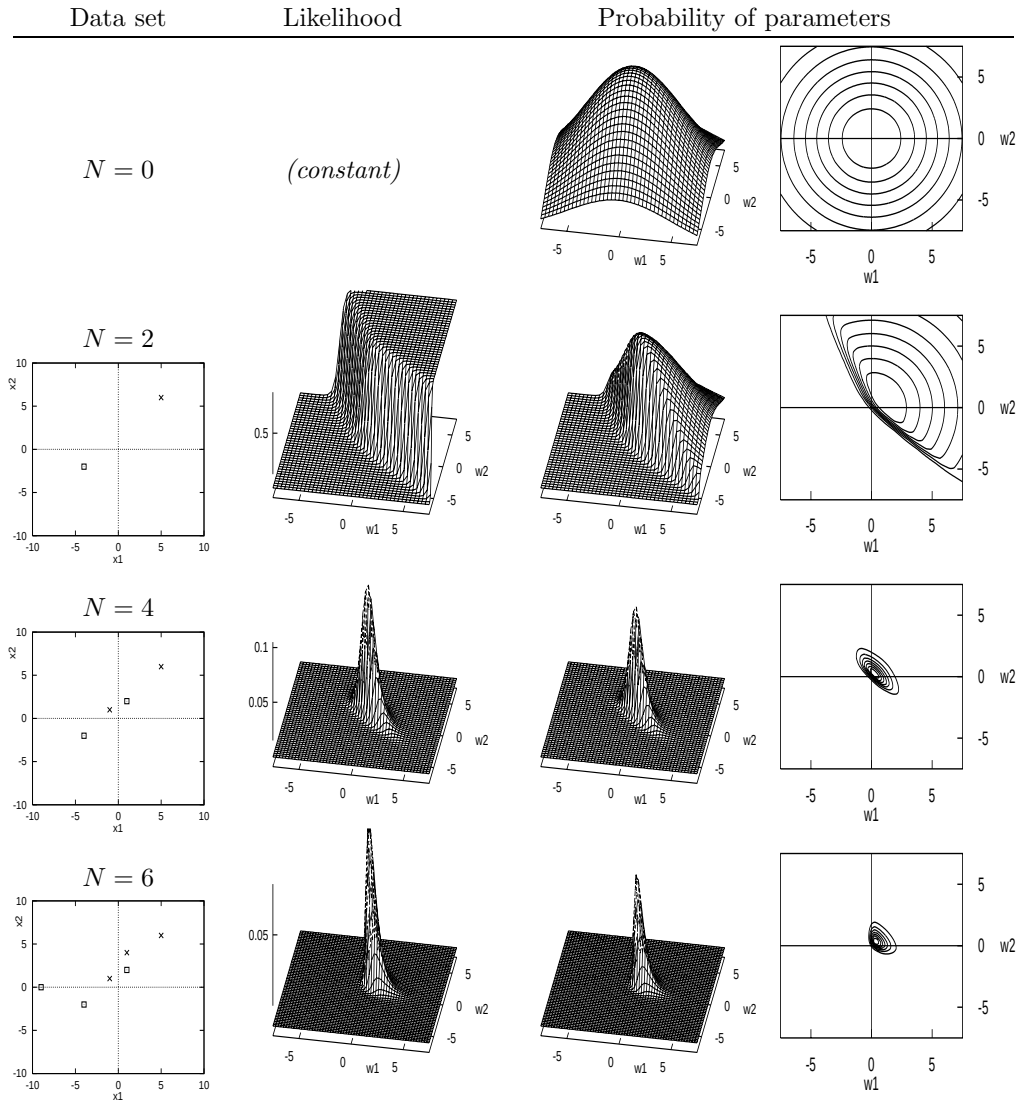


Figure 41.1. The Bayesian interpretation and generalization of traditional neural network learning. Evolution of the probability distribution over parameters as data arrive.

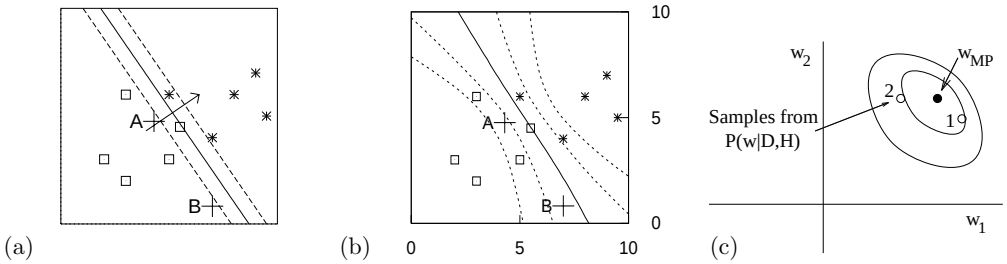


Figure 41.2. Making predictions. (a) The function performed by an optimized neuron $\mathbf{w}_{\rm MP}$ (shown by three of its contours) trained with weight decay, $\alpha = 0.01$ (from figure 39.6). The contours shown are those corresponding to $a = 0, \pm 1$, namely $y = 0.5, 0.27$ and 0.73. (b) Are 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 $\mathbf{w}$ (schematic); the Bayesian predictions shown in (b) were obtained by averaging together the predictions made by each possible value of the weights $\mathbf{w}$, with each value of $\mathbf{w}$ 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

$$M(\mathbf{w}) = G(\mathbf{w}) + \alpha E(\mathbf{w}). \tag{41.13}$$

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 $\mathbf{w}_{\rm MP}$, 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 $\mathbf{w}_{\rm MP} \cdot \mathbf{x}$. If we really knew that $\mathbf{w}$ was equal to $\mathbf{w}_{\rm MP}$, then these predictions would be correct. But we do not know $\mathbf{w}$. 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* $\mathbf{w}_{\rm MP}$ *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 $\mathbf{t}^{(N+1)}$ involves *marginalizing* over the parameters (and over anything else about which we are uncertain). For simplicity, let us assume that the weights $\mathbf{w}$ are the only uncertain quantities – the weight decay rate α and the model $\mathcal{H}$ itself are assumed to be fixed. Then by the sum rule, the predictive probability of a new target $\mathbf{t}^{(N+1)}$ at a location $\mathbf{x}^{(N+1)}$ is:

$$P(\mathbf{t}^{(N+1)} \,|\, \mathbf{x}^{(N+1)}, D, \alpha) = \int \mathrm{d}^K \mathbf{w}\, P(\mathbf{t}^{(N+1)} \,|\, \mathbf{x}^{(N+1)}, \mathbf{w}, \alpha) P(\mathbf{w} \,|\, D, \alpha), \tag{41.14}$$

where K is the dimensionality of $\mathbf{w}$, three in the toy problem. Thus the predictions are obtained by weighting the prediction for each possible $\mathbf{w}$,

$$\begin{array}{rcl} P(\mathbf{t}^{(N+1)} = 1 \,|\, \mathbf{x}^{(N+1)}, \mathbf{w}, \alpha) & = & y(\mathbf{x}^{(N+1)}; \mathbf{w}) \\ P(\mathbf{t}^{(N+1)} = 0 \,|\, \mathbf{x}^{(N+1)}, \mathbf{w}, \alpha) & = & 1 - y(\mathbf{x}^{(N+1)}; \mathbf{w}), \end{array} \tag{41.15}$$

with a weight given by the posterior probability of $\mathbf{w}$, $P(\mathbf{w} \,|\, D, \alpha)$, which we most recently wrote down in equation (41.10). This posterior probability is

$$P(\mathbf{w} \,|\, D, \alpha) = \frac{1}{Z_M} \exp(-M(\mathbf{w})), \tag{41.16}$$

where

$$Z_M = \int \mathrm{d}^K \mathbf{w} \exp(-M(\mathbf{w})). \tag{41.17}$$

In summary, we can get the Bayesian predictions if we can find a way of computing the integral

$$P(\mathbf{t}^{(N+1)} = 1 \,|\, \mathbf{x}^{(N+1)}, D, \alpha) = \int \mathrm{d}^K \mathbf{w}\ \ y(\mathbf{x}^{(N+1)}; \mathbf{w}) \frac{1}{Z_M} \exp(-M(\mathbf{w})), \tag{41.18}$$

which is the average of the output of the neuron at $\mathbf{x}^{(N+1)}$ under the posterior distribution of $\mathbf{w}$.

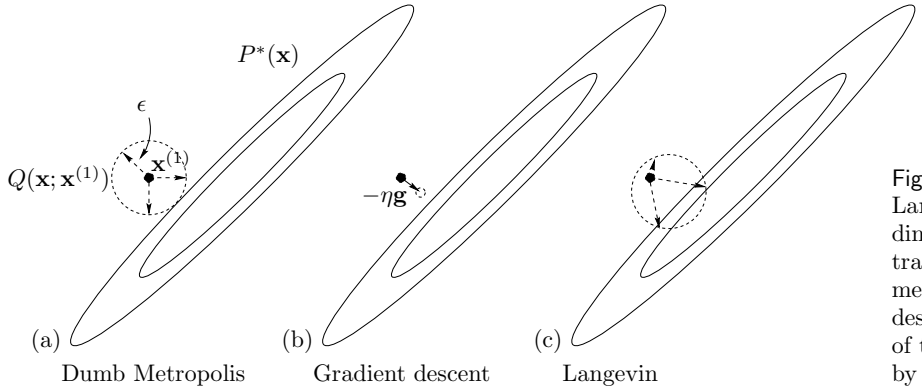


Figure 41.3. One step of the Langevin method in two dimensions (c), contrasted with a traditional 'dumb' Metropolis method (a) and with gradient descent (b). The proposal density of the Langevin method is given by 'gradient descent with noise'.

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 $P(\mathbf{w} \mid D, \alpha)$ 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 $y(\mathbf{x}^{(N+1)}; \mathbf{w})$ as a function f of $\mathbf{w}$ whose mean we compute using

$$\langle f(\mathbf{w}) \rangle \simeq \frac{1}{R} \sum_r f(\mathbf{w}^{(r)}) \tag{41.19}$$

where $\{\mathbf{w}^{(r)}\}$ are samples from the posterior distribution $\frac{1}{Z_M}\exp(-M(\mathbf{w}))$ (cf. 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 $P(t \mid D, \mathcal{H})$ 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 $\{\mathbf{w}^{(r)}\}$? 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 $\mathbf{p}$ is generated from a Gaussian with unit variance. The gradient $\mathbf{g}$ is computed,

```
g = gradM ( w ) ;                          # set gradient using initial w
M = findM ( w ) ;                          # set objective function too

for l = 1:L                                # loop L times
   p = randn ( size(w) ) ;                 # initial momentum is Normal(0,1)
   H = p' * p / 2 + M ;                    # evaluate H(w,p)

*  p = p - epsilon * g / 2 ;               # make half-step in p
*  wnew = w + epsilon * p ;                # make step in w
*  gnew = gradM ( wnew ) ;                 # find new gradient
*  p = p - epsilon * gnew / 2 ;            # make half-step in p

   Mnew = findM ( wnew ) ;                 # find new objective function
   Hnew = p' * p / 2 + Mnew ;              # evaluate new value of H
   dH = Hnew - H ;                         # decide whether to accept
   if ( dH < 0 )                 accept = 1 ;
   elseif ( rand() < exp(-dH) ) accept = 1 ;  # compare with a uniform
   else                          accept = 0 ;  #                variate
   endif
   if ( accept )      g = gnew ;   w = wnew ;    M = Mnew ;    endif
endfor

function gM = gradM ( w )                  # gradient of objective function
   a = x * w  ;                            # compute activations
   y = sigmoid(a) ;                        # compute outputs
   e = t - y   ;                           # compute errors
   g = - x' * e ;                          # compute the gradient of G(w)
   gM = alpha * w + g ;
endfunction

function M = findM ( w )                   # objective function
   G = - (t' * log(y) + (1-t') * log( 1-y )) ;
   EW = w' * w / 2 ;
   M = G + alpha * EW ;
endfunction
```

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).

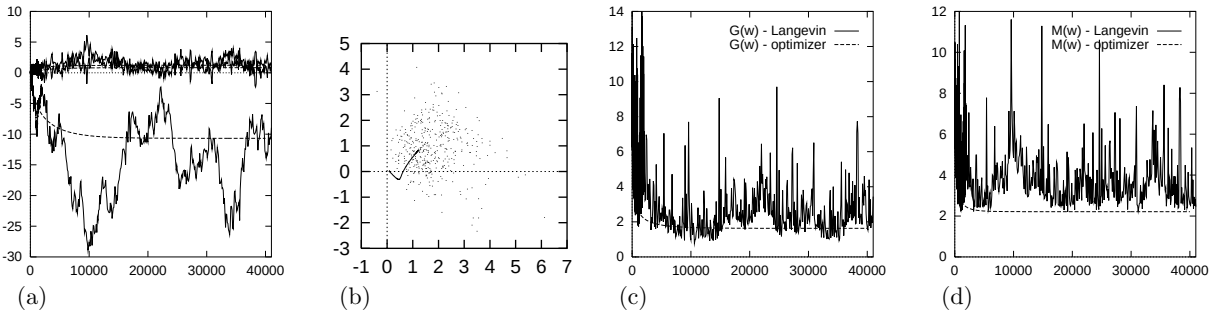


Figure 41.5. A single neuron learning under the Langevin Monte Carlo method. (a) Evolution of weights w_0, w_1 and w_2 as a function of number of iterations. (b) Evolution of weights w_1 and w_2 in weight space. Also shown by a line is the evolution of the weights using the optimizer of figure 39.6. (c) The error function $G(\mathbf{w})$ as a function of number of iterations. Also shown is the error function during the optimization of figure 39.6. (d) The objective function $M(\mathbf{x})$ as a function of number of iterations. See also figures 41.6 and 41.7.

and a step in $\mathbf{w}$ is made, given by

$$\Delta\mathbf{w} = -\tfrac{1}{2}\epsilon^2\mathbf{g} + \epsilon\mathbf{p}. \tag{41.20}$$

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

The Langevin method has one free parameter, ϵ, which controls the typical step size. If ϵ 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 $M(\mathbf{w}) = G(\mathbf{w}) + \alpha E_W(\mathbf{w})$, 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 $\mathbf{w}$ 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 10 000 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 ϵ had been used, but the value was chosen so that the 'descent rate' $\eta = \frac{1}{2}\epsilon^2$ 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 $\mathbf{x}$ 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 $\mathbf{x}$ 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.

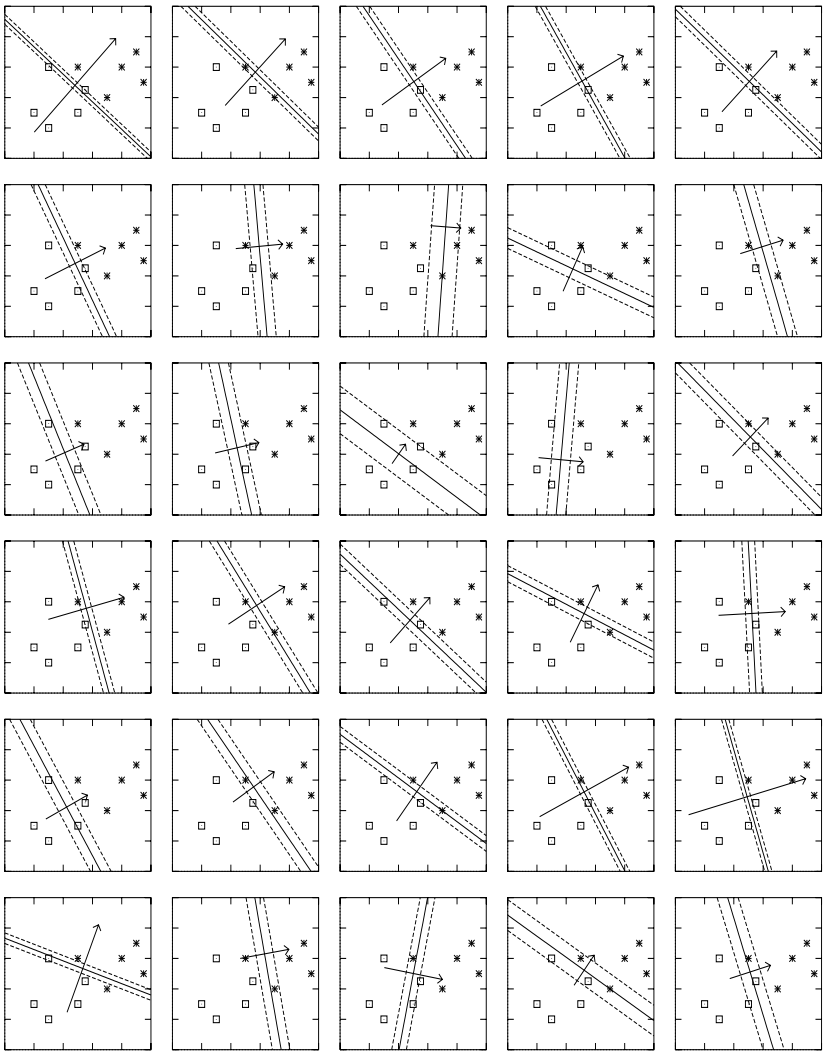


Figure 41.6. Samples obtained by the Langevin Monte Carlo method. The learning rate was set to $\eta = 0.01$ and the weight decay rate to $\alpha = 0.01$. The step size is given by $\epsilon = \sqrt{2\eta}$. The function performed by the neuron is shown by three of its contours every 1000 iterations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

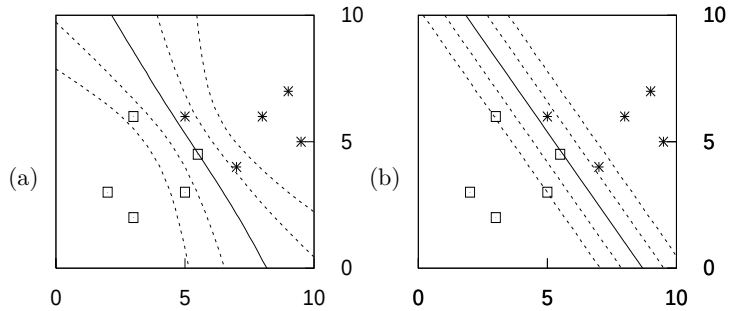


Figure 41.7. Bayesian predictions found by the Langevin Monte Carlo method compared with the predictions using the optimized parameters. (a) The predictive function obtained by averaging the predictions for 30 samples uniformly spaced between iterations 10 000 and 40 000, shown in figure 41.6. The contours 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 $M(\mathbf{w})$.

```
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.

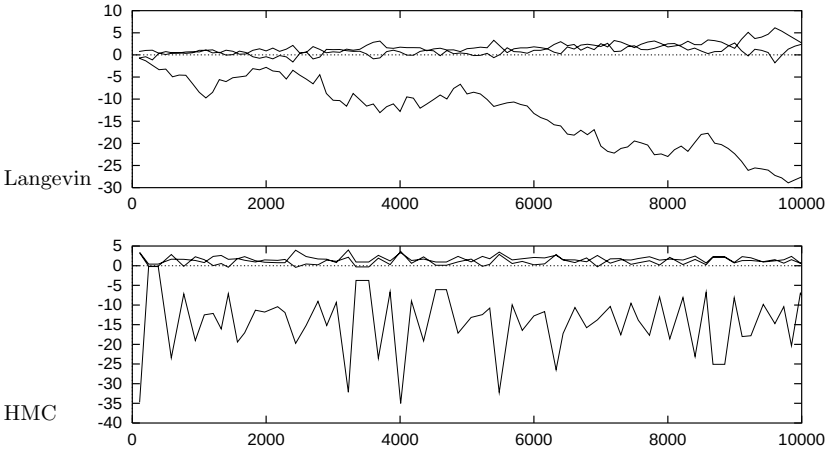


Figure 41.9. Comparison of sampling properties of the Langevin Monte Carlo method and the Hamiltonian Monte Carlo (HMC) method. The horizontal axis is the number of gradient evaluations made. Each figure shows the weights during the first 10,000 iterations. The rejection rate during this Hamiltonian Monte Carlo simulation was 8%.

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

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 overconfident.

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 $\mathbf{w}, \mathbf{p}$ space, where $\mathbf{p}$ are the extra 'momentum' variables of the Langevin and Hamiltonian Monte Carlo methods. The number of steps '`Tau`' was set at random to a number between 100 and 200 for each trajectory. The step size ϵ 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 offer an alternative strategy for dealing with the integral

$$P(\mathbf{t}^{(N+1)}=1 \,|\, \mathbf{x}^{(N+1)}, D, \alpha) = \int \mathrm{d}^K\mathbf{w}\; y(\mathbf{x}^{(N+1)};\mathbf{w}) \frac{1}{Z_M} \exp(-M(\mathbf{w})), \quad (41.21)$$

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 $M(\mathbf{w})$ (using a gradient-based optimizer) and Taylor-expand M there:

$$M(\mathbf{w}) \simeq M(\mathbf{w}_{\mathrm{MP}}) + \frac{1}{2}(\mathbf{w}-\mathbf{w}_{\mathrm{MP}})^{\mathsf{T}}\mathbf{A}(\mathbf{w}-\mathbf{w}_{\mathrm{MP}}) + \cdots, \quad (41.22)$$

where $\mathbf{A}$ is the matrix of second derivatives, also known as the *Hessian*, defined by

$$A_{ij} \equiv \frac{\partial^2}{\partial w_i \partial w_j} M(\mathbf{w}) \Big|_{\mathbf{w}=\mathbf{w}_{\mathrm{MP}}} . \quad (41.23)$$

We thus define our Gaussian approximation:

$$Q(\mathbf{w}; \mathbf{w}_{\mathrm{MP}}, \mathbf{A}) = [\det(\mathbf{A}/2\pi)]^{1/2} \exp\left[-\frac{1}{2}(\mathbf{w}-\mathbf{w}_{\mathrm{MP}})^{\mathsf{T}}\mathbf{A}(\mathbf{w}-\mathbf{w}_{\mathrm{MP}})\right]. \quad (41.24)$$

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

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

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

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

$$f'(a) = \frac{\mathrm{d}}{\mathrm{d}a} f(a) = f(a)(1-f(a)), \quad (41.26)$$

and

$$a^{(n)} = \sum_j w_j x_j^{(n)}. \quad (41.27)$$

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

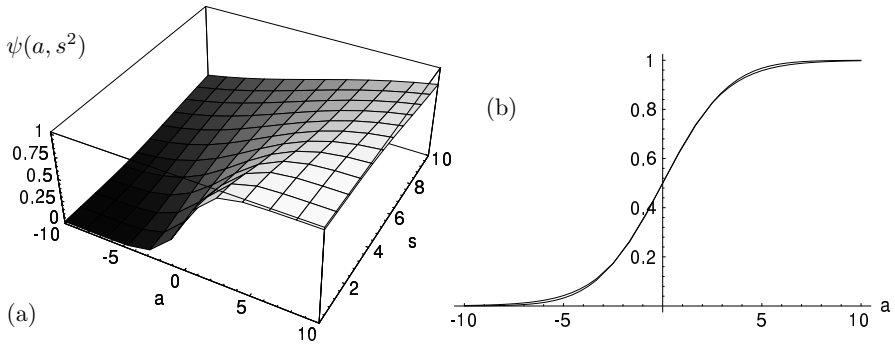


Figure 41.10. The marginalized probability, and an approximation to it. (a) The function $\psi(a, s^2)$, evaluated numerically. In (b) the functions $\psi(a, s^2)$ and $\phi(a, s^2)$ defined in the text are shown as a function of a for $s^2 = 4$. From MacKay (1992b).

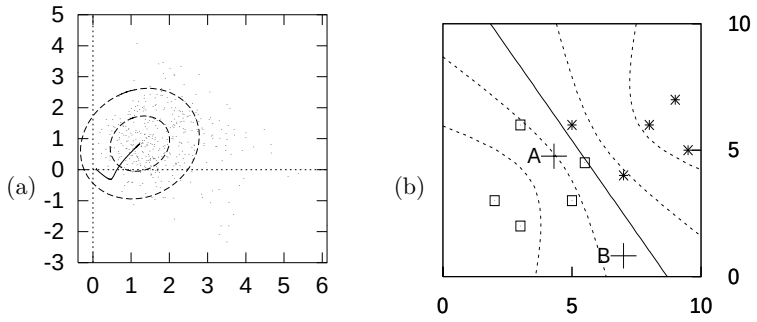


Figure 41.11. The Gaussian approximation in weight space and its approximate predictions in input space. (a) A projection of the Gaussian approximation onto the (w_1, w_2) plane of weight space. The one- and two-standard-deviation contours are shown. Also shown are the trajectory of the optimizer, and the Monte Carlo method's samples. (b) The predictive function obtained from the Gaussian approximation and equation (41.30). (cf. figure 42.2.)

Calculating the marginalized probability

The output $y(\mathbf{x}; \mathbf{w})$ depends on $\mathbf{w}$ only 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 $\mathbf{w} = \mathbf{w}_{\rm MP} + \Delta\mathbf{w}$, $P(\mathbf{w} \,|\, D, \alpha) \simeq (1/Z_Q) \exp(-\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbf{A}\Delta\mathbf{w})$. 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.[2] Assuming $\mathbf{w}$ is Gaussian-distributed with mean $\mathbf{w}_{\rm MP}$ and variance–covariance matrix $\mathbf{A}^{-1}$, show that the probability distribution of $a(\mathbf{x})$ is

$$P(a \,|\, \mathbf{x}, D, \alpha) = \text{Normal}(a_{\rm MP}, s^2) = \frac{1}{\sqrt{2\pi s^2}} \exp\left(-\frac{(a - a_{\rm MP})^2}{2s^2}\right), \tag{41.28}$$

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

This means that the marginalized output is:

$$P(t=1 \,|\, \mathbf{x}, D, \alpha) = \psi(a_{\rm MP}, s^2) \equiv \int \mathrm{d}a \, f(a) \, \text{Normal}(a_{\rm MP}, s^2). \tag{41.29}$$

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

$$\psi(a_{\rm MP}, s^2) \simeq \phi(a_{\rm MP}, s^2) \equiv f(\kappa(s) a_{\rm MP}) \tag{41.30}$$

with $\kappa = 1/\sqrt{1 + \pi s^2/8}$ (figure 41.10).

Demonstration

Figure 41.11 shows the result of fitting a Gaussian approximation at the optimum $\mathbf{w}_{\rm MP}$, and the results of using that Gaussian approximation and equa-

tion (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.[2] Is the Gaussian approximation to $P(\mathbf{w} \mid D, \alpha)$ too heavy-tailed or too light-tailed, or both? It may help to consider $P(\mathbf{w} \mid D, \alpha)$ 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 $a_{\mathrm{MP}} = 0$. 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.