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*.

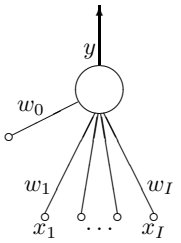


Figure 39.1. A single neuron

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.

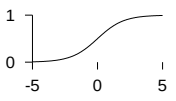
Architecture. A single neuron has a number I of *inputs* x_i and one *output* which we will here call y. (See figure 39.1.) Associated with each input is a *weight* w_i ($i = 1, \ldots, I$). There may be an additional parameter w_0 of the neuron called a *bias* which we may view as being the weight associated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

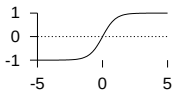
activation → activity
$a \quad \rightarrow \quad y(a)$

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



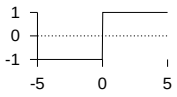
iii. Sigmoid (tanh).

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



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.

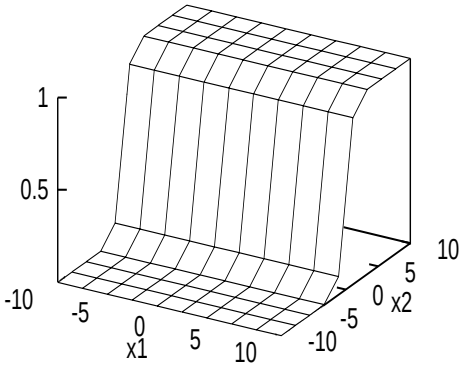


Figure 39.2. Output of a simple neural network as a function of its input.

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 $\mathbf{w}$.

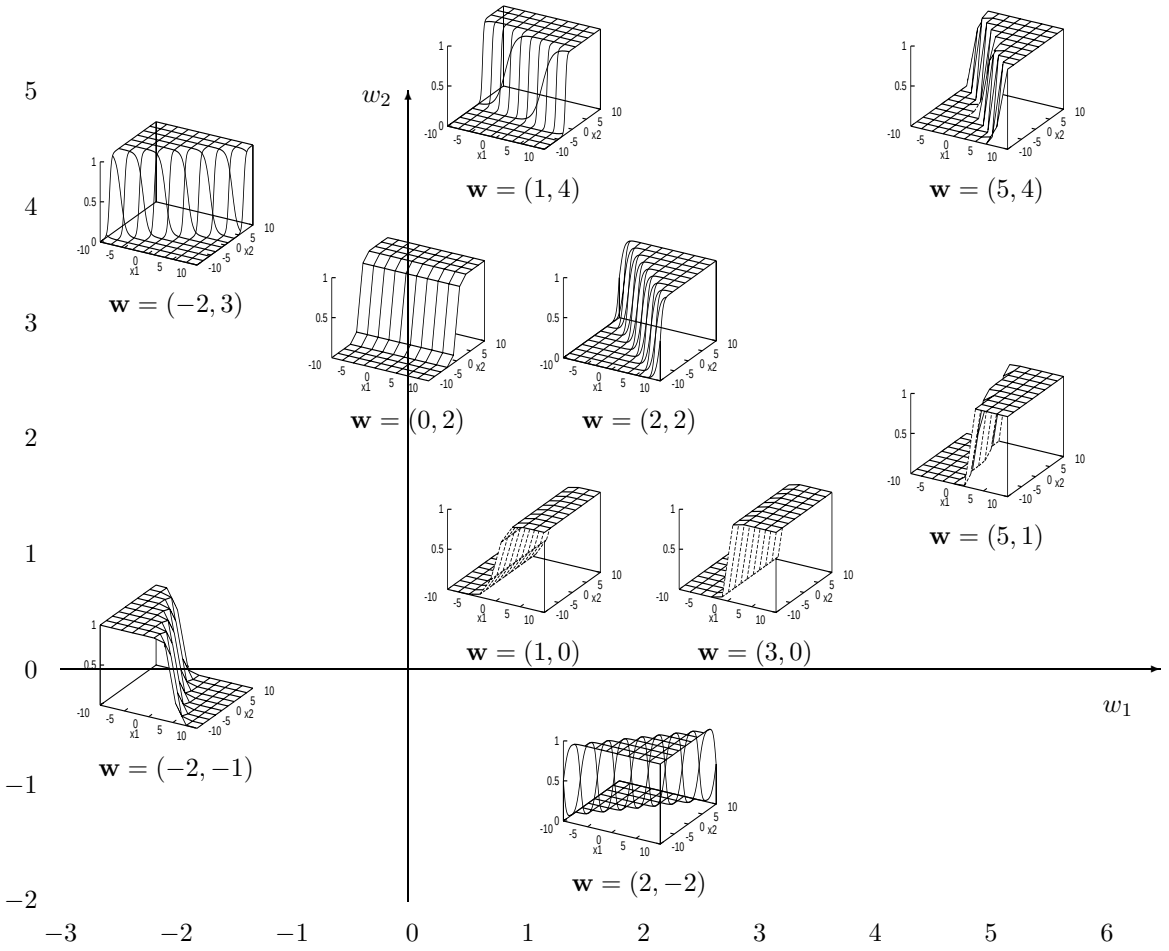


Figure 39.3. Weight space.

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.

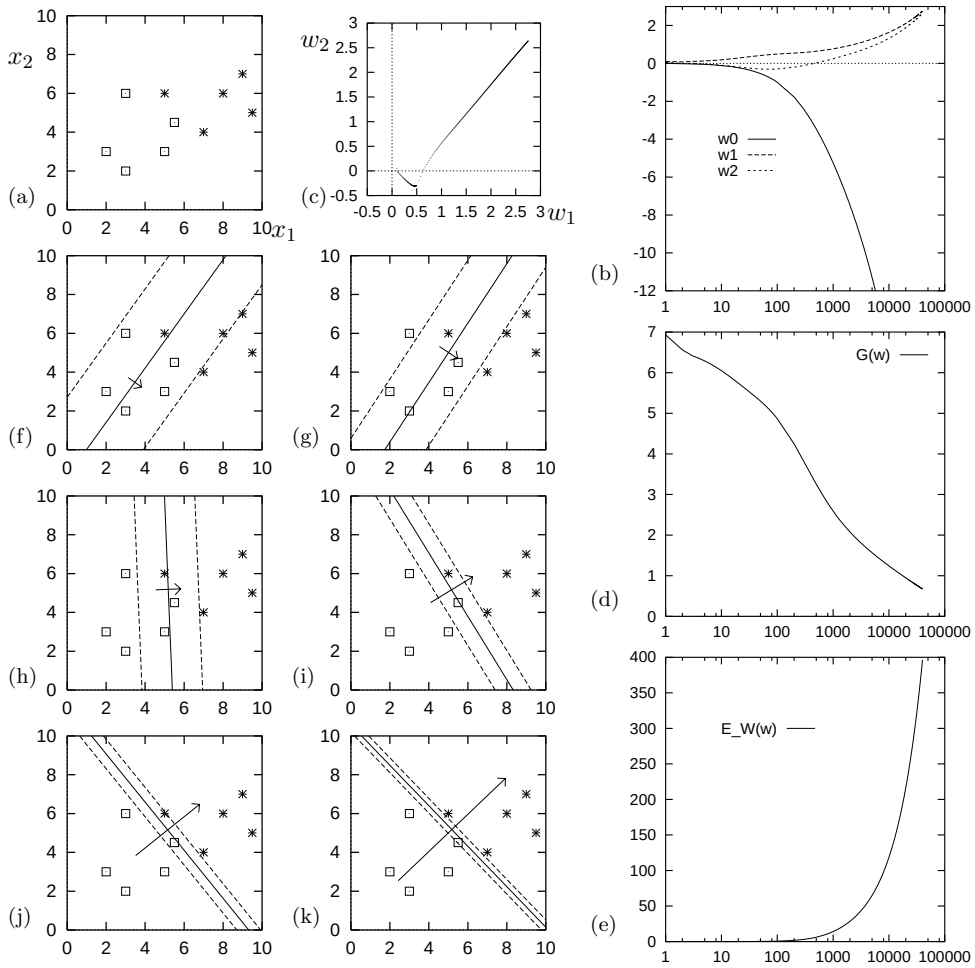


Figure 39.4. A single neuron learning to classify by gradient descent. The neuron has two weights w_1 and w_2 and a bias w_0. The learning rate was set to $\eta = 0.01$ and batch-mode gradient descent was performed using the code displayed in algorithm 39.5. (a) The training data. (b) Evolution of weights 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`.

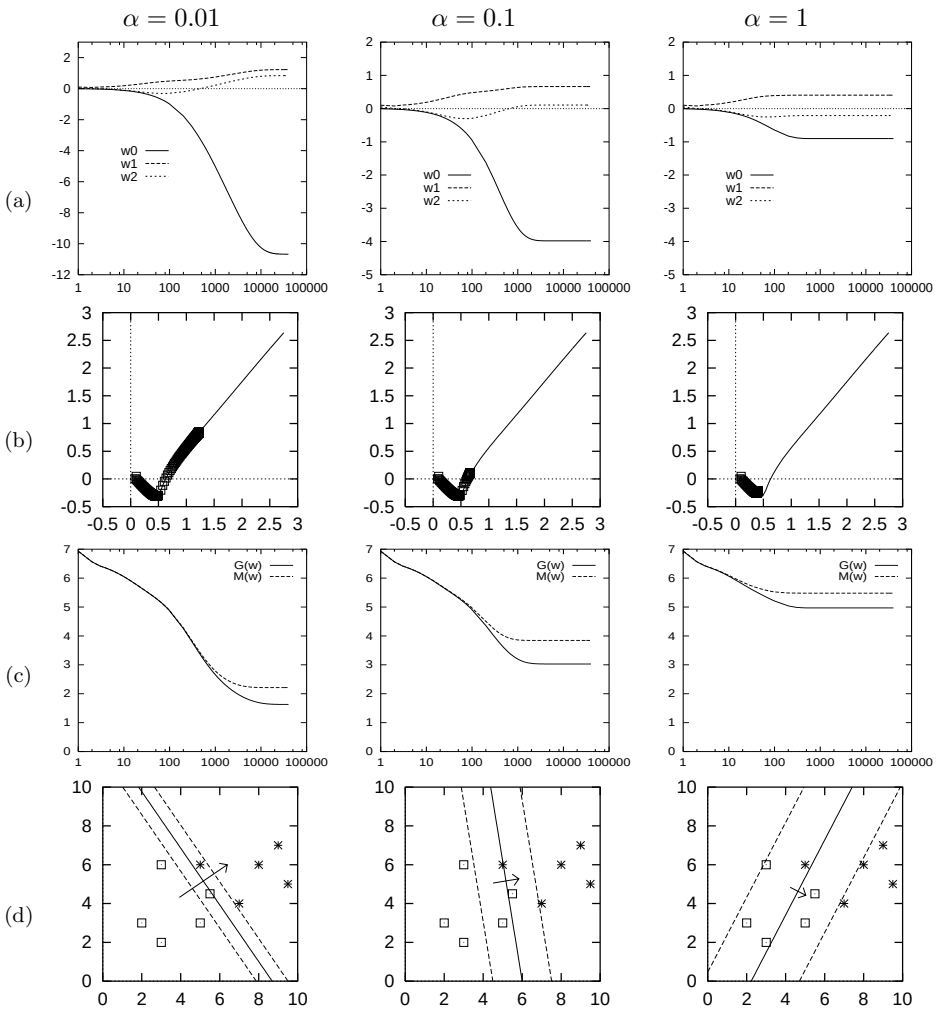


Figure 39.6. The influence of weight decay on a single neuron's learning. The objective function is $M(\mathbf{w}) = G(\mathbf{w}) + \alpha E_W(\mathbf{w})$. The learning method was as in figure 39.4. (a) Evolution of weights w_0, w_1 and w_2. (b) Evolution of weights w_1 and w_2 in weight space shown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

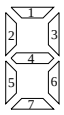
$$P(\mathbf{z} \mid s) = \prod_{i=1}^{I} \text{Normal}(z_i; 0, \sigma_{si}^2), \tag{39.24}$$

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'.