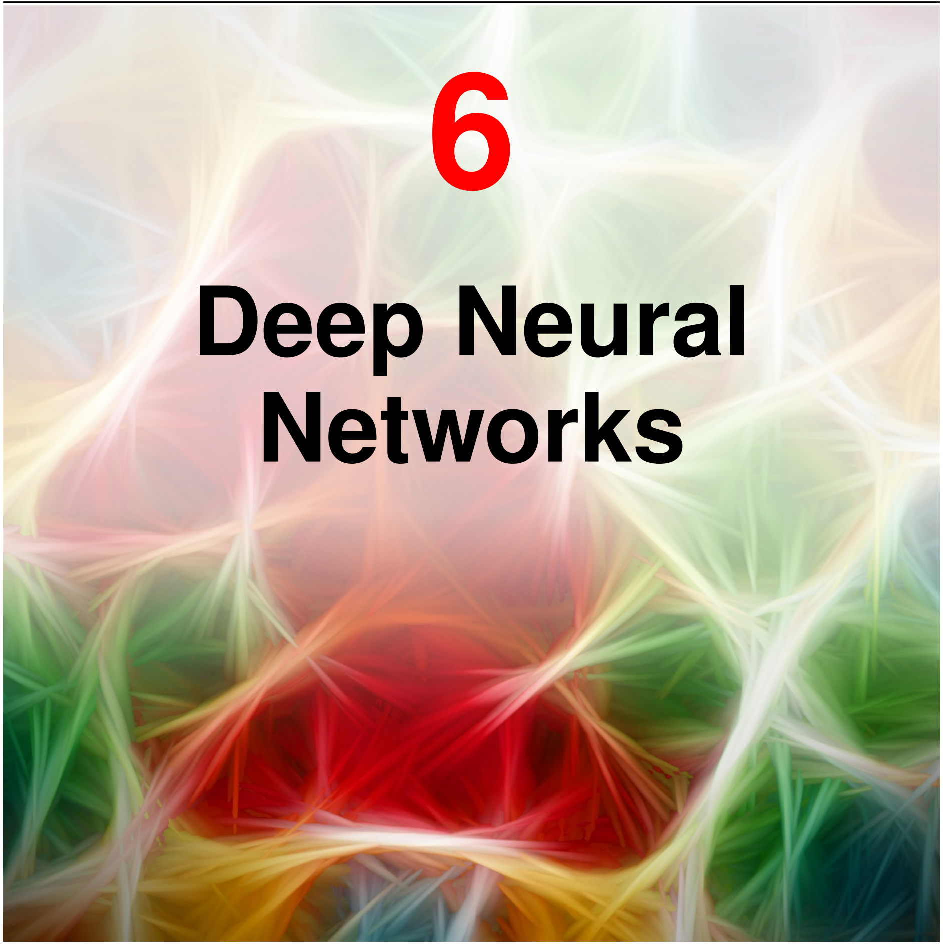


6 Deep Neural Networks

In recent years, neural networks have emerged as, by far, the most important machine learning technology for practical applications, and we therefore devote a large fraction of this book to studying them. Previous chapters have already laid many of the foundations we will need. In particular, we have seen that linear regression models that comprise linear combinations of fixed nonlinear basis functions can be expressed as neural networks having a single layer of weight and bias parameters. Likewise, classification models based on linear combinations of basis functions can also be viewed as single-layer neural networks. These allowed us to introduce several important concepts before we embark on a discussion of more complex multilayered networks in this chapter.

Chapter 4

Chapter 5

Given a sufficient number of suitably chosen basis functions, such linear models can approximate any given nonlinear transformation from inputs to outputs to any desired accuracy and might therefore appear to be sufficient to tackle any practical application. However, these models have some severe limitations, and so we will begin our discussion of neural networks by exploring these limitations and understanding why it is necessary to use basis functions that are themselves learned from data. This leads naturally to a discussion of neural networks having more than one layer of learnable parameters. These are known as *feed-forward networks* or *multilayer perceptrons*. We will also discuss the benefits of having many such layers of processing, leading to the key concept of *deep neural networks* that now dominate the field of machine learning.

Section 6.3.6

6.1. Limitations of Fixed Basis Functions

Chapter 5

Linear basis function models for classification are based on linear combinations of basis functions $\phi_j(\mathbf{x})$ and take the form

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x}) + w_0\right) \tag{6.1}$$

where $f(\cdot)$ is a nonlinear output activation function. Linear models for regression take the same form but with $f(\cdot)$ replaced by the identity. These models allow for an arbitrary set of nonlinear basis functions $\{\phi_i(\mathbf{x})\}$, and because of the generality of these basis functions, such models can in principle provide a solution to any regression or classification problem. This is true in a trivial sense in that if one of the basis functions corresponds to the desired input-to-output transformation, then the learnable linear layer simply has to copy the value of this basis function to the output of the model.

Chapter 4

More generally, we would expect that a sufficiently large and rich set of basis functions would allow any desired function to be approximated to arbitrary accuracy. It would seem therefore that such linear models constitute a general purpose framework for solving problems in machine learning. Unfortunately, there are some significant shortcomings with linear models, which arise from the assumption that the basis functions $\phi_j(\mathbf{x})$ are fixed and independent of the training data. To understand these limitations, we start by looking at the behaviour of linear models as the number of input variables is increased.

6.1.1 The curse of dimensionality

Consider a simple regression model for a single input variable given by a polynomial of order M in the form

Section 1.2

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M \tag{6.2}$$

and let us see what happens if we increase the number of inputs. If we have D input variables $\{x_1, \ldots, x_D\}$, then a general polynomial with coefficients up to order 3

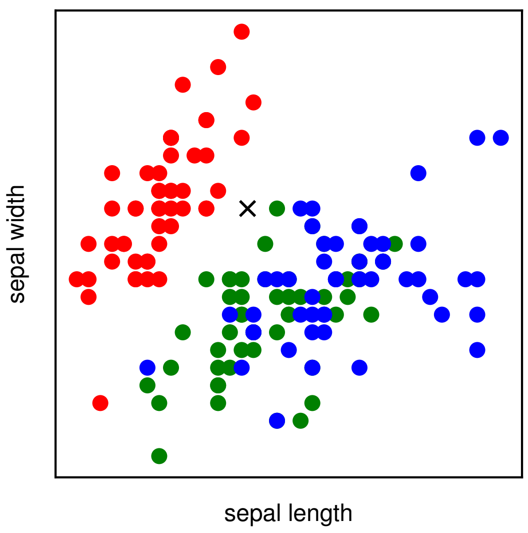


Figure 6.1 Plot of the Iris data in which red, green, and blue points denote three species of iris flower and the axes represent measurements of the length and width of the sepal, respectively. Our goal is to classify a new test point such as the one denoted by $\times$.

would take the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij} x_i x_j + \sum_{i=1}^{D}\sum_{j=1}^{D}\sum_{k=1}^{D} w_{ijk} x_i x_j x_k. \quad (6.3)$$

As D increases, the growth in the number of independent coefficients is $\mathcal{O}(D^3)$, whereas for a polynomial of order M, the growth in the number of coefficients is $\mathcal{O}(D^M)$ (Bishop, 2006). We see that in spaces of higher dimensionality, polynomials can rapidly become unwieldy and of little practical utility.

The severe difficulties that can arise in spaces of many dimensions is sometimes called the *curse of dimensionality* (Bellman, 1961). It is not limited to polynomial regression but is in fact quite general. Consider the use of linear models for solving classification problems. Figure 6.1 shows a plot of data from the Iris data set comprising 50 observations taken from each of three species of iris flowers. Each observation has four variables representing measurements of the sepal length, sepal width, petal length, and petal width. For this illustration, we consider only the sepal length and sepal width variables. Given these 150 observations as training data, our goal is to classify a new test point, such as the one denoted by the cross in Figure 6.1, by assigning it to one of the three species. We observe that the cross is close to several red points, and so we might suppose that it belongs to the red class. However, there are also some green points nearby, so we might think that it could instead belong to the green class. It seems less likely that it belongs to the blue class. The intuition here is that the identity of the cross should be determined more strongly by nearby points from the training set and less strongly by more distant points, and this intuition turns out to be reasonable.

One very simple way of converting this intuition into a learning algorithm would be to divide the input space into regular cells, as indicated in Figure 6.2. When we are given a test point and we wish to predict its class, we first decide which cell it

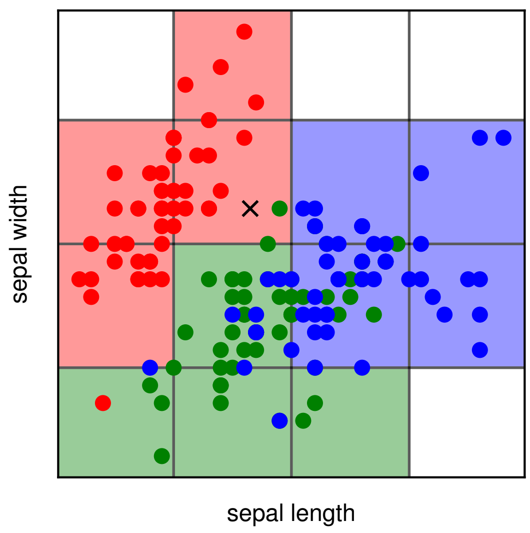


Figure 6.2 Illustration of a simple approach for solving classification problems in which the input space is divided into cells and any new test point is assigned to the class that has the most representatives in the same cell as the test point. As we shall see shortly, this simplistic approach has some severe shortcomings.

belongs to, and then we find all the training data points that fall in the same cell. The identity of the test point is predicted to be the same as the class having the largest number of training points in the same cell as the test point (with ties being broken at random). We can view this as a basis function model in which there is a basis function $\phi_i(\mathbf{x})$ for each grid cell, which simply returns zero if $\mathbf{x}$ lies outside the grid cell, and otherwise returns the majority class of the training data points that fall inside the cell. The output of the model is then given by the sum of the outputs of all the basis functions.

There are numerous problems with this naive approach, but one of the most severe becomes apparent when we consider its extension to problems having larger numbers of input variables, corresponding to input spaces of higher dimensionality. The origin of the problem is illustrated in Figure 6.3, which shows that, if we divide a region of a space into regular cells, then the number of such cells grows exponentially with the dimensionality of the space. The challenge with an exponentially large number of cells is that we would need an exponentially large quantity of training

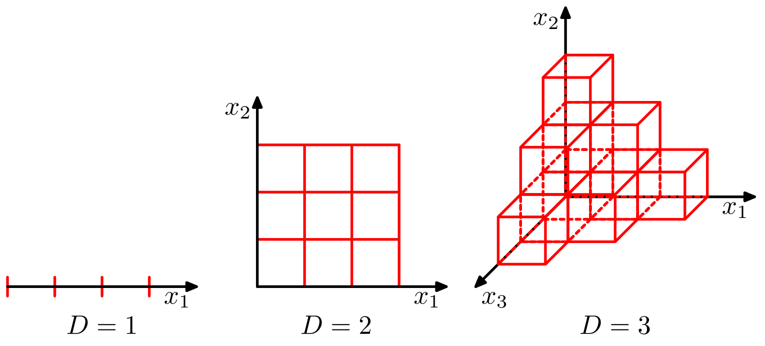


Figure 6.3 Illustration of the curse of dimensionality, showing how the number of regions of a regular grid grows exponentially with the dimensionality D of the space. For clarity, only a subset of the cubical regions are shown for $D = 3$.

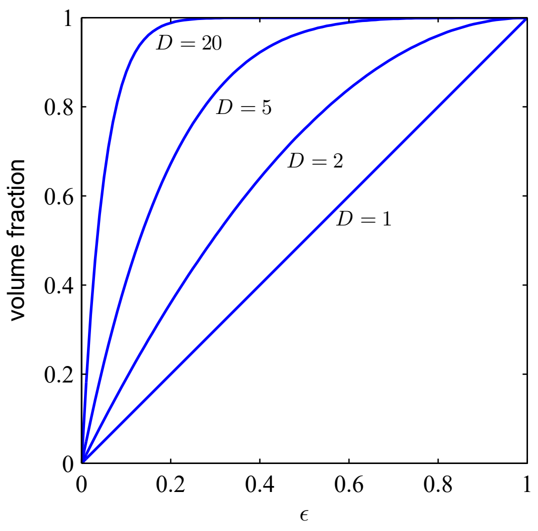


Figure 6.4 Plot of the fraction of the volume of a hypersphere of radius $r = 1$ lying in the range $r = 1 - \epsilon$ to $r = 1$ for various values of the dimensionality D.

data to ensure that the cells are not empty. We have already seen in Figure 6.2 that some cells contain no training points. Hence, a test point in such cells cannot be classified. Clearly, we have no hope of applying such a technique in a space of more than a few variables. The difficulties with both the polynomial regression example and the Iris data classification example arise because the basis functions were chosen independently of the problem being solved. We will need to be more sophisticated in our choice of basis functions if we are to circumvent the curse of dimensionality. *Section 6.1.4*

6.1.2 High-dimensional spaces

First, however, we will look more closely at the properties of spaces with higher dimensionality where our geometrical intuitions, formed through a life spent in a space of three dimensions, can fail badly. As a simple example, consider a hypersphere of radius $r = 1$ in a space of D dimensions, and ask what is the fraction of the volume of the hypersphere that lies between radius $r = 1 - \epsilon$ and $r = 1$. We can evaluate this fraction by noting that the volume $V_D(r)$ of a hypersphere of radius r in D dimensions must scale as r^D, and so we write

$$V_D(r) = K_D r^D \tag{6.4}$$

Exercise 6.1 where the constant K_D depends only on D. Thus, the required fraction is given by

$$\frac{V_D(1) - V_D(1-\epsilon)}{V_D(1)} = 1 - (1-\epsilon)^D, \tag{6.5}$$

which is plotted as a function of ϵ for various values of D in Figure 6.4. We see that, for large D, this fraction tends to 1 even for small values of ϵ. Thus, we arrive at the remarkable result that, in spaces of high dimensionality, most of the volume of a hypersphere is concentrated in a thin shell near the surface!

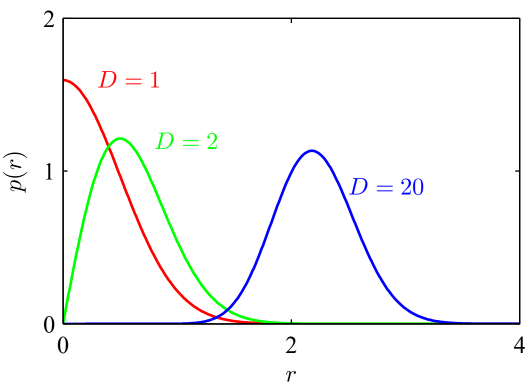


Figure 6.5 Plot of the probability density with respect to radius r of a Gaussian distribution for various values of the dimensionality D. In a high-dimensional space, most of the probability mass of a Gaussian is located within a thin shell at a specific radius.

As a further example of direct relevance to machine learning, consider the behaviour of a Gaussian distribution in a high-dimensional space. If we transform from Cartesian to polar coordinates and then integrate out the directional variables, we obtain an expression for the density $p(r)$ as a function of radius r from the origin. Thus, $p(r)\delta r$ is the probability mass inside a thin shell of thickness δr located at radius r. This distribution is plotted, for various values of D, in Figure 6.5, and we see that for large D, the probability mass of the Gaussian is concentrated in a thin shell at a specific radius.

Exercise 6.3

In this book, we make extensive use of illustrative examples involving one or two variables, because this makes it particularly easy to visualize these spaces graphically. The reader should be warned, however, that not all intuitions developed in spaces of low dimensionality will generalize to situations involving many dimensions.

Finally, although we have talked about the curse of dimensionality, there can also be advantages to working in high-dimensional spaces. Consider the situation shown in Figure 6.6. We see that this data set, in which each data point consists of a pair of values (x_1, x_2), is linearly separable, but when only the value of x_1 is observed, the classes have a strong overlap. The classification problem is therefore much easier in the higher-dimensional space.

6.1.3 Data manifolds

With both the polynomial regression model and the grid-based classifier in Figure 6.2, we saw that the number of basis functions grows rapidly with dimensionality, making such methods impractical for applications involving even a few dozen variables, never mind the millions of inputs that often arise with, say, image processing. The problem is that the basis functions are fixed ahead of time and do not depend on the data, or indeed even on the specific problem being solved. We need to find a way to create basis functions that are tuned to the particular application.

Although the curse of dimensionality certainly raises important issues for machine learning applications, it does not prevent us from finding effective techniques applicable to high-dimensional spaces. One reason for this is that real data will generally be confined to a region of the data space having lower effective dimen-

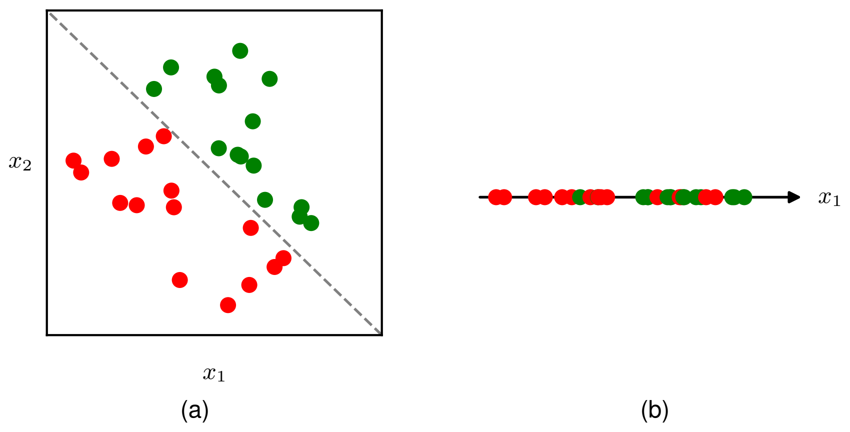


Figure 6.6 Illustration of a data set in two dimensions (x_1, x_2) in which data points from the two classes depicted using green and red circles can be separated by a linear decision surface, as seen in (a). If, however, only the variable x_1 is measured then the classes are no longer separable, as seen in (b).

sionality. Consider the images shown in Figure 6.7. Each image is a point in a high-dimensional space whose dimensionality is determined by the number of pixels. Because the objects can occur at different vertical and horizontal positions within the image and in different orientations, there are three degrees of freedom of variability between images, and a set of images will, to a first approximation, live on a three-dimensional *manifold* embedded within the high-dimensional space. Due to the complex relationships between the object position or orientation and the pixel intensities, this manifold will be highly nonlinear.

In fact, the number of pixels is really an artefact of the image generation process since they represent measurements of a continuous world. Capturing the same image at a higher resolution increases the dimensionality D of the data space without changing the fact that the images still live on a three-dimensional manifold. If we can associate localized basis functions with the data manifold, rather than with the entire high-dimensional data space, we might expect that the number of required basis functions would grow exponentially with the dimensionality of the manifold rather than with the dimensionality of the data space. Since the manifold will typically have a much lower dimensionality than the data space, this represents a huge

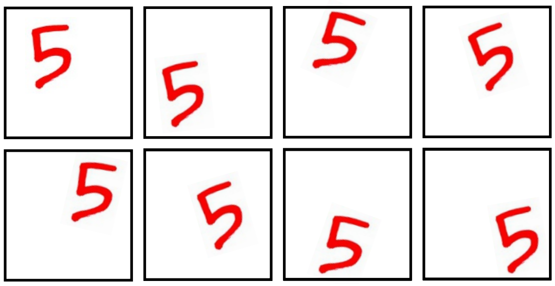


Figure 6.7 Examples of images of a handwritten digit that differ in the location of the digit within the images as well as in their orientation. This data lives on a nonlinear three-dimensional manifold within the high-dimensional image space.

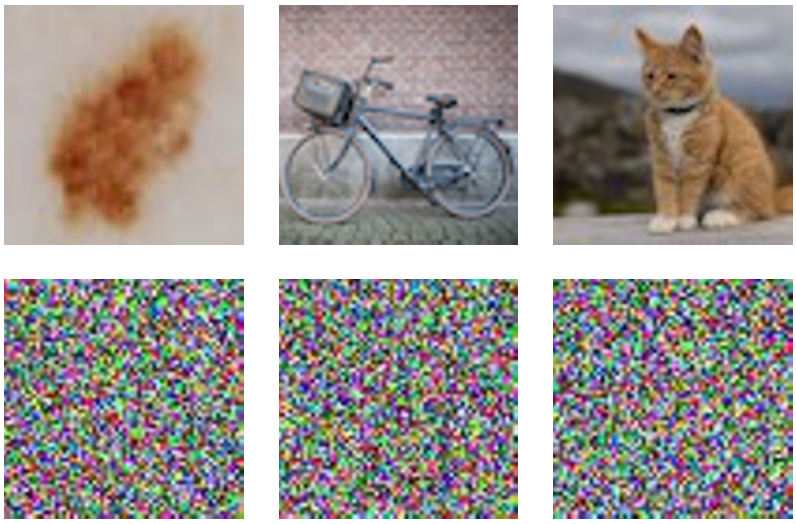


Figure 6.8 The top row shows examples of natural images of size 64×64 pixels, whereas the bottom row shows randomly generated images of the same size obtained by drawing pixel values from a uniform probability distribution over the possible pixel colours.

improvement. Effectively, neural networks learn a set of basis functions that are adapted to data manifolds. Moreover, for a particular application, not all directions within the manifold may be significant. For example, if we wish to determine only the orientation, and not the position, of the object in Figure 6.7, then there is only one relevant degree of freedom on the manifold and not three. Neural networks are also able to learn which directions on the manifold are relevant to predicting the desired outputs.

Another way to see that real data is confined to low-dimensional manifolds is to consider the task of generating random images. In Figure 6.8 we see examples of natural images along with examples of synthetic images of the same resolution generated by sampling each of the red, green, and blue intensities at each pixel independently at random from a uniform distribution. We see that none of the synthetic images look at all like natural images. The reason is that these random images lack the very strong correlations between pixels that natural images exhibit. For example, two adjacent pixels in a natural image have a much higher probability of having the same, or very similar, colour, than would two adjacent images in the random examples. Each of the images in Figure 6.8 corresponds to a point in a high-dimensional space, yet natural images cover only a tiny fraction of this space.

6.1.4 Data-dependent basis functions

We have seen that simple basis functions that are chosen independently of the problem being solved can run into significant limitations, particularly in spaces of high dimensionality. If we want to use basis functions in such situations, then one approach would be to use expert knowledge to hand-craft the basis functions in a way that is specific to each application. For many years, this was the mainstream approach in machine learning. Basis functions, often called *features*, would be determined by a combination of domain knowledge and trial-and-error. However, this approach met with limited success and was superseded by data-driven approaches in which basis functions are learned from the training data. Domain knowledge still plays a role in modern machine learning, but at a more qualitative level in designing network architectures where it can capture appropriate *inductive bias*, as we will see in later chapters.

Section 9.1

Since data in a high-dimensional space may be confined to a low-dimensional manifold, we do not need basis functions that densely fill the whole input space, but instead we can use basis functions that are themselves associated with the data manifold. One way to do this is to have one basis function associated with each data point in the training set, which ensures that the basis functions are automatically adapted to the underlying data manifold. An example of such a model is that of *radial basis functions* (Broomhead and Lowe, 1988), which have the property that each basis function depends only on the radial distance (typically Euclidean) from a central vector. If the basis centres are chosen to be the input data values $\{\mathbf{x}_n\}$ then there is one basis function $\phi_n(\mathbf{x})$ for each data point, which will therefore capture the whole of the data manifold. A typical choice for a radial basis function is

$$\phi_n(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_n\|^2}{s^2}\right) \tag{6.6}$$

where s is a parameter controlling the width of the basis function. Although it can be quick to set up such a model, a major problem with this technique is that it becomes computationally unwieldy for large data sets. Moreover, the model needs careful regularization to avoid severe over-fitting.

A related approach, called a *support vector machine* or SVM (Vapnik, 1995; Schölkopf and Smola, 2002; Bishop, 2006), addresses this by again defining basis functions that are centred on each of the training data points and then selecting a subset of these automatically during training. As a result, the effective number of basis functions in the resulting models is generally much smaller than the number of training points, although it is often still relatively large and typically increases with the size of the training set. Support vector machines also do not produce probabilistic outputs, and they do not naturally generalize to more than two classes. Methods such as radial basis functions and support vector machines have been superseded by deep neural networks, which are much better at exploiting very large data sets efficiently. Moreover, as we will see later, neural networks are able to learn deep *hierarchical* representations, which are crucial to achieving high prediction accuracy in more complex applications.

6.2. Multilayer Networks

In the previous section, we saw that to apply linear models of the form (6.1) to problems involving large-scale data sets and high-dimensional spaces, we need to find a set of basis functions that is tuned to the problem being solved. The key idea behind neural networks is to choose basis functions $\phi_j(\mathbf{x})$ that themselves have learnable parameters and then allow these parameters to be adjusted, along with the coefficients $\{w_j\}$, during training. We then optimize the whole model by minimizing an error function using gradient-based optimization methods, such as stochastic gradient descent, where the error function is defined jointly across all the parameters in the model.

Chapter 7

There are, of course, many ways to construct parametric nonlinear basis functions. One key requirement is that they must be differentiable functions of their learnable parameters so that we can apply gradient-based optimization. The most successful choice has been to use basis functions that follow the same form as (6.1), so that each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are learnable parameters. Note that this construction can clearly be extended recursively to give a hierarchical model with many layers, which forms the basis for deep neural networks.

Section 6.3

Consider a basic neural network model having two layers of learnable parameters. First, we construct M linear combinations of the input variables $x_1, \ldots, x_D$ in the form

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{6.7}$$

where $j = 1, \ldots, M$, and the superscript (1) indicates that the corresponding parameters are in the first 'layer' of the network. We will refer to the parameters $w_{ji}^{(1)}$ as *weights* and the parameters $w_{j0}^{(1)}$ as *biases*, while the quantities $a_j^{(1)}$ are called *pre-activations*. Each of the quantities a_j is then transformed using a differentiable, nonlinear *activation function* $h(\cdot)$ to give

Chapter 4

$$z_j^{(1)} = h(a_j^{(1)}), \tag{6.8}$$

which represent the outputs of the basis functions in (6.1). In the context of neural networks, these basis functions are called *hidden units*. We will explore various choices for the nonlinear function $h(\cdot)$ shortly, but here we note that provided the derivative $h'(\cdot)$ can be evaluated, then the overall network function will be differentiable. Following (6.1), these values are again linearly combined to give

$$a_k^{(2)} = \sum_{j=1}^{M} w_{kj}^{(2)} z_j^{(1)} + w_{k0}^{(2)} \tag{6.9}$$

where $k = 1, \ldots, K$, and K is the total number of outputs. This transformation corresponds to the second layer of the network, and again the $w_{k0}^{(2)}$ are bias parameters. Finally, the $\{a_k^{(2)}\}$ are transformed using an appropriate output-unit activation

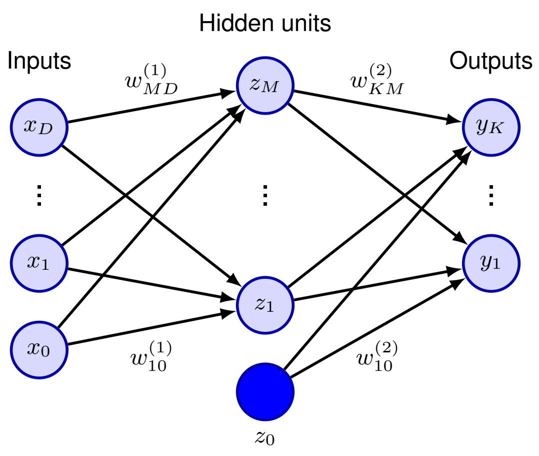


Figure 6.9 Network diagram for a two-layer neural network. The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes. The bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0 which are themselves denoted by solid nodes. Arrows denote the direction of information flow through the network during forward propagation.

function $f(\cdot)$ to give a set of network outputs y_k. A two-layer neural network can be represented in diagram form as shown in Figure 6.9.

6.2.1 Parameter matrices

Section 4.1.1

As we discussed in the context of linear regression models, the bias parameters in (6.7) can be absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$, so that (6.7) takes the form

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i. \tag{6.10}$$

We can similarly absorb the second-layer biases into the second-layer weights, so that the overall network function becomes

$$y_k(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right). \tag{6.11}$$

Another notation that will prove convenient at various points in the book is to represent the inputs as a column vector $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$ and then to gather the weight and bias parameters in (6.11) into matrices to give

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = f\left(\mathbf{W}^{(2)} h\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right) \tag{6.12}$$

where $f(\cdot)$ and $h(\cdot)$ are evaluated on each vector element separately.

6.2.2 Universal approximation

The capability of a two-layer network to model a broad range of functions is illustrated in Figure 6.10. This figure also shows how individual hidden units work collaboratively to approximate the final function. The role of hidden units in a simple classification problem is illustrated in Figure 6.11.

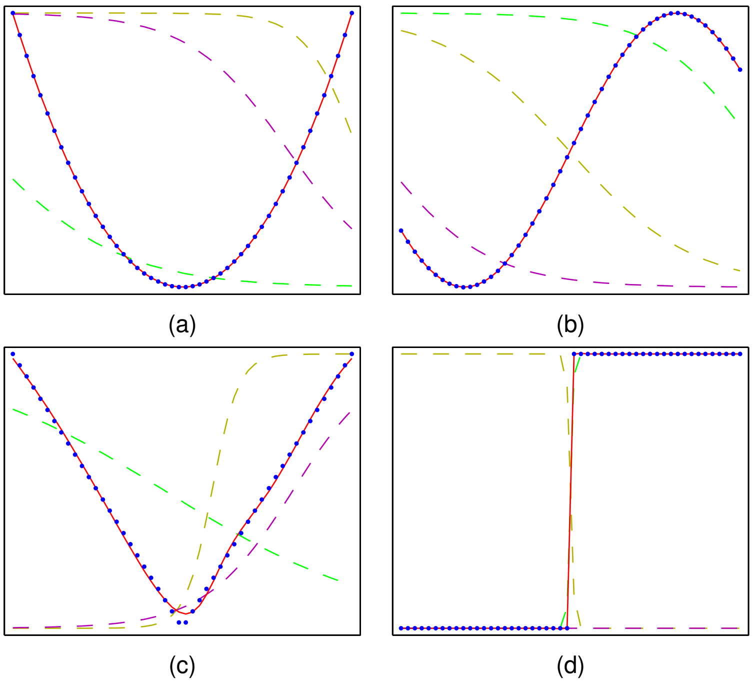


Figure 6.10 Illustration of the capability of a two-layer neural network to approximate four different functions: (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having three hidden units with tanh activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

The approximation properties of two-layer feed-forward networks were widely studied in the 1980s, with various theorems showing that, for a wide range of activation functions, such networks can approximate any function defined over a continuous subset of $\mathbb{R}^D$ to arbitrary accuracy (Funahashi, 1989; Cybenko, 1989; Hornik, Stinchcombe, and White, 1989; Leshno *et al.*, 1993). A similar result holds for functions from any finite-dimensional discrete space to any another. Neural networks are therefore said to be *universal approximators*.

Although such theorems are reassuring, they tell us only that there exists a network that can represent the required function. In some cases, they may require networks that have an exponentially large number of hidden units. Moreover, they say nothing about whether such a network can be found by a learning algorithm. Furthermore, we will see later that the *no free lunch theorem* says that we can never find a truly universal machine learning algorithm. Finally, although networks having two layers of weights are universal approximators, in a practical application, there can be huge benefits in considering networks having many more than two layers that can learn hierarchical internal representations. All these points support the drive towards deep learning.

Section 9.1.2

6.2.3 Hidden unit activation functions

We have seen that the activation functions for the output units are determined by the kind of distribution being modelled. For the hidden units, however, the only requirement is that they need to be differentiable, which leaves a wide range of pos-

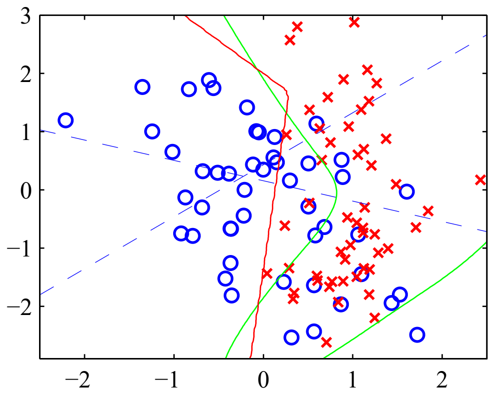


Figure 6.11 Example of the solution of a simple two-class classification problem involving synthetic data using a neural network having two inputs, two hidden units with tanh activation functions, and a single output having a logistic-sigmoid activation function. The dashed blue lines show the $z = 0.5$ contours for each of the hidden units, and the red line shows the $y = 0.5$ decision surface for the network. For comparison, the green lines denote the optimal decision boundary computed from the distributions used to generate the data.

sibilities. In most cases, all the hidden units in a network will be given the same activation function, although in principle there is no reason why different choices could not be applied in different parts of the network.

The simplest option for a hidden unit activation function is the identity function, which means that all the hidden units become linear. However, for any such network, we can always find an equivalent network without hidden units. This follows from the fact that the composition of successive linear transformations is itself a linear transformation, and so its representational capability is no greater than that of a single linear layer. However, if the number of hidden units is smaller than either the number of input or output units, then the transformations that such a network can generate are not the most general possible linear transformation from inputs to outputs because information is lost in the dimensionality reduction at the hidden units. Consider a network with N inputs, M hidden units, and K outputs, and where all activation functions are linear. Such a network has $M(N + K)$ parameters, whereas a linear transformation of inputs directly to outputs would have NK parameters. If M is small relative to N or K, or both, this leads to a two-layer linear network having fewer parameters than the direct linear mapping, corresponding to a rank-deficient transformation. Such 'bottleneck' networks of linear units corresponds to a standard data analysis technique called *principal component analysis*. In general, however, there is limited interest in using multilayer networks of linear units since the overall function computed by such a network is still linear.

Chapter 16

A simple, nonlinear differentiable function is the logistic sigmoid given by

$$\sigma(a) = \frac{1}{1 + \exp(-a)}, \tag{6.13}$$

which is plotted in Figure 5.12. This was widely used in the early years of work on multilayer neural networks and was partly inspired by studies of the properties of

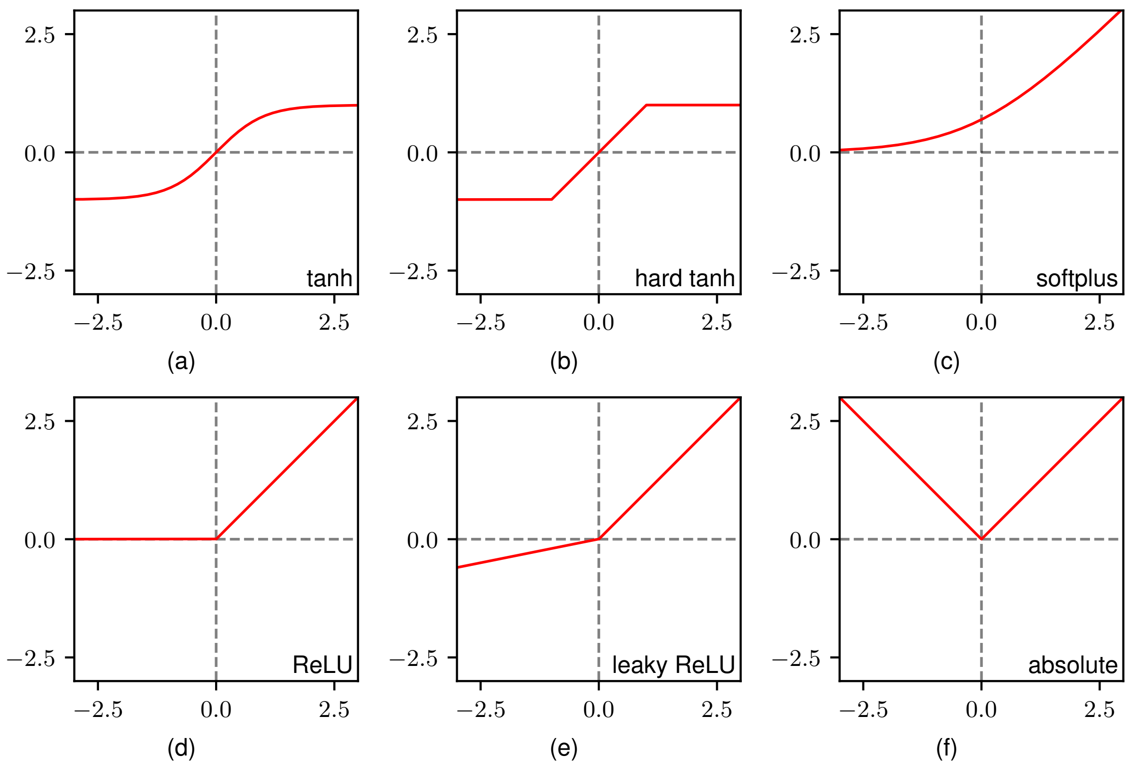


Figure 6.12 A variety of nonlinear activation functions.

biological neurons. A closely related function is tanh, which is defined by

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}, \tag{6.14}$$

which is plotted in Figure 6.12(a). This function differs from the logistic sigmoid by a linear transformation of its input and its output values, and so for any network with logistic-sigmoid hidden-unit activation functions, there is an equivalent network with tanh activation functions. However, when training a network, these are not necessarily equivalent because for gradient-based optimization, the network weights and biases need to be initialized, and so if the activation functions are changed, then the initialization scheme must be adjusted accordingly. A 'hard' version of the tanh function (Collobert, 2004) is given by

Exercise 6.4

$$h(a) = \max\left(-1, \min(1, a)\right) \tag{6.15}$$

and is plotted in Figure 6.12(b).

A major drawback of both the logistic sigmoid and the tanh activation functions is that the gradients go to zero exponentially when the inputs have either large positive or large negative values. We will discuss this 'vanishing gradients' issue later,

Section 7.4.2

but for the moment, we note that it will generally be better to use activation functions with non-zero gradients, at least when the input takes a large positive value. One such choice is the *softplus activation function* given by

Exercise 6.7

$$h(a) = \ln(1 + \exp(a)), \tag{6.16}$$

which is plotted in Figure 6.12(c). For $a \gg 1$, we have $h(a) \simeq a$, and so the gradient remains non-zero even when the input to the activation function is large and positive, thereby helping to alleviate the vanishing gradients problem.

An even simpler choice of activation function is the *rectified linear unit* or *ReLU*, which is defined by

$$h(a) = \max(0, a) \tag{6.17}$$

and which is plotted in Figure 6.12(d). Empirically, this is one of the best-performing activation functions, and it is in widespread use. Note that strictly speaking, the derivative of the ReLU function is not defined when $a = 0$, but in practice this can be safely ignored. The softplus function (6.16) can be viewed as a smoothed version of the ReLU and is therefore also sometimes called *soft ReLU*.

Exercise 6.5

Although the ReLU has a non-zero gradient for positive input values, this is not the case for negative inputs, which can mean that some hidden units receive no 'error signal' during training. A modification of ReLU that seeks to avoid this issue is called a *leaky ReLU* and is defined by

$$h(a) = \max(0, a) + \alpha \min(0, a), \tag{6.18}$$

where $0 < \alpha < 1$. This function is plotted in Figure 6.12(e). Unlike ReLU, this has a nonzero gradient for input values $a < 0$, which ensures that there is a signal to drive training. A variant of this activation function uses $\alpha = -1$, in which case $h(a) = |a|$, which is plotted in Figure 6.12(f). Another variant allows each hidden unit to have its own value α_j, which can be learned during network training by evaluating gradients with respect to the $\{\alpha_j\}$ along with the gradients with respect to the weights and biases.

The introduction of ReLU gave a big improvement in training efficiency over previous sigmoidal activation functions (Krizhevsky, Sutskever, and Hinton, 2012). As well as allowing deeper networks to be trained efficiently, it is much less sensitive to the random initialization of the weights. It is also well suited to a low-precision implementation, such as 8-bit fixed versus 64-bit floating point, and it is computationally cheap to evaluate. Many practical applications simply use ReLU units as the default unless the goal is explicitly to explore the effects of different choices of activation function.

6.2.4 Weight-space symmetries

One property of feed-forward networks is that multiple distinct choices for the weight vector $\mathbf{w}$ can all give rise to the same mapping function from inputs to outputs (Chen, Lu, and Hecht-Nielsen, 1993). Consider a two-layer network of the form shown in Figure 6.9 with M hidden units having tanh activation functions and full connectivity in both layers. If we change the sign of all the weights and the bias feeding into a particular hidden unit, then, for a given input data point, the sign of the pre-activation of the hidden unit will be reversed, and therefore so too will the activation, because tanh is an odd function, so that $\tanh(-a) = -\tanh(a)$. This transformation can be exactly compensated for by changing the sign of all the weights leading out of that hidden unit. Thus, by changing the signs of a particular group of weights (and a bias), the input–output mapping function represented by the network is unchanged, and so we have found two different weight vectors that give rise to the same mapping function. For M hidden units, there will be M such 'sign-flip' symmetries, and thus, any given weight vector will be one of a set 2^M equivalent weight vectors .

Similarly, imagine that we interchange the values of all of the weights (and the bias) leading both into and out of a particular hidden unit with the corresponding values of the weights (and bias) associated with a different hidden unit. Again, this clearly leaves the network input–output mapping function unchanged, but it corresponds to a different choice of weight vector. For M hidden units, any given weight vector will belong to a set of $M \times (M-1) \times \cdots \times 2 \times 1 = M!$ equivalent weight vectors associated with this interchange symmetry, corresponding to the $M!$ different orderings of the hidden units. The network will therefore have an overall weight-space symmetry factor of $M!\,2^M$. For networks with more than two layers of weights, the total level of symmetry will be given by the product of such factors, one for each layer of hidden units.

It turns out that these factors account for all the symmetries in weight space (except for possible accidental symmetries due to specific choices for the weight values). Furthermore, the existence of these symmetries is not a particular property of the tanh function but applies to a wide range of activation functions (Kurková and Kainen, 1994). In general, these symmetries in weight space are of little practical consequence, since network training aims to find a specific setting for the parameters, and the existence of other, equivalent, settings is of little consequence. However, weight-space symmetries do play a role when Bayesian methods are used to evaluate the probability distribution over networks of different sizes (Bishop, 2006).

6.3. Deep Networks

We have motivated the development of neural networks by making the basis functions of a linear regression or classification model themselves be governed by learnable parameters, giving rise to the two-layer network model shown in Figure 6.9. For many years, this was the most widely used architecture, primarily because it proved difficult to train networks with more than two layers effectively. However, extending neural networks to have more than two layers, known as deep neural networks, brings many advantages as we will discuss shortly, and recent advances in techniques for training neural networks are effective for networks with many layers.

Chapter 7

We can easily extend the two-layer network architecture (6.12) to any finite number L of layers, in which layer $l = 1, \ldots, L$ computes the following function:

$$\mathbf{z}^{(l)} = h^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)}\right) \tag{6.19}$$

where $h^{(l)}$ denotes the activation function associated with layer l, and $\mathbf{W}^{(l)}$ denotes the corresponding matrix of weight and bias parameters. Also, $\mathbf{z}^{(0)} = \mathbf{x}$ represents the input vector and $\mathbf{z}^{(L)} = \mathbf{y}$ represents the output vector.

Note that there has been some confusion in the literature regarding the terminology for counting the number of layers in such networks. Thus, the network in Figure 6.9 is sometimes described as a three-layer network (which counts the number of layers of units and treats the inputs as units) or sometimes as a single-hidden-layer network (which counts the number of layers of hidden units). We recommend a terminology in which Figure 6.9 is called a two-layer network, because it is the number of layers of learnable weights that is important for determining the network properties.

We have seen that a network of the form shown in Figure 6.9, having two layers of learnable parameters, has universal approximation capabilities. However, networks with more than two layers can sometimes represent a given function with far fewer parameters than a two-layer network. Montúfar *et al.* (2014) show that the network function divides the input space into a number of regions that is exponential in the depth of the network, but which is only polynomial in the width of the hidden layers. To represent the same function using a two-layer network would require an exponential number of hidden units.

6.3.1 Hierarchical representations

Although this is an interesting result, a more compelling reason to explore deep neural networks is that the network architecture encodes a particular form of inductive bias, namely that the outputs are related to the input space through a hierarchical representation. A good example is the task of recognizing objects in images. The relationship between the pixels of an image and a high-level concept such as 'cat' is highly complex and nonlinear, and would be an extremely challenging problem for a two-layer network. However, a deep neural network can learn to detect low-level features, such as edges, in the early layers, and can then combine these in subsequent layers to make higher-level features such as eyes or whiskers, which in turn can be combined in later layers to detect the presence of a cat. This can be viewed as a *compositional* inductive bias, in which higher-level objects, such as a cat, are composed of lower-level objects, such as eyes, which in turn have yet lower-level elements such as edges. We can also think of this in reverse by considering the process of generating an image starting with low-level features such as edges, then combining these to form simple shapes such as circles, and then combining those in turn to form higher-level objects such as cats. At each stage there are many ways to combine different components, giving an exponential gain in the number of possibilities with increasing depth.

Chapter 10

6.3.2 Distributed representations

Neural networks can take advantage of another form of compositionality called a *distributed representation*. Conceptually, each unit in a hidden layer can be thought of as representing a 'feature' at that level of the network, with a high value of the activation indicating that the corresponding feature is present and a low value indicating its absence. With M units in a given layer, such a network can represent M different features. However, the network could potentially learn a different representation in which *combinations* of hidden units represent features, thereby potentially allowing a hidden layer with M units to represent 2^M different features, growing exponentially with the number of units. Consider, for example, a network designed to process images of faces. Each particular face image may or may not have glasses, it may or may not have a hat, and it may or may not have a beard, leading to eight different combinations. Although this could be represented by eight units each of which 'turns on' when it detects the corresponding combination, it could also be represented more compactly by just three units, one for each attribute. These can be present independently of each other (although statistically their presence is likely to be correlated to some degree). Later, we will explore in detail the kinds of internal representations that deep learning networks discover for themselves during training. *Chapter 10*

6.3.3 Representation learning

We can view the successive layers of a deep neural network as performing transformations of the data, that make it easier to solve the desired task or tasks. For example, a neural network that successfully learns to classify skin lesions as benign or malignant must have learned to transform the original image data into a new space, *Section 1.1.1* represented by the outputs of the final layer of hidden units, such that the final layer of the network can distinguish the two classes. This final layer can be viewed as a simple linear classifier, and so in the representation of the last hidden layer, the two classes must be well separated by a linear surface. This ability to discover a nonlinear transformation of the data that makes subsequent tasks easier to solve is called *representation learning* (Bengio, Courville, and Vincent, 2012). The learned representation, sometimes called the *embedding space*, is given by the outputs of one of the hidden layers of the network, so that any input vector, either from the training set or from some new data set, can be transformed into this representation by forward propagation through the network.

Representation learning is especially powerful because it allows us to exploit unlabelled data. Often it is easy to collect a large quantity of unlabelled data, but acquiring the associated labels may be more difficult. For example, a video camera on a vehicle can gather large numbers of images of urban scenes as the vehicle is driven around a city, but taking those images and identifying relevant objects, such as pedestrians and road signs, would require expensive and time-consuming human labelling.

Learning from unlabelled data is called *unsupervised learning*, and many different algorithms have been developed to do this. For example, a neural network can be trained to take images as input and to create the same images as the output. To make this into a non-trivial task, the network may use hidden layers with fewer units than the number of pixels in the image, thereby forcing the network to learn some kind of compression of the images. Only unlabelled data is needed because each image in the training set acts as both the input vector and the target vector. Such networks are known as *autoencoders*. *Section 19.1* The goal is that this type of training will force the network to discover some internal representation for the data that is useful for solving other tasks, such as image classification.

Historically, unsupervised learning played an important role in enabling the first deep networks (apart from convolutional networks) to be successfully trained. Each layer of the network was first pre-trained using unsupervised learning and then the entire network was trained further using gradient-based supervised training. It was later discovered that the pre-training phase could be omitted and a deep network could be trained from scratch purely using supervised learning given appropriate conditions.

However, pre-training and representation learning remain central to deep learning in other contexts. The most notable example of pre-training is in natural language processing in which transformer models are trained on large quantities of text and are able to learn highly sophisticated internal representations of language that facilitates an impressive range of capabilities at human level and beyond.

Chapter 12

6.3.4 Transfer learning

The internal representation learned for one particular task might also be useful for related tasks. For example, a network trained on a large labelled data set of everyday objects can learn how to transform an image representation into one that is much better suited for classifying objects. Then, the final classification layer of the network can be retrained using a smaller labelled data set of skin lesion images to create a lesion classifier. This is an example of *transfer learning* (Hospedales *et al.*, 2021), which allows higher accuracy to be achieved than if only the lesion image data were used for training, because the network can exploit commonalities shared by natural images in general. Transfer learning is illustrated in Figure 6.13.

Section 1.1.1

In general, transfer learning can be used to improve performance on some task A, for which training data is in short supply, by using data from a related task B, for which data is more plentiful. The two tasks should have the same kind of inputs, and there should be some commonality between the tasks so that low-level features, or internal representations, learned from task B will be useful for task A. When we look at convolutional networks we will see that many image processing tasks require similar low-level features corresponding to the early layers of a deep neural network, whereas later layers are more specialized to a particular task, making such networks well suited to transfer learning applications.

Chapter 10

When data for task A is very scarce, we might simply re-train the final layer of the network. In contrast, if there are more data points, it is feasible to retrain several layers. The process of learning parameters using one task that are then applied to one or more other tasks is called *pre-training*. Note that for the new task, instead of applying stochastic gradient descent to the whole network, it is much more efficient to send the new training data once through the fixed pre-trained network so as to evaluate the training inputs in the new representation. Iterative gradient-based optimization can then be applied just to the smaller network consisting of the final layers. As well as using a pre-trained network as a fixed pre-processor for a different task, it is also possible to apply *fine-tuning* in which the whole network is adapted to the data for task A. This is generally done with a very small learning rate for a lim-

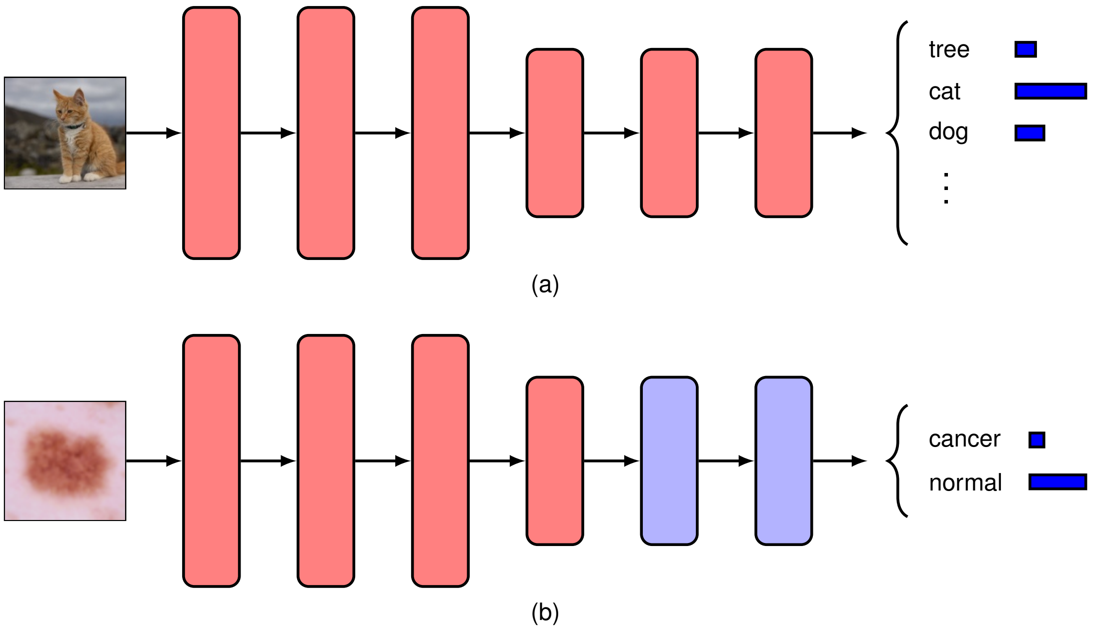


Figure 6.13 Schematic illustration of transfer learning. (a) A network is first trained on a task with abundant data, such as object classification of natural images. (b) The early layers of the network (shown in red) are copied from the first task and the final few layers of the network (shown in blue) are then retrained on a new task such as skin lesion classification for which training data is more scarce.

ited number of iterations to ensure that the network does not over-fit to the relatively small data set available for the new task.

A related approach is *multitask learning* (Caruana, 1997) in which a network jointly learns more than one related task at the same time. For example, we might wish to construct a spam email filter that allows different users to have different classifiers tuned to their particular preferences. The training data may comprise examples of spam email and non-spam email for many different users, but the number of examples for any one user may be quite limited, and therefore training a separate classifier for each user would give poor results. Instead, we can combine the data sets to train a single larger network that might, for example, share early layers but have separate learnable parameters for the different users in later layers. Sharing data across tasks allows the network to exploit commonalities amongst the tasks, thereby improving the accuracy for all users. With a large number of training examples, a deeper network with more parameters can be used, again leading to improved performance.

Learning across multiple tasks can be extended to *meta-learning*, which is also called *learning to learn*. Whereas multitask learning aims to make predictions for a fixed set of tasks, the aim of meta-learning is to make predictions for future tasks that were not seen during training. This can be done by not only learning a shared internal representation across tasks but also by learning the learning algorithm itself (Hospedales *et al.*, 2021). Meta-learning can be used to facilitate generalization of, for example, a classification model to new classes when there are very few labelled examples of the new classes. This is referred to as *few-shot learning*. When only a single labelled example is used it is called *one-shot learning*.

6.3.5 Contrastive learning

One of the most common and powerful representation learning methods is *contrastive learning* (Gutmann and Hyvärinen, 2010; Oord, Li, and Vinyals, 2018; Chen, Kornblith, *et al.*, 2020). The idea is to learn a representation such that certain pairs of inputs, referred to as positive pairs, are close in the embedding space, and other pairs of inputs, called negative pairs, are far apart. The intuition is that if we choose our positive pairs in such a way that they are semantically similar and choose negative pairs that are semantically dissimilar, then we will learn a representation space in which similar inputs are close, making downstream tasks, such as classification, much easier. As with other forms of representation learning, the outputs of the trained network are typically not used directly, and instead the activations at some earlier layer are used to form the embedding space. Contrastive learning is unlike most other machine learning tasks, in that the error function for a given input is defined only with respect to other inputs, instead of having a per-input label or target output.

Suppose we have a given data point $\mathbf{x}$ called the *anchor*, for which we have specified another data point $\mathbf{x}^+$ that together with $\mathbf{x}$ makes up a positive pair. We must also specify a set of data points $\{\mathbf{x}_1^-, \dots, \mathbf{x}_N^-\}$ each of which makes up a negative pair with $\mathbf{x}$. We now need a loss function that will reward close proximity between the representations of $\mathbf{x}$ and $\mathbf{x}^+$ while encouraging a large distance between each pair $\{\mathbf{x}, \mathbf{x}_n^-\}$. One example of such a function, and the most commonly used loss function for contrastive learning, is called the *InfoNCE* loss (Gutmann and Hyvärinen, 2010; Oord, Li, and Vinyals, 2018), where NCE denotes ‘noise contrastive estimation’. Suppose we have a neural network function $\mathbf{f}_{\mathbf{w}}(\mathbf{x})$ that maps points from the input space $\mathbf{x}$ to a representation space, governed by learnable parameters $\mathbf{w}$. This representation is normalized so that $||\mathbf{f}_{\mathbf{w}}(\mathbf{x})|| = 1$. Then, for a data point $\mathbf{x}$, the InfoNCE loss is defined by

$$E(\mathbf{w}) = -\ln \frac{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x})^{\mathrm{T}}\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)\}}{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x})^{\mathrm{T}}\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)\} + \sum_{n=1}^{N} \exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x})^{\mathrm{T}}\mathbf{f}_{\mathbf{w}}(\mathbf{x}_n^-)\}}. \tag{6.20}$$

We can see that in this function, the cosine similarity $\mathbf{f}_{\mathbf{w}}(\mathbf{x})^{\mathrm{T}}\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)$ between the representation $\mathbf{f}_{\mathbf{w}}(\mathbf{x})$ of the anchor and the representation $\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)$ of the positive example provides our measure of how close the positive pair examples are in the learned space, and the same measure is used to assess how close the anchor is to the negative examples. Note that the function resembles a classification cross-entropy error function in which the cosine similarity of the positive pair gives the logit for the label class and the cosine similarities for the negative pairs give the logits for the incorrect classes. Also note that the negative pairs are crucial as without them the embedding would simply learn the degenerate solution of mapping every point to the same representation.

A particular contrastive learning algorithm is defined predominantly by how the positive and negative pairs are chosen, which is how we use our prior knowledge to specify what a good representation should be. For example, consider the problem of learning representations of images. Here, a common choice is to create positive pairs by corrupting the input images in ways that should preserve the semantic information of the image while greatly altering the image in the pixel space (Wu *et al.*, 2018; He *et al.*, 2019; Chen, Kornblith, *et al.*, 2020). Corruptions are closely related to *data augmentations*, and examples include rotation, translation, and colour shifts. Other images from the data set can then be used to create the negative pairs. This approach to contrastive learning is known as *instance discrimination*.

Section 9.1.3

If, however, we have access to class labels, then we can use images of the same class as positive pairs and images of different classes as negative pairs. This relaxes the reliance on specifying the augmentations that the representation should be invariant to and also avoids treating two semantically similar images as a negative pair. This is referred to as supervised contrastive learning (Khosla *et al.*, 2020) because of the reliance on the class labels, and it can often yield better results than simply learning the representation using cross-entropy classification.

The members of positive and negative pairs do not necessarily have to come from the same data modality. In contrastive-language image pretraining, or CLIP (Radford *et al.*, 2021), a positive pair consists of an image and its corresponding text caption, and two separate functions, one for each modality, are used to map the inputs to the same representation space. Negative pairs are then mismatched images and captions. This is often referred to as *weakly supervised*, as it relies on captioned images, which are often easier to obtain by scraping data from the internet than by manually labelling images with their classes. The loss function in this case is given by

$$\begin{aligned} E(\mathbf{w}) = &-\frac{1}{2}\ln\frac{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\}}{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\} + \sum_{n=1}^{N}\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}_n^-)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\}} \\ &-\frac{1}{2}\ln\frac{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\}}{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\} + \sum_{m=1}^{M}\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}_m^-)\}} \end{aligned} \tag{6.21}$$

where $\mathbf{x}^+$ and $\mathbf{y}^+$ represent a positive pair in which $\mathbf{x}$ is an image and $\mathbf{y}$ is its corresponding text caption, $\mathbf{f}_{\mathbf{w}}$ represents the mapping from images to the representation space, and $\mathbf{g}_{\boldsymbol{\theta}}$ is the mapping from text input to the representation space. We also require a set $\{\mathbf{x}_1^-, \ldots, \mathbf{x}_N^-\}$ of other images from the data set, for which we can assume the text caption $\mathbf{y}^+$ is inappropriate, and a set $\{\mathbf{y}_1^-, \ldots, \mathbf{y}_M^-\}$ of text captions that are similarly mismatched to the input image $\mathbf{x}$. The two terms in the loss function ensure that (a) the representation of the image is close to its text caption representation relative to other image representations and (b) the text caption representation is close to the representation of the image it describes relative to other representations of text captions. Although CLIP uses text and image pairs, any data

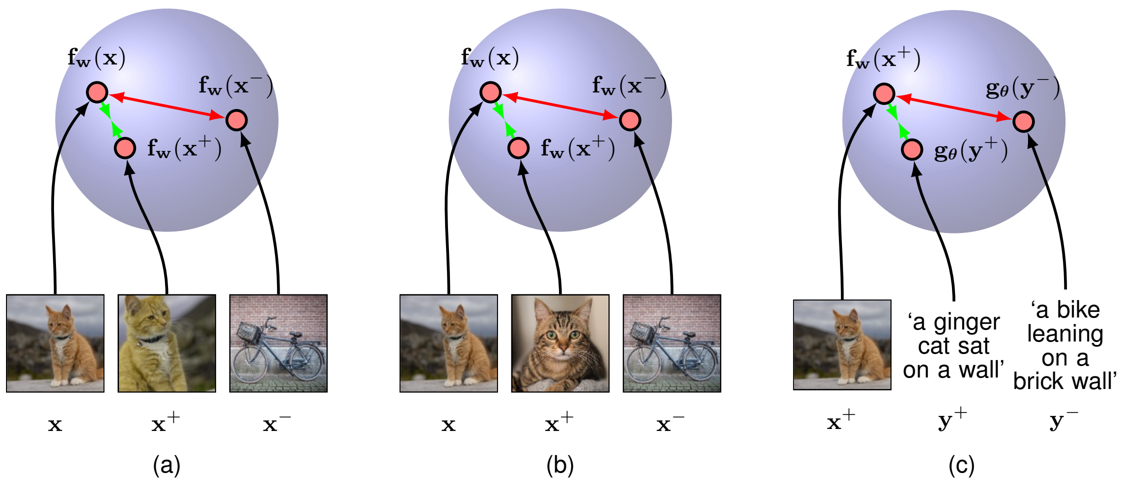


Figure 6.14 Illustration of three different contrastive learning paradigms. (a) The instance discrimination approach, where the positive pair is made up of the anchor and an augmented version of the same image. These are mapped to points in a normalized space that can be thought of as a unit hypersphere. The coloured arrows show that the loss encourages the representations of the positive pair to be closer together but pushes negative pairs further apart. (b) Supervised contrastive learning in which the positive pair consists of two different images from the same class. (c) The CLIP model in which the positive pair is made up of an image and an associated text snippet.

set with paired modalities can be used to learn representations. A comparison of the different contrastive learning methods we have discussed is shown in Figure 6.14.

6.3.6 General network architectures

So far, we have explored neural network architectures that are organized into a sequence of fully-connected layers. However, because there is a direct correspondence between a network diagram and its mathematical function, we can develop more general network mappings by considering more complex network diagrams. These must be restricted to a *feed-forward* architecture, in other words to one having no closed directed cycles, to ensure that the outputs are deterministic functions of the inputs. This is illustrated with a simple example in Figure 6.15. Each (hidden or output) unit in such a network computes a function given by

$$z_k = h\left(\sum_{j\in\mathcal{A}(k)} w_{kj}z_j + b_k\right) \tag{6.22}$$

where $\mathcal{A}(k)$ denotes the set of *ancestors* of node k, in other words the set of units that send connections to unit k, and b_k denotes the associated bias parameter. For a given set of values applied to the inputs of the network, successive application of (6.22) allows the activations of all units in the network to be evaluated including those of the output units.

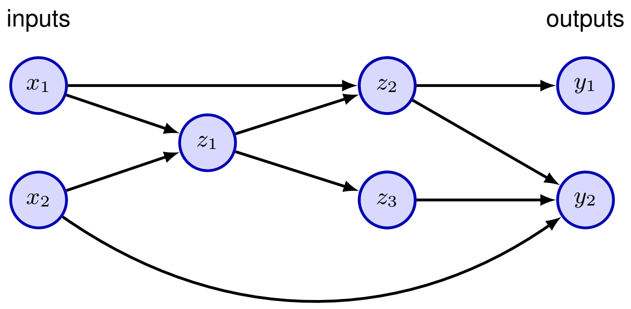


Figure 6.15 Example of a neural network having a general feed-forward topology. Note that each hidden and output unit has an associated bias parameter (omitted for clarity).

6.3.7 Tensors

We see that linear algebra plays a central role in neural networks, with quantities such as data sets, activations, and network parameters represented as scalars, vectors, and matrices. However, we also encounter variables of higher dimensionality. Consider, for example, a data set of N colour images each of which is I pixels high and J pixels wide. Each pixel is indexed by its row and column within the image and has red, green, and blue values. We have one such value for each image in the data set, and so we can represent a particular intensity value by a four-dimensional array $\mathbf{X}$ with elements x_{ijkn} where $i \in \{1, \dots, I\}$ and $j \in \{1, \dots, J\}$ index the row and column within the image, $k \in \{1, 2, 3\}$ indexes the red, green, and blue intensities, and $n \in \{1, \dots, N\}$ indexes the particular image within the data set. These higher-dimensional arrays are called *tensors* and include scalars, vectors, and matrices as special cases. We will see many examples of such tensors when we discuss more sophisticated neural network architectures later in the book. Massively parallel processors such as GPUs are especially well suited to processing tensors.

6.4. Error Functions

Chapter 4
Chapter 5

In earlier chapters, we explored linear models for regression and classification, and in the process we derived suitable forms for the error functions along with corresponding choices for the output-unit activation function. The same considerations for choosing an error function apply for multilayer neural networks, and so for convenience, we will summarize the key points here.

6.4.1 Regression

We start by discussing regression problems, and for the moment we consider a single target variable t that can take any real value. Following the discussion of regression in single-layer networks, we assume that t has a Gaussian distribution with an $\mathbf{x}$-dependent mean, which is given by the output of the neural network, so that

Section 2.3.4

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(t|y(\mathbf{x}, \mathbf{w}), \sigma^2\right) \tag{6.23}$$

where σ^2 is the variance of the Gaussian noise. Of course this is a somewhat restrictive assumption, and in some applications we will need to extend this approach to allow for more general distributions. For the conditional distribution given by (6.23), it is sufficient to take the output-unit activation function to be the identity, because such a network can approximate any continuous function from $\mathbf{x}$ to y. Given a data set of N i.i.d. observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, along with corresponding target values $\mathsf{t} = \{t_1, \ldots, t_N\}$, we can construct the corresponding likelihood function:

Section 6.5

$$p(\mathsf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} p(t_n|y(\mathbf{x}_n, \mathbf{w}), \sigma^2). \tag{6.24}$$

Note that in the machine learning literature, it is usual to consider the minimization of an error function rather than the maximization of the likelihood, and so here we will follow this convention. Taking the negative logarithm of the likelihood function (6.24), we obtain the error function

$$\frac{1}{2\sigma^2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 + \frac{N}{2}\ln\sigma^2 + \frac{N}{2}\ln(2\pi), \tag{6.25}$$

which can be used to learn the parameters $\mathbf{w}$ and σ^2. Consider first the determination of $\mathbf{w}$. Maximizing the likelihood function is equivalent to minimizing the sum-of-squares error function given by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 \tag{6.26}$$

where we have discarded additive and multiplicative constants. The value of $\mathbf{w}$ found by minimizing $E(\mathbf{w})$ will be denoted $\mathbf{w}^\star$. Note that this will typically not correspond to the global maximum of the likelihood function because the nonlinearity of the network function $y(\mathbf{x}_n, \mathbf{w})$ causes the error $E(\mathbf{w})$ to be non-convex, and so finding the global optimum is generally infeasible. Moreover, regularization terms may be added to the error function and other modifications may be made to the training process, so that the resulting solution for the network parameters may differ significantly from the maximum likelihood solution.

Chapter 9

Having found $\mathbf{w}^\star$, the value of σ^2 can be found by minimizing the error function (6.25) to give

Exercise 6.8

$$\sigma^{2\star} = \frac{1}{N}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}^\star) - t_n\}^2. \tag{6.27}$$

Note that this can be evaluated once the iterative optimization required to find $\mathbf{w}^\star$ is completed.

If we have multiple target variables, and we assume that they are independent, conditional on $\mathbf{x}$ and $\mathbf{w}$, with shared noise variance σ^2, then the conditional distribution of the target values is given by

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \sigma^2\mathbf{I}\right). \tag{6.28}$$

Following the same argument as for a single target variable, we see that maximizing the likelihood function with respect to the weights is equivalent to minimizing the sum-of-squares error function: *Exercise 6.9*

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2. \tag{6.29}$$

The noise variance is then given by

$$\sigma^{2\star} = \frac{1}{NK}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}^\star) - \mathbf{t}_n\|^2 \tag{6.30}$$

where K is the dimensionality of the target variable. The assumption of conditional independence of the target variables can be dropped at the expense of a slightly more complex optimization problem. *Exercise 6.10*

Recall that there is a natural pairing of the error function (given by the negative log likelihood) and the output-unit activation function. In regression, we can view the network as having an output activation function that is the identity, so that $y_k = a_k$. The corresponding sum-of-squares error function then has the property *Section 5.4.6*

$$\frac{\partial E}{\partial a_k} = y_k - t_k. \tag{6.31}$$

6.4.2 Binary classification

Now consider binary classification in which we have a single target variable t such that $t = 1$ denotes class $\mathcal{C}_1$ and $t = 0$ denotes class $\mathcal{C}_2$. Following the discussion of canonical link functions, we consider a network having a single output whose activation function is a logistic sigmoid (6.13) so that $0 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$. We can interpret $y(\mathbf{x}, \mathbf{w})$ as the conditional probability $p(\mathcal{C}_1|\mathbf{x})$, with $p(\mathcal{C}_2|\mathbf{x})$ given by $1 - y(\mathbf{x}, \mathbf{w})$. The conditional distribution of targets given inputs is then a Bernoulli distribution of the form *Section 5.4.6*

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \left\{1 - y(\mathbf{x}, \mathbf{w})\right\}^{1-t}. \tag{6.32}$$

If we consider a training set of independent observations, then the error function, which is given by the negative log likelihood, is then a *cross-entropy* error of the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n)\ln(1 - y_n)\} \tag{6.33}$$

where y_n denotes $y(\mathbf{x}_n, \mathbf{w})$. Simard, Steinkraus, and Platt (2003) found that using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as well as improved generalization.

Note that there is no analogue of the noise variance σ^2 in (6.32) because the target values are assumed to be correctly labelled. However, the model is easily extended to allow for labelling errors by introducing a probability ϵ that the target *Exercise 6.11* value t has been flipped to the wrong value (Opper and Winther, 2000). Here ϵ may be set in advance, or it may be treated as a hyperparameter whose value is inferred from the data.

If we have K separate binary classifications to perform, then we can use a network having K outputs each of which has a logistic-sigmoid activation function. Associated with each output is a binary class label $t_k \in \{0, 1\}$, where $k = 1, \ldots, K$. If we assume that the class labels are independent, given the input vector, then the conditional distribution of the targets is

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k} \left[1 - y_k(\mathbf{x}, \mathbf{w})\right]^{1-t_k}. \tag{6.34}$$

Exercise 6.13

Taking the negative logarithm of the corresponding likelihood function then gives the following error function:

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} \{t_{nk} \ln y_{nk} + (1 - t_{nk}) \ln(1 - y_{nk})\} \tag{6.35}$$

Exercise 6.14

where y_{nk} denotes $y_k(\mathbf{x}_n, \mathbf{w})$. Again, the derivative of the error function with respect to the pre-activation for a particular output unit takes the form (6.31), just as in the regression case.

6.4.3 multiclass classification

Section 5.1.3

Finally, we consider the standard multiclass classification problem in which each input is assigned to one of K mutually exclusive classes. The binary target variables $t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the class, and the network outputs are interpreted as $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$, leading to the error function (5.80), which we reproduce here:

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w}). \tag{6.36}$$

Section 5.4.4

The output-unit activation function, which corresponds to the canonical link, is given by the softmax function:

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}, \tag{6.37}$$

Chapter 9

Exercise 6.15

which satisfies $0 \leqslant y_k \leqslant 1$ and $\sum_k y_k = 1$. Note that the $y_k(\mathbf{x}, \mathbf{w})$ are unchanged if a constant is added to all of the $a_k(\mathbf{x}, \mathbf{w})$, causing the error function to be constant for some directions in weight space. This degeneracy is removed if an appropriate regularization term is added to the error function. Once again, the derivative of the error function with respect to the pre-activation for a particular output unit takes the familiar form (6.31).

In summary, there is a natural choice of both output-unit activation function and matching error function according to the type of problem being solved. For regression, we use linear outputs and a sum-of-squares error, for multiple independent binary classifications, we use logistic sigmoid outputs and a cross-entropy error function, and for multi-class classification, we use softmax outputs with the corresponding multi-class cross-entropy error function. For classification problems involving two classes, we can use a single logistic sigmoid output, or alternatively, we can use a network with two outputs having a softmax output activation function.

This procedure is quite general, and by considering other forms of conditional distribution, we can derive the associated error functions as the corresponding negative log likelihood. We will see an example of this in the next section when we consider multimodal network outputs.

6.5. Mixture Density Networks

So far in this chapter we have discussed neural networks whose outputs represent simple probability distributions comprising either a Gaussian for continuous variables or a binary distribution for discrete variables. We close the chapter by showing how a neural network can represent more general conditional probabilities by treating the outputs of the network as the parameters of a more complex distribution, in this case a Gaussian mixture model. This is known as a *mixture density network*, and we will see how to define the associated error function and the corresponding output-unit activation functions.

6.5.1 Robot kinematics example

The goal of supervised learning is to model a conditional distribution $p(\mathbf{t}|\mathbf{x})$, which for many simple regression problems is chosen to be Gaussian. However, practical machine learning problems can often have significantly non-Gaussian distributions. These can arise, for example, with *inverse problems* in which the distribution can be multimodal, in which case the Gaussian assumption can lead to very poor predictions.

Exercise 6.16

As a simple illustration of an inverse problem, consider the kinematics of a robot arm, as illustrated in Figure 6.16. The *forward problem* involves finding the end effector position given the joint angles and has a unique solution. However, in practice we wish to move the end effector of the robot to a specific position, and to do this we must set appropriate joint angles. We therefore need to solve the inverse problem, which has two solutions, as seen in Figure 6.16.

Forward problems often correspond to causality in a physical system and generally have a unique solution. For instance, a specific pattern of symptoms in the human body may be caused by the presence of a particular disease. In machine learning, however, we typically have to solve an inverse problem, such as trying to predict the presence of a disease given a set of symptoms. If the forward problem involves a many-to-one mapping, then the inverse problem will have multiple solutions. For instance, several different diseases may result in the same symptoms.

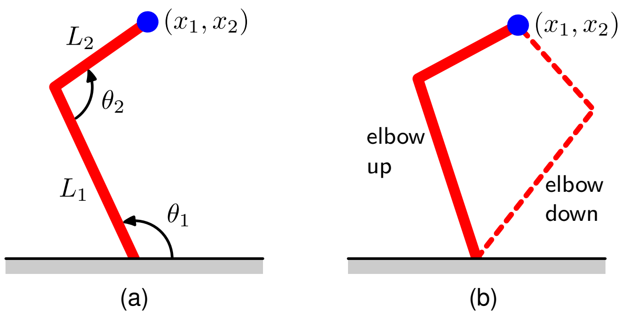


Figure 6.16 (a) A two-link robot arm, in which the Cartesian coordinates (x_1, x_2) of the end effector are determined uniquely by the two joint angles θ_1 and θ_2 and the (fixed) lengths L_1 and L_2 of the arms. This is known as the *forward kinematics* of the arm. (b) In practice, we have to find the joint angles that will give rise to a desired end effector position. This *inverse kinematics* has two solutions corresponding to 'elbow up' and 'elbow down'.

In the robotics example, the kinematics is defined by geometrical equations, and the multimodality is readily apparent. However, in many machine learning problems the presence of multimodality, particularly in problems involving spaces of high dimensionality, can be less obvious. For tutorial purposes, however, we will consider a simple toy problem for which we can easily visualize the multimodality. The data for this problem is generated by sampling a variable x uniformly over the interval $(0, 1)$, to give a set of values $\{x_n\}$, and the corresponding target values t_n are obtained by computing the function $x_n + 0.3\sin(2\pi x_n)$ and then adding uniform noise over the interval $(-0.1, 0.1)$. The inverse problem is then obtained by keeping the same data points but exchanging the roles of x and t. Figure 6.17 shows the data sets for the forward and inverse problems, along with the results of fitting two-layer neural networks having six hidden units and a single linear output unit by minimizing a sum-of-squares error function. Least squares corresponds to maximum likelihood under a Gaussian assumption. We see that this leads to a good model for the forward problem but a very poor model for the highly non-Gaussian inverse problem.

6.5.2 Conditional mixture distributions

We therefore seek a general framework for modelling conditional probability distributions. This can be achieved by using a mixture model for $p(\mathbf{t}|\mathbf{x})$ in which

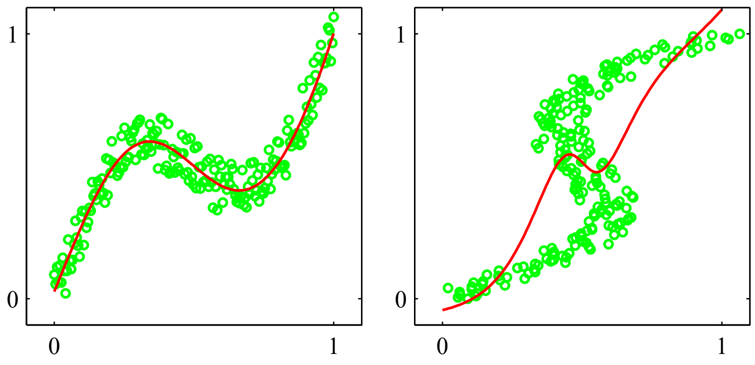


Figure 6.17 On the left is the data set for a simple forward problem in which the red curve shows the result of fitting a two-layer neural network by minimizing the sum-of-squares error function. The corresponding inverse problem, shown on the right, is obtained by exchanging the roles of x and t. Here the same network, again trained by minimizing the sum-of-squares error function, gives a poor fit to the data due to the multimodality of the data set.

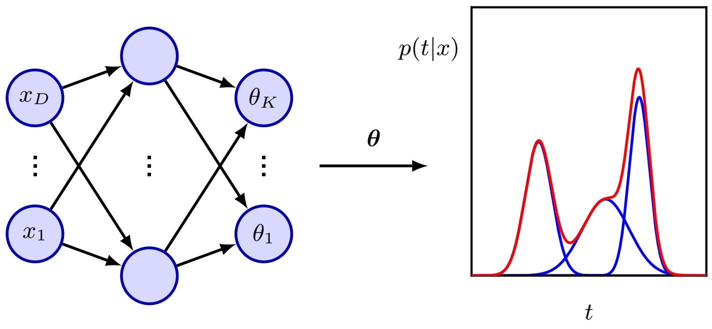


Figure 6.18 The *mixture density network* can represent general conditional probability densities $p(\mathbf{t}|\mathbf{x})$ by considering a parametric mixture model for the distribution of $\mathbf{t}$ whose parameters are determined by the outputs of a neural network that takes $\mathbf{x}$ as its input vector.

both the mixing coefficients as well as the component densities are flexible functions of the input vector $\mathbf{x}$, giving rise to a *mixture density network*. For any given value of $\mathbf{x}$, the mixture model provides a general formalism for modelling an arbitrary conditional density function $p(\mathbf{t}|\mathbf{x})$. Provided we consider a sufficiently flexible network, we then have a framework for approximating arbitrary conditional distributions.

Here we will develop the model explicitly for Gaussian components, so that

$$p(\mathbf{t}|\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\mathbf{x}) \mathcal{N}\left(\mathbf{t}|\boldsymbol{\mu}_k(\mathbf{x}), \sigma_k^2(\mathbf{x})\right). \tag{6.38}$$

This is an example of a *heteroscedastic* model in which the noise variance on the data is a function of the input vector $\mathbf{x}$. Instead of Gaussians, we can use other distributions for the components, such as Bernoulli distributions if the target variables are binary rather than continuous. We have also specialized to the case of isotropic covariances for the components, although the mixture density network can readily be extended to allow for general covariance matrices by representing the covariances using a Cholesky factorization (Williams, 1996). Even with isotropic components, the conditional distribution $p(\mathbf{t}|\mathbf{x})$ does not assume factorization with respect to the components of $\mathbf{t}$ (in contrast to the standard sum-of-squares regression model) as a consequence of the mixture distribution.

We now take the various parameters of the mixture model, namely the mixing coefficients $\pi_k(\mathbf{x})$, the means $\boldsymbol{\mu}_k(\mathbf{x})$, and the variances $\sigma_k^2(\mathbf{x})$, to be governed by the outputs of a neural network that takes $\mathbf{x}$ as its input. The structure of this mixture density network is illustrated in Figure 6.18. The mixture density network is closely related to the mixture-of-experts model (Jacobs *et al.*, 1991). The principal difference is that a mixture of experts has independent parameters for each component model in the mixture, whereas in a mixture density network, the same function is used to predict the parameters of all the component densities as well as the mixing coefficients, and so the nonlinear hidden units are shared amongst the input-dependent functions.

The neural network in Figure 6.18 can, for example, be a two-layer network having sigmoidal ($\tanh$) hidden units. If there are K components in the mixture model (6.38), and if $\mathbf{t}$ has L components, then the network will have K output-unit pre-activations denoted by a_k^π that determine the mixing coefficients $\pi_k(\mathbf{x})$, K outputs denoted by a_k^σ that determine the Gaussian standard deviations $\sigma_k(\mathbf{x})$, and $K \times L$ outputs denoted by a_{kj}^μ that determine the components $\mu_{kj}(\mathbf{x})$ of the Gaussian means $\boldsymbol{\mu}_k(\mathbf{x})$. The total number of network outputs is given by $(L+2)K$, unlike the usual L outputs for a network that simply predicts the conditional means of the target variables.

The mixing coefficients must satisfy the constraints

$$\sum_{k=1}^{K} \pi_k(\mathbf{x}) = 1, \qquad 0 \leqslant \pi_k(\mathbf{x}) \leqslant 1, \tag{6.39}$$

which can be achieved using a set of softmax outputs:

$$\pi_k(\mathbf{x}) = \frac{\exp(a_k^\pi)}{\sum_{l=1}^{K} \exp(a_l^\pi)}. \tag{6.40}$$

Similarly, the variances must satisfy $\sigma_k^2(\mathbf{x}) \geqslant 0$ and so can be represented in terms of the exponentials of the corresponding network pre-activations using

$$\sigma_k(\mathbf{x}) = \exp(a_k^\sigma). \tag{6.41}$$

Finally, because the means $\boldsymbol{\mu}_k(\mathbf{x})$ have real components, they can be represented directly by the network outputs:

$$\mu_{kj}(\mathbf{x}) = a_{kj}^\mu \tag{6.42}$$

in which the output-unit activation functions are given by the identity $f(a) = a$.

The learnable parameters of the mixture density network comprise the vector $\mathbf{w}$ of weights and biases in the neural network, which can be set by maximum likelihood or equivalently by minimizing an error function defined to be the negative logarithm of the likelihood. For independent data, this error function takes the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k(\mathbf{x}_n, \mathbf{w}) \mathcal{N}\left(\mathbf{t}_n | \boldsymbol{\mu}_k(\mathbf{x}_n, \mathbf{w}), \sigma_k^2(\mathbf{x}_n, \mathbf{w})\right) \right\} \tag{6.43}$$

where we have made the dependencies on $\mathbf{w}$ explicit.

6.5.3 Gradient optimization

To minimize the error function, we need to calculate the derivatives of the error $E(\mathbf{w})$ with respect to the components of $\mathbf{w}$. We will see later how to compute these derivatives automatically. It is instructive, however, to derive suitable expressions for the derivatives of the error with respect to the output-unit pre-activations explicitly as this highlights the probabilistic interpretation of these quantities. Because the error function (6.43) is composed of a sum of terms, one for each training data point, we can consider the derivatives for a particular input vector $\mathbf{x}_n$ with associated target vector $\mathbf{t}_n$. The derivatives of the total error E are obtained by summing over all

Chapter 8

data points, or the individual gradients for each data point can be used directly in gradient-based optimization algorithms. *Chapter 7*

It is convenient to introduce the following variables:

$$\gamma_{nk} = \gamma_k(\mathbf{t}_n|\mathbf{x}_n) = \frac{\pi_k \mathcal{N}_{nk}}{\sum_{l=1}^K \pi_l \mathcal{N}_{nl}} \tag{6.44}$$

where $\mathcal{N}_{nk}$ denotes $\mathcal{N}\left(\mathbf{t}_n|\boldsymbol{\mu}_k(\mathbf{x}_n), \sigma_k^2(\mathbf{x}_n)\right)$. These quantities have a natural interpretation as posterior probabilities for the components of the mixture in which the mixing coefficients $\pi_k(\mathbf{x})$ are viewed as $\mathbf{x}$-dependent prior probabilities. *Exercise 6.17*

The derivatives of the error function with respect to the network output pre-activations governing the mixing coefficients are given by *Exercise 6.18*

$$\frac{\partial E_n}{\partial a_k^{\pi}} = \pi_k - \gamma_{nk}. \tag{6.45}$$

Similarly, the derivatives with respect to the output pre-activations controlling the component means are given by *Exercise 6.19*

$$\frac{\partial E_n}{\partial a_{kl}^{\mu}} = \gamma_{nk}\left\{\frac{\mu_{kl} - t_{nl}}{\sigma_k^2}\right\}. \tag{6.46}$$

Finally, the derivatives with respect to the output pre-activations controlling the component variances are given by *Exercise 6.20*

$$\frac{\partial E_n}{\partial a_k^{\sigma}} = \gamma_{nk}\left\{L - \frac{\|\mathbf{t}_n - \boldsymbol{\mu}_k\|^2}{\sigma_k^2}\right\}. \tag{6.47}$$

6.5.4 Predictive distribution

We illustrate the use of a mixture density network by returning to the toy example of an inverse problem shown in Figure 6.17. Plots of the mixing coefficients $\pi_k(x)$, the means $\mu_k(x)$, and the conditional density contours corresponding to $p(t|x)$, are shown in Figure 6.19. The outputs of the neural network, and hence the parameters in the mixture model, are necessarily continuous single-valued functions of the input variables. However, we see from Figure 6.19(c) that the model is able to produce a conditional density that is unimodal for some values of x and trimodal for other values by modulating the amplitudes of the mixing components $\pi_k(\mathbf{x})$.

Once a mixture density network has been trained, it can predict the conditional density function of the target data for any given value of the input vector. This conditional density represents a complete description of the generator of the data, so far as the problem of predicting the value of the output vector is concerned. From this density function, we can calculate more specific quantities that may be of interest in different applications. One of the simplest of these is the mean, corresponding to the conditional average of the target data, and is given by

$$\mathbb{E}\left[\mathbf{t}|\mathbf{x}\right] = \int \mathbf{t} p(\mathbf{t}|\mathbf{x})\,\mathrm{d}\mathbf{t} = \sum_{k=1}^{K} \pi_k(\mathbf{x})\boldsymbol{\mu}_k(\mathbf{x}) \tag{6.48}$$

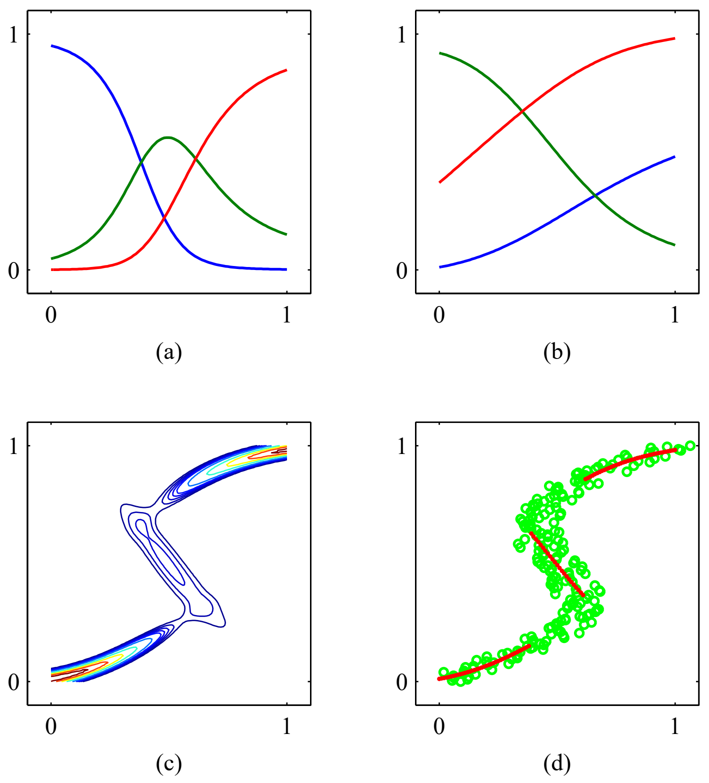


Figure 6.19 (a) Plot of the mixing coefficients $\pi_k(x)$ as a function of x for the three mixture components in a mixture density network trained on the data shown in Figure 6.17. The model has three Gaussian components and uses a two-layer neural network with five $\tanh$ sigmoidal units in the hidden layer and nine outputs (corresponding to the three means and three variances of the Gaussian components and the three mixing coefficients). At both small and large values of x, where the conditional probability density of the target data is unimodal, only one of the Gaussian components has a high value for its prior probability, whereas at intermediate values of x, where the conditional density is trimodal, the three mixing coefficients have comparable values. (b) Plots of the means $\mu_k(x)$ using the same colour coding as for the mixing coefficients. (c) Plot of the contours of the corresponding conditional probability density of the target data for the same mixture density network. (d) Plot of the approximate conditional mode, shown by the red points, of the conditional density.

where we have used (6.38). Because a standard network trained by least squares approximates the conditional mean, we see that a mixture density network can reproduce the conventional least-squares result as a special case. Of course, as we have already noted, for a multimodal distribution the conditional mean is of limited value.

Exercise 6.21

We can similarly evaluate the variance of the density function about the conditional average, to give

$$s^2(\mathbf{x}) = \mathbb{E}\left[\|\mathbf{t} - \mathbb{E}[\mathbf{t}|\mathbf{x}]\|^2 \,|\mathbf{x}\right] \tag{6.49}$$

$$= \sum_{k=1}^{K} \pi_k(\mathbf{x}) \left\{ \sigma_k^2(\mathbf{x}) + \left\| \boldsymbol{\mu}_k(\mathbf{x}) - \sum_{l=1}^{K} \pi_l(\mathbf{x}) \boldsymbol{\mu}_l(\mathbf{x}) \right\|^2 \right\} \tag{6.50}$$

where we have used (6.38) and (6.48). This is more general than the corresponding least-squares result because the variance is a function of $\mathbf{x}$.

We have seen that for multimodal distributions, the conditional mean can give a poor representation of the data. For instance, in controlling the simple robot arm shown in Figure 6.16, we need to pick one of the two possible joint angle settings to achieve the desired end-effector location, but the average of the two solutions is not itself a solution. In such cases, the conditional mode may be of more value. Because the conditional mode for the mixture density network does not have a simple analytical solution, a numerical iteration is required. A simple alternative is to take the mean of the most probable component (i.e., the one with the largest mixing coefficient) at each value of $\mathbf{x}$. This is shown for the toy data set in Figure 6.19(d).

Exercises

6.1 ($\star\star\star$) Use the result (2.126) to derive an expression for the surface area S_D and the volume V_D of a hypersphere of unit radius in D dimensions. To do this, consider the following result, which is obtained by transforming from Cartesian to polar coordinates:

$$\prod_{i=1}^{D} \int_{-\infty}^{\infty} e^{-x_i^2} \,\mathrm{d}x_i = S_D \int_0^{\infty} e^{-r^2} r^{D-1} \,\mathrm{d}r. \tag{6.51}$$

Using the gamma function, defined by

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} \,\mathrm{d}t \tag{6.52}$$

together with (2.126), evaluate both sides of this equation, and hence show that

$$S_D = \frac{2\pi^{D/2}}{\Gamma(D/2)}. \tag{6.53}$$

Next, by integrating with respect to the radius from 0 to 1, show that the volume of the unit hypersphere in D dimensions is given by

$$V_D = \frac{S_D}{D}. \tag{6.54}$$

Finally, use the results $\Gamma(1) = 1$ and $\Gamma(3/2) = \sqrt{\pi}/2$ to show that (6.53) and (6.54) reduce to the usual expressions for $D = 2$ and $D = 3$.

6.2 ($\star\star\star$) Consider a hypersphere of radius a in D dimensions together with the concentric hypercube of side $2a$, so that the hypersphere touches the hypercube at the centres of each of its sides. By using the results of Exercise 6.1, show that the ratio of the volume of the hypersphere to the volume of the cube is given by

$$\frac{\text{volume of hypersphere}}{\text{volume of cube}} = \frac{\pi^{D/2}}{D 2^{D-1} \Gamma(D/2)}. \tag{6.55}$$

Now make use of Stirling's formula in the form

$$\Gamma(x+1) \simeq (2\pi)^{1/2} e^{-x} x^{x+1/2}, \tag{6.56}$$

which is valid for $x \gg 1$, to show that, as $D \to \infty$, the ratio (6.55) goes to zero. Show also that the distance from the centre of the hypercube to one of the corners divided by the perpendicular distance to one of the sides is $\sqrt{D}$, which therefore goes to ∞ as $D \to \infty$. From these results, we see that, in a space of high dimensionality, most of the volume of a cube is concentrated in the large number of corners, which themselves become very long 'spikes'!

6.3 ($\star\star\star$) In this exercise, we explore the behaviour of the Gaussian distribution in high-dimensional spaces. Consider a Gaussian distribution in D dimensions given by

$$p(\mathbf{x}) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{\|\mathbf{x}\|^2}{2\sigma^2}\right). \tag{6.57}$$

We wish to find the density as a function of the radius in polar coordinates in which the direction variables have been integrated out. To do this, show that the integral of the probability density over a thin shell of radius r and thickness ϵ, where $\epsilon \ll 1$, is given by $p(r)\epsilon$ where

$$p(r) = \frac{S_D r^{D-1}}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{r^2}{2\sigma^2}\right) \tag{6.58}$$

where S_D is the surface area of a unit hypersphere in D dimensions. Show that the function $p(r)$ has a single stationary point located, for large D, at $\widehat{r} \simeq \sqrt{D}\sigma$. By considering $p(\widehat{r} + \epsilon)$ where $\epsilon \ll \widehat{r}$, show that for large D,

$$p(\widehat{r} + \epsilon) = p(\widehat{r}) \exp\left(-\frac{3\epsilon^2}{2\sigma^2}\right), \tag{6.59}$$

which shows that $\widehat{r}$ is a maximum of the radial probability density and also that $p(r)$ decays exponentially away from its maximum at $\widehat{r}$ with length scale σ. We have already seen that $\sigma \ll \widehat{r}$ for large D, and so we see that most of the probability mass is concentrated in a thin shell at large radius. Finally, show that the probability density $p(\mathbf{x})$ is larger at the origin than at the radius $\widehat{r}$ by a factor of $\exp(D/2)$. We therefore see that most of the probability mass in a high-dimensional Gaussian distribution is located at a different radius from the region of high probability density.

6.4 ($\star\star$) Consider a two-layer network function of the form (6.11) in which the hidden-unit nonlinear activation functions $h(\cdot)$ are given by logistic sigmoid functions of the form

$$\sigma(a) = \{1 + \exp(-a)\}^{-1}. \tag{6.60}$$

Show that there exists an equivalent network, which computes exactly the same function, but with hidden-unit activation functions given by $\tanh(a)$ where the tanh function is defined by (6.14). Hint: first find the relation between $\sigma(a)$ and $\tanh(a)$, and then show that the parameters of the two networks differ by linear transformations.

6.5 ($\star\star$) The *swish* activation function (Ramachandran, Zoph, and Le, 2017) is defined by

$$h(x) = x\sigma(\beta x) \tag{6.61}$$

where $\sigma(x)$ is the logistic-sigmoid activation function defined by (6.13). When used in a neural network, β can be treated as a learnable parameter. Either sketch or plot using software graphs of the swish activation function as well as its first derivative for $\beta = 0.1$, $\beta = 1.0$, and $\beta = 10$. Show that when $\beta \to \infty$, the swish function becomes the ReLU function.

6.6 ($\star$) We saw in (5.72) that the derivative of the logistic-sigmoid activation function can be expressed in terms of the function value itself. Derive the corresponding result for the $\tanh$ activation function defined by (6.14).

6.7 ($\star\star$) Show that the softplus activation function $\zeta(a)$ given by (6.16) satisfies the properties:

$$\zeta(a) - \zeta(-a) = a \tag{6.62}$$

$$\ln \sigma(a) = -\zeta(-a) \tag{6.63}$$

$$\frac{\mathrm{d}\zeta(a)}{\mathrm{d}a} = \sigma(a) \tag{6.64}$$

$$\zeta^{-1}(a) = \ln\left(\exp(a) - 1\right) \tag{6.65}$$

where $\sigma(a)$ is the logistic-sigmoid activation function given by (6.13).

6.8 ($\star$) Show that minimization of the error function (6.25) with respect to the variance σ^2 gives the result (6.27).

6.9 ($\star$) Show that maximizing the likelihood function under the conditional distribution (6.28) for a multioutput neural network is equivalent to minimizing the sum-of-squares error function (6.29). Also, show that the noise variance that minimizes this error function is given by (6.30).

6.10 ($\star\star$) Consider a regression problem involving multiple target variables in which it is assumed that the distribution of the targets, conditioned on the input vector $\mathbf{x}$, is a Gaussian of the form

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \boldsymbol{\Sigma}) \tag{6.66}$$

where $\mathbf{y}(\mathbf{x}, \mathbf{w})$ is the output of a neural network with input vector $\mathbf{x}$ and weight vector $\mathbf{w}$, and $\boldsymbol{\Sigma}$ is the covariance of the assumed Gaussian noise on the targets. Given a set of independent observations of $\mathbf{x}$ and $\mathbf{t}$, write down the error function that must be minimized to find the maximum likelihood solution for $\mathbf{w}$, if we assume that $\boldsymbol{\Sigma}$ is fixed and known. Now assume that $\boldsymbol{\Sigma}$ is also to be determined from the data, and write down an expression for the maximum likelihood solution for $\boldsymbol{\Sigma}$. Note that the optimizations of $\mathbf{w}$ and $\boldsymbol{\Sigma}$ are now coupled, in contrast to the case of independent target variables discussed in Section 6.4.1.

6.11 ($\star\star$) Consider a binary classification problem in which the target values are $t \in \{0, 1\}$, with a network output $y(\mathbf{x}, \mathbf{w})$ that represents $p(t = 1|\mathbf{x})$, and suppose that there is a probability ϵ that the class label on a training data point has been incorrectly set. Assuming i.i.d. data, write down the error function corresponding to the negative log likelihood. Verify that the error function (6.33) is obtained when $\epsilon = 0$. Note that this error function makes the model robust to incorrectly labelled data, in contrast to the usual cross-entropy error function.

6.12 ($\star\star$) The error function (6.33) for binary classification problems was derived for a network having a logistic-sigmoid output activation function, so that $0 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$, and data having target values $t \in \{0, 1\}$. Derive the corresponding error function if we consider a network having an output $-1 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$ and target values $t = 1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$. What would be the appropriate choice of output-unit activation function?

6.13 ($\star$) Show that maximizing the likelihood for a multi-class neural network model in which the network outputs have the interpretation $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$ is equivalent to minimizing the cross-entropy error function (6.36).

6.14 ($\star$) Show that the derivative of the error function (6.33) with respect to the pre-activation a_k for an output unit having a logistic-sigmoid activation function $y_k = \sigma(a_k)$, where $\sigma(a)$ is given by (6.13), satisfies (6.31).

6.15 ($\star$) Show that the derivative of the error function (6.36) with respect to the pre-activation a_k for output units having a softmax activation function (6.37) satisfies (6.31).

6.16 ($\star\star$) Write down a pair of equations that express the Cartesian coordinates (x_1, x_2) for the robot arm shown in Figure 6.16 in terms of the joint angles θ_1 and θ_2 and the lengths L_1 and L_2 of the links. Assume the origin of the coordinate system is given by the attachment point of the lower arm. These equations define the forward kinematics of the robot arm.

6.17 ($\star\star$) Show that the variable γ_{nk} defined by (6.44) can be viewed as the posterior probabilities $p(k|\mathbf{t})$ for the components of the mixture distribution (6.38) in which the mixing coefficients $\pi_k(\mathbf{x})$ are viewed as $\mathbf{x}$-dependent prior probabilities $p(k)$.

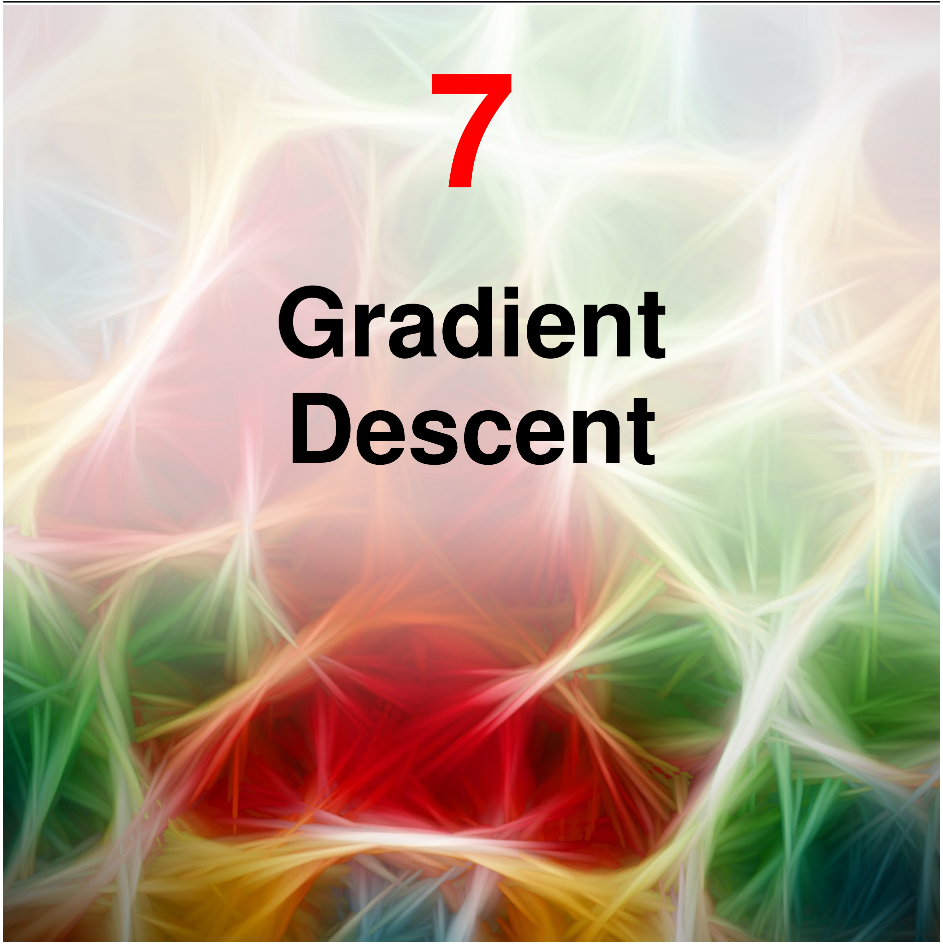
6.18 ($\star\star$) Derive the result (6.45) for the derivative of the error function with respect to the network output pre-activations controlling the mixing coefficients in the mixture density network.

6.19 ($\star\star$) Derive the result (6.46) for the derivative of the error function with respect to the network output pre-activations controlling the component means in the mixture density network.

6.20 ($\star\star$) Derive the result (6.47) for the derivative of the error function with respect to the network output pre-activations controlling the component variances in the mixture density network.

6.21 ($\star\star\star$) Verify the results (6.48) and (6.50) for the conditional mean and variance of the mixture density network model.

7 Gradient Descent



In the previous chapter we saw that neural networks are a very broad and flexible class of functions and are able in principle to approximate any desired function to arbitrarily high accuracy given a sufficiently large number of hidden units. Moreover, we saw that deep neural networks can encode inductive biases corresponding to hierarchical representations, which prove valuable in a wide range of practical applications. We now turn to the task of finding a suitable setting for the network parameters (weights and biases), based on a set of training data.

As with the regression and classification models discussed in earlier chapters, we choose the model parameters by optimizing an error function. We have seen how to define a suitable error function for a particular application by using maximum likelihood. Although in principle the error function could be minimized numerically through a series of direct error function evaluations, this turns out to be very inefficient. Instead, we turn to another core concept that is used in deep learning, which

Section 6.4

is that optimizing the error function can be done much more efficiently by making use of gradient information, in other words by evaluating the derivatives of the error function with respect to the network parameters. This is why we took care to ensure that the function represented by the neural network is differentiable by design. Likewise, the error function itself also needs to be differentiable.

Chapter 8

The required derivatives of the error function with respect to each of the parameters in the network can be evaluated efficiently using a technique called *backpropagation*, which involves successive computations that flow backwards through the network in a way that is analogous to the forward flow of function computations during the evaluation of the network outputs.

Section 9.3.2

Chapter 9

Although the likelihood is used to define an error function, the goal when optimizing the error function in a neural network is to achieve good generalization on test data. In classical statistics, maximum likelihood is used to fit a parametric model to a finite data set, in which the number of data points typically far exceeds the number of parameters in the model. The optimal solution has the maximum value of the likelihood function, and the values found for the fitted parameters are of direct interest. By contrast, modern deep learning works with very rich models containing huge numbers of learnable parameters, and the goal is never simply exact optimization. Instead, the properties and behaviour of the learning algorithm itself, along with various methods for regularization, are important in determining how well the solution generalizes to new data.

7.1. Error Surfaces

Our goal during training is to find values for the weights and biases in the neural network that will allow it to make effective predictions. For convenience we will group these parameters into a single vector $\mathbf{w}$, and we will optimize $\mathbf{w}$ by using a chosen error function $E(\mathbf{w})$. At this point, it is useful to have a geometrical picture of the error function, which we can view as a surface sitting over 'weight space', as shown in Figure 7.1.

First note that if we make a small step in weight space from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ then the change in the error function is given by

$$\delta E \simeq \delta\mathbf{w}^{\mathrm{T}}\nabla E(\mathbf{w}) \tag{7.1}$$

where the vector $\nabla E(\mathbf{w})$ points in the direction of the greatest rate of increase of the error function. Provided the error $E(\mathbf{w})$ is a smooth, continuous function of $\mathbf{w}$, its smallest value will occur at a point in weight space such that the gradient of the error function vanishes, so that

$$\nabla E(\mathbf{w}) = 0 \tag{7.2}$$

as otherwise we could make a small step in the direction of $-\nabla E(\mathbf{w})$ and thereby further reduce the error. Points at which the gradient vanishes are called stationary points and may be further classified into minima, maxima, and saddle points.

Section 7.1.1

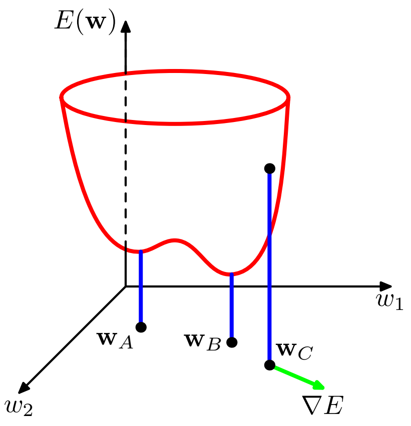


Figure 7.1 Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum, so that $E(\mathbf{w}_\mathrm{A}) > E(\mathbf{w}_\mathrm{B})$. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.

We will aim to find a vector $\mathbf{w}$ such that $E(\mathbf{w})$ takes its smallest value. However, the error function typically has a highly nonlinear dependence on the weights and bias parameters, and so there will be many points in weight space at which the gradient vanishes (or is numerically very small). Indeed, for any point $\mathbf{w}$ that is a local minimum, there will generally be other points in weight space that are equivalent minima. For instance, in a two-layer network of the kind shown in Figure 6.9, with M hidden units, each point in weight space is a member of a family of $M!\,2^M$ equivalent points. *Section 6.2.4*

Furthermore, there may be multiple non-equivalent stationary points and in particular multiple non-equivalent minima. A minimum that corresponds to the smallest value of the error function across the whole of $\mathbf{w}$-space is said to be a *global minimum*. Any other minima corresponding to higher values of the error function are said to be *local minima*. The error surfaces for deep neural networks can be very complex, and it was thought that gradient-based methods might become trapped in poor local minima. In practice, this seems not to be the case, and large networks can reach solutions with similar performance under a variety of initial conditions. *Section 9.3.2*

7.1.1 Local quadratic approximation

Insight into the optimization problem and into the various techniques for solving it can be obtained by considering a local quadratic approximation to the error function. The Taylor expansion of $E(\mathbf{w})$ around some point $\widehat{\mathbf{w}}$ in weight space is given by

$$E(\mathbf{w}) \simeq E(\widehat{\mathbf{w}}) + (\mathbf{w}-\widehat{\mathbf{w}})^\mathrm{T}\mathbf{b} + \frac{1}{2}(\mathbf{w}-\widehat{\mathbf{w}})^\mathrm{T}\mathbf{H}(\mathbf{w}-\widehat{\mathbf{w}}) \tag{7.3}$$

where cubic and higher terms have been omitted. Here $\mathbf{b}$ is defined to be the gradient of E evaluated at $\widehat{\mathbf{w}}$

$$\mathbf{b} \equiv \nabla E|_{\mathbf{w}=\widehat{\mathbf{w}}}\,. \tag{7.4}$$

The *Hessian* is defined to be the corresponding matrix of second derivatives

$$\mathbf{H}(\widehat{\mathbf{w}}) = \nabla\nabla E(\mathbf{w})|_{\mathbf{w}=\widehat{\mathbf{w}}}\,. \tag{7.5}$$

If there is a total of W weights and biases in the network, then $\mathbf{w}$ and $\mathbf{b}$ have length W and $\mathbf{H}$ has dimensionality $W \times W$. From (7.3), the corresponding local approximation to the gradient is given by

$$\nabla E(\mathbf{w}) = \mathbf{b} + \mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}). \tag{7.6}$$

For points $\mathbf{w}$ that are sufficiently close to $\widehat{\mathbf{w}}$, these expressions will give reasonable approximations for the error and its gradient.

Consider the particular case of a local quadratic approximation around a point $\mathbf{w}^\star$ that is a minimum of the error function. In this case there is no linear term, because $\nabla E = 0$ at $\mathbf{w}^\star$, and (7.3) becomes

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^\star) \tag{7.7}$$

where the Hessian $\mathbf{H}$ is evaluated at $\mathbf{w}^\star$. To interpret this geometrically, consider the eigenvalue equation for the Hessian matrix:

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{7.8}$$

Appendix A

where the eigenvectors $\mathbf{u}_i$ form a complete orthonormal set so that

$$\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}. \tag{7.9}$$

We now expand $(\mathbf{w} - \mathbf{w}^\star)$ as a linear combination of the eigenvectors in the form

$$\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i. \tag{7.10}$$

Appendix A

Exercise 7.1

This can be regarded as a transformation of the coordinate system in which the origin is translated to the point $\mathbf{w}^\star$ and the axes are rotated to align with the eigenvectors through the orthogonal matrix whose columns are $\{\mathbf{u}_1, \ldots, \mathbf{u}_W\}$. By substituting (7.10) into (7.7) and using (7.8) and (7.9), the error function can be written in the form

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2. \tag{7.11}$$

Suppose we set all $\alpha_i = 0$ for $i \neq j$ and then vary α_j, corresponding to moving $\mathbf{w}$ away from $\mathbf{w}^\star$ in the direction of $\mathbf{u}_j$. We see from (7.11) that the error function will increase if the corresponding eigenvalue λ_j is positive and will decrease if it is negative. If all eigenvalues are positive then $\mathbf{w}^\star$ corresponds to a local minimum of the error function, whereas if they are all negative then $\mathbf{w}^\star$ corresponds to a local maximum. If we have a mix of positive and negative eigenvalues then $\mathbf{w}^\star$ represents a saddle point.

A matrix $\mathbf{H}$ is said to be *positive definite* if, and only if,

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} > 0, \qquad \text{for all } \mathbf{v}. \tag{7.12}$$

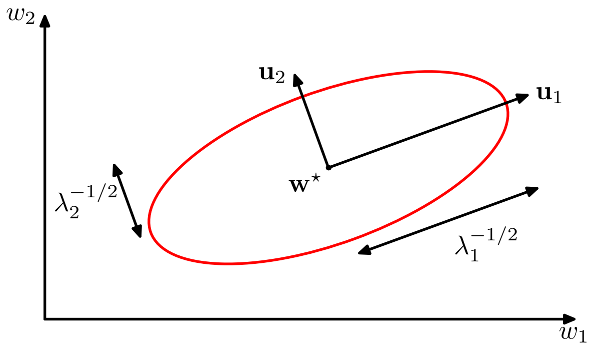


Figure 7.2 In the neighbourhood of a minimum $\mathbf{w}^\star$, the error function can be approximated by a quadratic. Contours of constant error are then ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ of the Hessian matrix, with lengths that are inversely proportional to the square roots of the corresponding eigenvectors λ_i.

Because the eigenvectors $\{\mathbf{u}_i\}$ form a complete set, an arbitrary vector $\mathbf{v}$ can be written in the form

$$\mathbf{v} = \sum_i c_i \mathbf{u}_i. \tag{7.13}$$

From (7.8) and (7.9), we then have

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} = \sum_i c_i^2 \lambda_i \tag{7.14}$$

Exercise 7.2

Exercise 7.3

Exercise 7.6

and so $\mathbf{H}$ will be positive definite if, and only if, all its eigenvalues are positive. Thus, a necessary and sufficient condition for $\mathbf{w}^\star$ to be a local minimum is that the gradient of the error function should vanish at $\mathbf{w}^\star$ and the Hessian matrix evaluated at $\mathbf{w}^\star$ should be positive definite. In the new coordinate system, whose basis vectors are given by the eigenvectors $\{\mathbf{u}_i\}$, the contours of constant $E(\mathbf{w})$ are axis-aligned ellipses centred on the origin, as illustrated in Figure 7.2.

7.2. Gradient Descent Optimization

There is little hope of finding an analytical solution to the equation $\nabla E(\mathbf{w}) = 0$ for an error function as complex as one defined by a neural network, and so we resort to iterative numerical procedures. The optimization of continuous nonlinear functions is a widely studied problem, and there exists an extensive literature on how to solve it efficiently. Most techniques involve choosing some initial value $\mathbf{w}^{(0)}$ for the weight vector and then moving through weight space in a succession of steps of the form

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} + \Delta\mathbf{w}^{(\tau-1)} \tag{7.15}$$

where τ labels the iteration step. Different algorithms involve different choices for the weight vector update $\Delta\mathbf{w}^{(\tau)}$.

Because of the complex shape of the error surface for all but the simplest neural networks, the solution found will depend, among other things, on the particular choice of initial parameter values $\mathbf{w}^{(0)}$. To find a sufficiently good solution, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point, and comparing the resulting performance on an independent validation set.

7.2.1 Use of gradient information

Chapter 8

The gradient of an error function for a deep neural network can be evaluated efficiently using the technique of error backpropagation, and applying this gradient information can lead to significant improvements in the speed of network training. We can see why this is so, as follows.

Exercise 7.7

In the quadratic approximation to the error function given by (7.3), the error surface is specified by the quantities $\mathbf{b}$ and $\mathbf{H}$, which contain a total of $W(W+3)/2$ independent elements (because the matrix $\mathbf{H}$ is symmetric), where W is the dimensionality of $\mathbf{w}$ (i.e., the total number of learnable parameters in the network). The location of the minimum of this quadratic approximation therefore depends on $\mathcal{O}(W^2)$ parameters, and we should not expect to be able to locate the minimum until we have gathered $\mathcal{O}(W^2)$ independent pieces of information. If we do not make use of gradient information, we would expect to have to perform $\mathcal{O}(W^2)$ function evaluations, each of which would require $\mathcal{O}(W)$ steps. Thus, the computational effort needed to find the minimum using such an approach would be $\mathcal{O}(W^3)$.

Chapter 8

Now compare this with an algorithm that makes use of the gradient information. Because ∇E is a vector of length W, each evaluation of ∇E brings W pieces of information, and so we might hope to find the minimum of the function in $\mathcal{O}(W)$ gradient evaluations. As we shall see, by using error backpropagation, each such evaluation takes only $\mathcal{O}(W)$ steps and so the minimum can now be found in $\mathcal{O}(W^2)$ steps. Although the quadratic approximation only holds in the neighbourhood of a minimum, the efficiency gains are generic. For this reason, the use of gradient information forms the basis of all practical algorithms for training neural networks.

7.2.2 Batch gradient descent

The simplest approach to using gradient information is to choose the weight update in (7.15) such that there is a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla E(\mathbf{w}^{(\tau-1)}) \tag{7.16}$$

where the parameter $\eta > 0$ is known as the *learning rate*. After each such update, the gradient is re-evaluated for the new weight vector $\mathbf{w}^{(\tau+1)}$ and the process repeated. At each step, the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as *gradient descent* or *steepest descent*. Note that the error function is defined with respect to a training set, and so to evaluate ∇E, each step requires that the entire training set be processed. Techniques that use the whole data set at once are called *batch* methods.

7.2.3 Stochastic gradient descent

Deep learning methods benefit greatly from very large data sets. However, batch methods can become extremely inefficient if there are many data points in the training set because each error function or gradient evaluation requires the entire data set

Algorithm 7.1: Stochastic gradient descent

```
Input: Training set of data points indexed by $n \in \{1, \dots, N\}$
       Error function per data point $E_n(\mathbf{w})$
       Learning rate parameter $\eta$
       Initial weight vector $\mathbf{w}$
Output: Final weight vector $\mathbf{w}$
$n \leftarrow 1$
repeat
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_n(\mathbf{w})$ // update weight vector
    $n \leftarrow n + 1 (\mathrm{mod}\ N)$ // iterate over data
until convergence
return $\mathbf{w}$
```

to be processed. To find a more efficient approach, note that error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{7.17}$$

The most widely used training algorithms for large data sets are based on a sequential version of gradient descent known as *stochastic gradient descent* (Bottou, 2010), or SGD, which updates the weight vector based on one data point at a time, so that

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla E_n(\mathbf{w}^{(\tau-1)}). \tag{7.18}$$

This update is repeated by cycling through the data. A complete pass through the whole training set is known as a training *epoch*. This technique is also known as *online gradient descent*, especially if the data arises from a continuous stream of new data points. Stochastic gradient descent is summarized in Algorithm 7.1.

A further advantage of stochastic gradient descent, compared to batch gradient descent, is that it handles redundancy in the data much more efficiently. To see this, consider an extreme example in which we take a data set and double its size by duplicating every data point. Note that this simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function, if the value of the learning rate is adjusted to compensate. Batch methods will require double the computational effort to evaluate the batch error function gradient, whereas stochastic gradient descent will be unaffected. Another property of stochastic gradient descent is the possibility of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

7.2.4 Mini-batches

A downside of stochastic gradient descent is that the gradient of the error function computed from a single data point provides a very noisy estimate of the gradient of the error function computed on the full data set. We can consider an intermediate approach in which a small subset of data points, called a *mini-batch*, is used to evaluate the gradient at each iteration. In determining the optimum size for the mini-batch, note that the error in computing the mean from N samples is given by $\sigma/\sqrt{N}$ where σ is the standard deviation of the distribution generating the data. This indicates that there are diminishing returns in estimating the true gradient from increasing the batch size. If we increase the size of the mini-batch by a factor of 100 then the error only reduces by a factor of 10. Another consideration in choosing the mini-batch size is the desire to make efficient use of the hardware architecture on which the code is running. For example, on some hardware platforms, mini-batch sizes that are powers of 2 (for example, 64, 128, 256, . . .) work well.

Exercise 7.8

One important consideration when using mini-batches is that the constituent data points should be chosen randomly from the data set, since in raw data sets there may be correlations between successive data points arising from the way the data was collected (for example, if the data points have been ordered alphabetically or by date). This is often handled by randomly shuffling the entire data set and then subsequently drawing mini-batches as successive blocks of data. The data set can also be reshuffled between iterations through the data set, so that each mini-batch is unlikely to have been used before, which can help escape local minima. The variant of stochastic gradient descent with mini-batches is summarized in Algorithm 7.2. Note that the learning algorithm is often still called 'stochastic gradient descent' even when mini-batches are used.

7.2.5 Parameter initialization

Iterative algorithms such as gradient descent require that we choose some initial setting for the parameters being learned. The specific initialization can have a significant effect on how long it takes to reach a solution and on the generalization performance of the resulting trained network. Unfortunately, there is relatively little theory to guide the initialization strategy.

One key consideration, however, is *symmetry breaking*. Consider a set of hidden units or output units that take the same inputs. If the parameters were all initialized with the same value, for example if they were all set to zero, the parameters of these units would all be updated in unison and the units would each compute the same function and hence be redundant. This problem can be addressed by initializing parameters randomly from some distribution to break symmetry. If computational resources permit, the network might be trained multiple times starting from different random initializations and the results compared on held-out data.

The distribution used to initialize the weights is typically either a uniform distribution in the range $[-\epsilon, \epsilon]$ or a zero-mean Gaussian of the form $\mathcal{N}(0, \epsilon^2)$. The choice of the value of ϵ is important, and various heuristics to select it have been proposed. One widely used approach is called *He initialization* (He *et al.*, 2015b). Consider a

Algorithm 7.2: Mini-batch stochastic gradient descent

```
Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
       Batch size $B$
       Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
       Learning rate parameter $\eta$
       Initial weight vector $\mathbf{w}$
Output: Final weight vector $\mathbf{w}$
----------------------------------------------------------------
$n \leftarrow 1$
repeat
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_{n:n+B-1}(\mathbf{w})$ // weight vector update
    $n \leftarrow n + B$
    if $n > N$ then
        shuffle data
        $n \leftarrow 1$
    end if
until convergence
return $\mathbf{w}$
```

network in which layer l evaluates the following transformations

$$a_i^{(l)} = \sum_{j=1}^{M} w_{ij} z_j^{(l-1)} \tag{7.19}$$

$$z_i^{(l)} = \mathrm{ReLU}(a_i^{(l)}) \tag{7.20}$$

where M is the number of units that send connections to unit i, and the ReLU activation function is given by (6.17). Suppose we initialize the weights using a Gaussian $\mathcal{N}(0, \epsilon^2)$, and suppose that the outputs $z_j^{(l-1)}$ of the units in layer $l-1$ have variance λ^2. Then we can easily show that

Exercise 7.9

$$\mathbb{E}[a_i^{(l)}] = 0 \tag{7.21}$$

$$\mathrm{var}[z_j^{(l)}] = \frac{M}{2}\epsilon^2\lambda^2 \tag{7.22}$$

where the factor of $1/2$ arises from the ReLU activation function. Ideally we want to ensure that the variance of the pre-activations neither decays to zero nor grows significantly as we propagate from one layer to the next. If we therefore require that the units at layer l also have variance λ^2 then we arrive at the following choice for the standard deviation of the Gaussian used to initialize the weights that feed into a

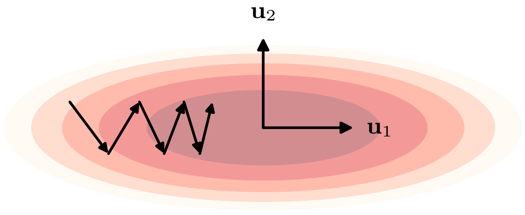


Figure 7.3 Schematic illustration of fixed-step gradient descent for an error function that has substantially different curvatures along different directions. The error surface E has the form of a long valley, as depicted by the ellipses. Note that, for most points in weight space, the local negative gradient vector $-\nabla E$ does not point towards the minimum of the error function. Successive steps of gradient descent can therefore oscillate across the valley, leading to very slow progress along the valley towards the minimum. The vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ are the eigenvectors of the Hessian matrix.

unit with M inputs:

$$\epsilon = \sqrt{\frac{2}{M}}. \tag{7.23}$$

It is also possible to treat the scale ϵ of the initialization distribution as a hyperparameter and to explore different values across multiple training runs. The bias parameters are typically set to small positive values to ensure that most pre-activations are initially active during learning. This is particularly helpful with ReLU units, where we want the pre-activations to be positive so that there is a non-zero gradient to drive learning.

Another important class of techniques for initializing the parameters of a neural network is by using the values that result from training the network on a different task or by exploiting various forms of unsupervised training. These techniques fall into the broad class of *transfer learning* techniques. *Section 6.3.4*

7.3. Convergence

When applying gradient descent in practice, we need to choose a value for the learning rate parameter η. Consider the simple error surface depicted in Figure 7.3 for a hypothetical two-dimensional weight space in which the curvature of E varies significantly with direction, creating a 'valley'. At most points on the error surface, the local gradient vector for batch gradient descent, which is perpendicular to the local contour, does not point directly towards the minimum. Intuitively we might expect that increasing the value of η should lead to bigger steps through weight space and hence faster convergence. However, the successive steps oscillate back and forth across the valley, and if we increase η too much, those oscillations will become divergent. Because η must be kept sufficiently small to avoid divergent oscillations across the valley, progress along the valley is very slow. Gradient descent then takes many small steps to reach the minimum and is a very inefficient procedure.

We can gain deeper insight into the nature of this problem by considering the quadratic approximation to the error function in the neighbourhood of the minimum. *Section 7.1.1* From (7.7), (7.8), and (7.10), the gradient of the error function in this approximation can be written as

$$\nabla E = \sum_i \alpha_i \lambda_i \mathbf{u}_i. \tag{7.24}$$

Again using (7.10) we can express the change in the weight vector in terms of corresponding changes in the coefficients $\{\alpha_i\}$:

$$\Delta \mathbf{w} = \sum_i \Delta\alpha_i \mathbf{u}_i. \tag{7.25}$$

Combining (7.24) with (7.25) and the gradient descent formula (7.16) and using the orthonormality relation (7.9) for the eigenvectors of the Hessian, we obtain the following expression for the change in α_i at each step of the gradient descent algorithm:

$$\Delta\alpha_i = -\eta\lambda_i\alpha_i \tag{7.26}$$

Exercise 7.10

from which it follows that

$$\alpha_i^{\text{new}} = (1-\eta\lambda_i)\alpha_i^{\text{old}} \tag{7.27}$$

where 'old' and 'new' denote values before and after a weight update. Using the orthonormality relation (7.9) for the eigenvectors together with (7.10), we have

$$\mathbf{u}_i^{\mathrm{T}}(\mathbf{w}-\mathbf{w}^\star) = \alpha_i \tag{7.28}$$

and so α_i can be interpreted as the distance to the minimum along the direction $\mathbf{u}_i$. From (7.27) we see that these distances evolve independently such that, at each step, the distance along the direction of $\mathbf{u}_i$ is multiplied by a factor $(1-\eta\lambda_i)$. After a total of T steps we have

$$\alpha_i^{(T)} = (1-\eta\lambda_i)^T \alpha_i^{(0)}. \tag{7.29}$$

It follows that, provided $|1-\eta\lambda_i| < 1$, the limit $T \to \infty$ leads to $\alpha_i = 0$, which from (7.28) shows that $\mathbf{w} = \mathbf{w}^\star$ and so the weight vector has reached the minimum of the error.

Note that (7.29) demonstrates that gradient descent leads to linear convergence in the neighbourhood of a minimum. Also, convergence to the stationary point requires that all the λ_i be positive, which in turn implies that the stationary point is indeed a minimum. By making η larger we can make the factor $(1-\eta\lambda_i)$ smaller and hence improve the speed of convergence. There is a limit to how large η can be made, however. We can permit $(1-\eta\lambda_i)$ to go negative (which gives oscillating values of α_i), but we must ensure that $|1-\eta\lambda_i| < 1$ otherwise the α_i values will diverge. This limits the value of η to $\eta < 2/\lambda_{\max}$ where $\lambda_{\max}$ is the largest of the eigenvalues. The rate of convergence, however, is dominated by the smallest eigenvalue, so with η set to its largest permitted value, the convergence along the direction corresponding to the smallest eigenvalue (the long axis of the ellipse in Figure 7.3) will be governed by

$$\left(1 - \frac{2\lambda_{\min}}{\lambda_{\max}}\right) \tag{7.30}$$

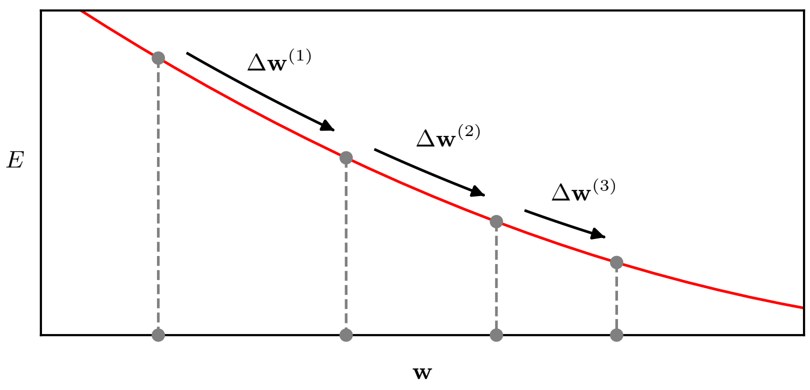


Figure 7.4 With a fixed learning rate parameter, gradient descent down a surface with low curvature leads to successively smaller steps corresponding to linear convergence. In such a situation, the effect of a momentum term is like an increase in the effective learning rate parameter.

where λ_{min} is the smallest eigenvalue. If the ratio $\lambda_{\text{min}}/\lambda_{\text{max}}$ (whose reciprocal is known as the *condition number* of the Hessian) is very small, corresponding to highly elongated elliptical error contours as in Figure 7.3, then progress towards the minimum will be extremely slow.

7.3.1 Momentum

One simple technique for dealing with the problem of widely differing eigenvalues is to add a *momentum* term to the gradient descent formula. This effectively adds inertia to the motion through weight space and smooths out the oscillations depicted in Figure 7.3. The modified gradient descent formula is given by

$$\Delta\mathbf{w}^{(\tau-1)} = -\eta\nabla E\left(\mathbf{w}^{(\tau-1)}\right) + \mu\Delta\mathbf{w}^{(\tau-2)} \tag{7.31}$$

where μ is called the momentum parameter. The weight vector is then updated using (7.15).

To understand the effect of the momentum term, consider first the motion through a region of weight space for which the error surface has relatively low curvature, as indicated in Figure 7.4. If we make the approximation that the gradient is unchanging, then we can apply (7.31) iteratively to a long series of weight updates, and then sum the resulting arithmetic series to give

$$\Delta\mathbf{w} = -\eta\nabla E\{1 + \mu + \mu^2 + \ldots\} \tag{7.32}$$

$$= -\frac{\eta}{1-\mu}\nabla E \tag{7.33}$$

and we see that the result of the momentum term is to increase the effective learning rate from η to $\eta/(1-\mu)$.

By contrast, in a region of high curvature in which gradient descent is oscillatory, as indicated in Figure 7.5, successive contributions from the momentum term will

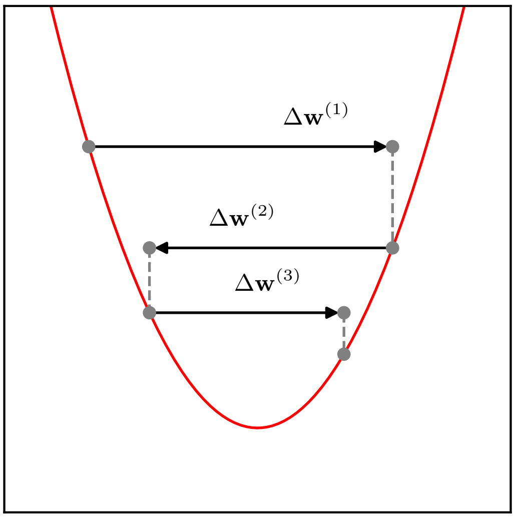


Figure 7.5 For a situation in which successive steps of gradient descent are oscillatory, a momentum term has little influence on the effective value of the learning rate parameter.

tend to cancel and the effective learning rate will be close to η. Thus, the momentum term can lead to faster convergence towards the minimum without causing divergent oscillations. A schematic illustration of the effect of a momentum term is shown in Figure 7.6.

Although the inclusion of momentum can lead to an improvement in the performance of gradient descent, it also introduces a second parameter μ whose value needs to be chosen, in addition to that of the learning rate parameter η. From (7.33) we see that μ should be in the range $0 \leqslant \mu \leqslant 1$. A typical value used in practice is $\mu = 0.9$. Stochastic gradient descent with momentum is summarized in Algorithm 7.3.

The convergence can be further accelerated using a modified version of momentum called *Nesterov momentum* (Nesterov, 2004; Sutskever *et al.*, 2013). In conventional stochastic gradient descent with momentum, we first compute the gradient at the current location then take a step that is amplified by adding momentum from the previous step. With the Nesterov method, we change the order of these and first compute a step based on the previous momentum, then calculate the gradient at this

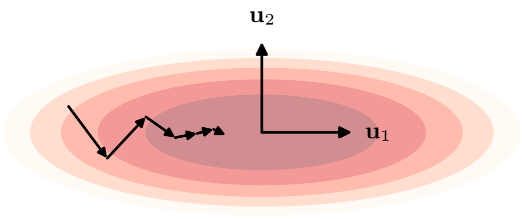


Figure 7.6 Illustration of the effect of adding a momentum term to the gradient descent algorithm, showing the more rapid progress along the valley of the error function, compared with the unmodified gradient descent shown in Figure 7.3.

Algorithm 7.3: Stochastic gradient descent with momentum

```
Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
       Batch size $B$
       Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
       Learning rate parameter $\eta$
       Momentum parameter $\mu$
       Initial weight vector $\mathbf{w}$
Output: Final weight vector $\mathbf{w}$
$n \leftarrow 1$
$\Delta\mathbf{w} \leftarrow \mathbf{0}$
repeat
    $\Delta\mathbf{w} \leftarrow -\eta\nabla E_{n:n+B-1}(\mathbf{w}) + \mu\Delta\mathbf{w}$ // calculate update term
    $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ // weight vector update
    $n \leftarrow n + B$
    if $n > N$ then
        shuffle data
        $n \leftarrow 1$
    end if
until convergence
return $\mathbf{w}$
```

new location to find the update, so that

$$\Delta\mathbf{w}^{(\tau-1)} = -\eta\nabla E\left(\mathbf{w}^{(\tau-1)} + \mu\Delta\mathbf{w}^{(\tau-2)}\right) + \mu\Delta\mathbf{w}^{(\tau-2)}. \tag{7.34}$$

For batch gradient descent, Nesterov momentum can improve the rate of convergence, although for stochastic gradient descent it can be less effective.

7.3.2 Learning rate schedule

In the stochastic gradient descent learning algorithm (7.18), we need to specify a value for the learning rate parameter η. If η is very small then learning will proceed slowly. However, if η is increased too much it can lead to instability. Although some oscillation can be tolerated, it should not be divergent. In practice, the best results are obtained by using a larger value for η at the start of training and then reducing the learning rate over time, so that the value of η becomes a function of the step index τ:

Section 7.3.1

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta^{(\tau-1)}\nabla E_n(\mathbf{w}^{(\tau-1)}). \tag{7.35}$$

Examples of learning rate schedules include linear, power law, and exponential decay:

$$\eta^{(\tau)} = (1 - \tau/K)\,\eta^{(0)} + (\tau/K)\eta^{(K)} \tag{7.36}$$

$$\eta^{(\tau)} = \eta^{(0)}\,(1 + \tau/s)^{c} \tag{7.37}$$

$$\eta^{(\tau)} = \eta^{(0)} c^{\tau/s} \tag{7.38}$$

where in (7.36) the value of η reduces linearly over K steps, after which its value is held constant at $\eta^{(K)}$. Good values for the hyperparameters $\eta^{(0)}$, $\eta^{(K)}$, K, S, and c must be found empirically. It can be very helpful in practice to monitor the *learning curve* showing how the error function evolves during the gradient descent iteration to ensure that it is decreasing at a suitable rate.

7.3.3 RMSProp and Adam

Section 7.3

We saw that the optimal learning rate depends on the local curvature of the error surface, and moreover that this curvature can vary according to the direction in parameter space. This motivates several algorithms that use different learning rates for each parameter in the network. The values of these learning rates are adjusted automatically during training. Here we review some of the most widely used examples. Note, however, that this intuition really applies only if the principal curvature directions are aligned with the axes in weight space, corresponding to a locally diagonal Hessian matrix, which is unlikely to be the case in practice. Nevertheless, these types of algorithms can be effective and are widely used.

The key idea behind *AdaGrad*, short for 'adaptive gradient', is to reduce each learning rate parameter over time by using the accumulated sum of squares of all the derivatives calculated for that parameter (Duchi, Hazan, and Singer, 2011). Thus, parameters associated with high curvature are reduced most rapidly. Specifically,

$$r_i^{(\tau)} = r_i^{(\tau-1)} + \left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right)^2 \tag{7.39}$$

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \frac{\eta}{\sqrt{r_i^{\tau}} + \delta}\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right) \tag{7.40}$$

where η is the learning rate parameter, and δ is a small constant, say 10^{-8}, that ensures numerical stability in the event that r_i is close to zero. The algorithm is initialized with $r_i^{(0)} = 0$. Here $E(\mathbf{w})$ is the error function for a particular mini-batch, and the update (7.40) is standard stochastic gradient descent but with a modified learning rate that is specific to each parameter.

One problem with AdaGrad is that it accumulates the squared gradients from the very start of training, and so the associated weight updates can become very small, which can slow down training too much in the later phases. The idea behind the *RMSProp* algorithm, which is short for 'root mean square propagation', is to replace the sum of squared gradients of AdaGrad with an exponentially weighted average

(Hinton, 2012), giving

$$r_i^{(\tau)} = \beta r_i^{(\tau-1)} + (1-\beta)\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right)^2 \tag{7.41}$$

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \frac{\eta}{\sqrt{r_i^\tau} + \delta}\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right) \tag{7.42}$$

where $0 < \beta < 1$ and a typical value is $\beta = 0.9$.

If we combine RMSProp with momentum, we obtain the *Adam* optimization method (Kingma and Ba, 2014) where the name is derived from 'adaptive moments'. Adam stores the momentum for each parameter separately using update equations that consist of exponentially weighted moving averages for both the gradients and the squared gradients in the form

$$s_i^{(\tau)} = \beta_1 s_i^{(\tau-1)} + (1-\beta_1)\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right) \tag{7.43}$$

$$r_i^{(\tau)} = \beta_2 r_i^{(\tau-1)} + (1-\beta_2)\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right)^2 \tag{7.44}$$

$$\widehat{s_i}^{(\tau)} = \frac{s_i^{(\tau)}}{1-\beta_1^\tau} \tag{7.45}$$

$$\widehat{r_i}^\tau = \frac{r_i^\tau}{1-\beta_2^\tau} \tag{7.46}$$

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \eta\frac{\widehat{s_i}^\tau}{\sqrt{\widehat{r_i}^\tau} + \delta}. \tag{7.47}$$

Here the factors $1/(1-\beta_1^\tau)$ and $1/(1-\beta_2^\tau)$ correct for a bias introduced by initializing $s_i^{(0)}$ and $r_i^{(0)}$ to zero. Note that the bias goes to zero as τ becomes large, since $\beta_i < 1$, and so in practice this bias correction is sometimes omitted. Typical values for the weighting parameters are $\beta_1 = 0.9$ and $\beta_2 = 0.99$. Adam is the most widely adopted learning algorithm in deep learning and is summarized in Algorithm 7.4.

Exercise 7.12

7.4. Normalization

Normalization of the variables computed during the forward pass through a neural network removes the need for the network to deal with extremely large or extremely small values. Although in principle the weights and biases in a neural network can adapt to whatever values the input and hidden variables take, in practice normalization can be crucial for ensuring effective training. Here we consider three kinds of normalization according to whether we are normalizing across the input data, across mini-batches, or across layers.

Algorithm 7.4: Adam optimization

```
Input: Training set of data points indexed by $n \in \{1, \dots, N\}$
       Batch size $B$
       Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
       Learning rate parameter $\eta$
       Decay parameters $\beta_1$ and $\beta_2$
       Stabilization parameter $\delta$
Output: Final weight vector $\mathbf{w}$
$n \leftarrow 1$
$\mathbf{s} \leftarrow \mathbf{0}$
$\mathbf{r} \leftarrow \mathbf{0}$
repeat
    Choose a mini-batch at random from $\mathcal{D}$
    $\mathbf{g} = -\nabla E_{n:n+B-1}(\mathbf{w})$ // evaluate gradient vector
    $\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_1)\mathbf{g}$
    $\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$ // element-wise multiply
    $\widehat{\mathbf{s}} \leftarrow \mathbf{s}/(1 - \beta_1^\tau)$ // bias correction
    $\widehat{\mathbf{r}} \leftarrow \mathbf{r}/(1 - \beta_2^\tau)$ // bias correction
    $\Delta\mathbf{w} \leftarrow -\eta \dfrac{\widehat{\mathbf{s}}}{\sqrt{\widehat{\mathbf{r}}} + \delta}$ // element-wise operations
    $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ // weight vector update
    $n \leftarrow n + B$
    if $n + B > N$ then
        shuffle data
        $n \leftarrow 1$
    end if
until convergence
return $\mathbf{w}$
```

7.4.1 Data normalization

Sometimes we encounter data sets in which different input variables span very different ranges. For example, in health data, a patient's height might be measured in meters, such as 1.8m, whereas their blood platelet count might be measured in platelets per microliter, such as 300,000 platelets per μL. Such variations can make gradient descent training much more challenging. Consider a single-layer regression network with two weights in which the two corresponding input variables have very different ranges. Changes in the value of one of the weights produce much larger changes in the output, and hence in the error function, than would similar changes in the other weight. This corresponds to an error surface with very different curvatures along different axes as illustrated in Figure 7.3.

For continuous input variables, it can therefore be very beneficial to re-scale the input values so that they span similar ranges. This is easily done by first evaluating the mean and variance of each input:

$$\mu_i = \frac{1}{N}\sum_{n=1}^{N} x_{ni} \tag{7.48}$$

$$\sigma_i^2 = \frac{1}{N}\sum_{n=1}^{N}(x_{ni} - \mu_i)^2, \tag{7.49}$$

which is a calculation that is performed once, before any training is started. The input values are then re-scaled using

$$\widetilde{x}_{ni} = \frac{x_{ni} - \mu_i}{\sigma_i} \tag{7.50}$$

Exercise 7.14

so that the re-scaled values $\{\widetilde{x}_{ni}\}$ have zero mean and unit variance. Note that the same values of μ_i and σ_i must be used to pre-process any development, validation, or test data to ensure that all inputs are scaled in the same way. Input data normalization is illustrated in Figure 7.7.

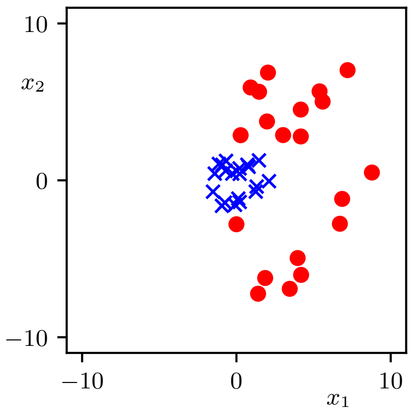


Figure 7.7 Illustration of the effect of input data normalization. The red circles show the original data points for a data set with two variables. The blue crosses show the data set after normalization such that each variable now has zero mean and unit variance across the data set.

7.4.2 Batch normalization

We have seen the importance of normalizing the input data, and we can apply similar reasoning to the variables in each hidden layer of a deep network. If there is wide variation in the range of activation values in a particular hidden layer, then normalizing those values to have zero mean and unit variance should make the learning problem easier for the next layer. However, unlike normalization of the input values, which can be done once prior to the start of training, normalization of the hidden-unit values will need to be repeated during training every time the weight values are updated. This is called *batch normalization* (Ioffe and Szegedy, 2015).

A further motivation for batch normalization arises from the phenomena of *vanishing gradients* and *exploding gradients*, which occur when we try to train very deep neural networks. From the chain rule of calculus, the gradient of an error function E with respect to a parameter in the first layer of the network is given by

Section 8.1.5

$$\frac{\partial E}{\partial w_i} = \sum_m \cdots \sum_l \sum_j \frac{\partial z_m^{(1)}}{\partial w_i} \cdots \frac{\partial z_j^{(K)}}{\partial z_l^{(K-1)}} \frac{\partial E}{\partial z_j^{(K)}} \tag{7.51}$$

where $z_j^{(k)}$ denotes the activation of node j in layer k, and each of the partial derivatives on the right-hand side of (7.51) represents the elements of the Jacobian matrix for that layer. The product of a large number of such terms will tend towards 0 if most of them have a magnitude < 1 and will tend towards ∞ if most of them have a magnitude > 1. Consequently, as the depth of a network increases, error function gradients can tend to become either very large or very small. Batch normalization largely resolves this issue.

Section 8.1.5

To see how batch normalization is defined, consider a specific layer within a multi-layer network. Each hidden unit in that layer computes a nonlinear function of its input pre-activation $z_i = h(a_i)$, and so we have a choice of whether to normalize the pre-activation values a_i or the activation values z_i. In practice, either approach may be used, and here we illustrate the procedure by normalizing the pre-activations. Because weight values are updated after each mini-batch of examples, we apply the normalization to each mini-batch. Specifically, for a mini-batch of size K, we define

$$\mu_i = \frac{1}{K} \sum_{n=1}^{K} a_{ni} \tag{7.52}$$

$$\sigma_i^2 = \frac{1}{K} \sum_{n=1}^{K} (a_{ni} - \mu_i)^2 \tag{7.53}$$

$$\widehat{a}_{ni} = \frac{a_{ni} - \mu_i}{\sqrt{\sigma_i^2 + \delta}} \tag{7.54}$$

where the summations over $n = 1, \ldots, K$ are taken over the elements of the mini-batch. Here δ is a small constant, introduced to avoid numerical issues in situations where σ_i^2 is small.

By normalizing the pre-activations in a given layer of the network, we reduce the number of degrees of freedom in the parameters of that layer and hence we

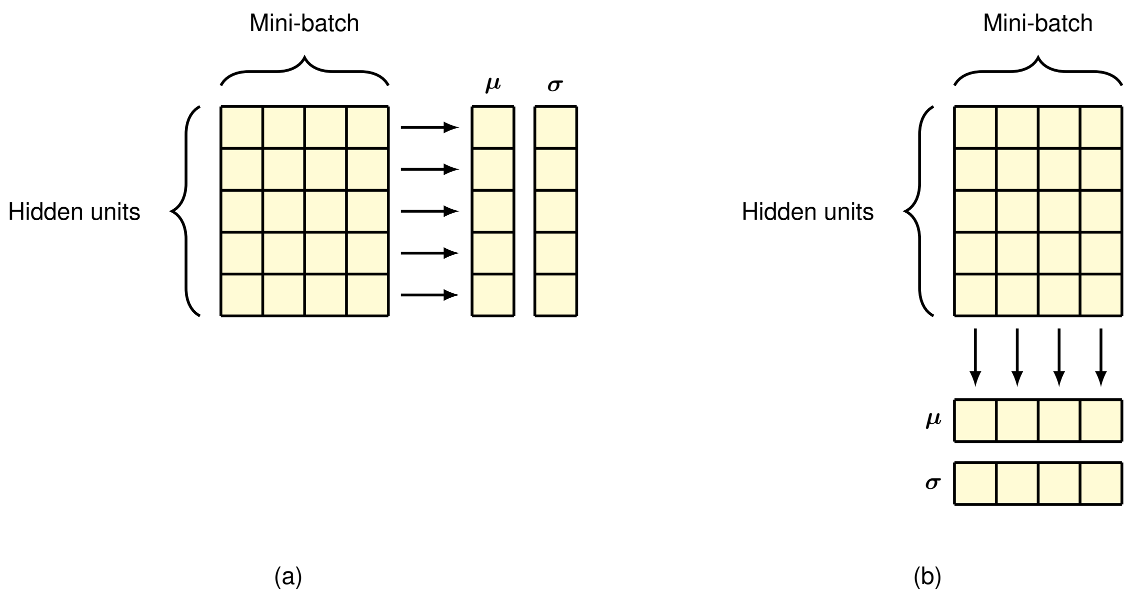


Figure 7.8 Illustration of batch normalization and layer normalization in a neural network. In batch normalization, shown in (a), the mean and variance are computed across the mini-batch separately for each hidden unit. In layer normalization, shown in (b), the mean and variance are computed across the hidden units separately for each data point.

reduce its representational capability. We can compensate for this by re-scaling the pre-activations of the batch to have mean β_i and standard deviation γ_i using

$$\widetilde{a}_{ni} = \gamma_i \widehat{a}_{ni} + \beta_i \tag{7.55}$$

where β_i and γ_i are adaptive parameters that are learned by gradient descent jointly with the weights and biases of the network. These learnable parameters represent a key difference compared to input data normalization. *Section 7.4.1*

It might appear that the transformation (7.55) has simply undone the effect of the batch normalization since the mean and variance can now adapt to arbitrary values again. However, the crucial difference is in the way the parameters evolve during training. For the original network, the mean and variance across a mini-batch are determined by a complex function of all the weights and biases in the layer, whereas in the representation given by (7.55), they are determined directly by independent parameters β_i and γ_i, which turn out to be much easier to learn during gradient descent.

Equations (7.52) – (7.55) describe a transformation of the variables that is differentiable with respect to the learnable parameters β_i and γ_i. This can be viewed as an additional layer in the neural network, and so each standard hidden layer can be followed by a batch normalization layer. The structure of the batch-normalization process is illustrated in Figure 7.8.

Once the network is trained and we want to make predictions on new data, we no longer have the training mini-batches available, and we cannot determine a mean and variance from just one data example. To solve this, we could in principle evaluate μ_i and σ_i^2 for each layer across the whole training set after we have made the final update to the weights and biases. However, this would involve processing the whole data set just to evaluate these quantities and is therefore usually too expensive. Instead, we compute moving averages throughout the training phase:

$$\overline{\mu}_i^{(\tau)} = \alpha\overline{\mu}_i^{(\tau-1)} + (1-\alpha)\mu_i \tag{7.56}$$

$$\overline{\sigma}_i^{(\tau)} = \alpha\overline{\sigma}_i^{(\tau-1)} + (1-\alpha)\sigma_i \tag{7.57}$$

where $0 \leqslant \alpha \leqslant 1$. These moving averages play no role during training but are used to process new data points during the inference phase.

Although batch normalization is very effective in practice, there is uncertainty as to why it works so well. Batch normalization was originally motivated by noting that updates to weights in earlier layers of the network change the distribution of values seen by later layers, a phenomenon called *internal covariate shift*. However, later studies (Santurkar *et al.*, 2018) suggest that covariate shift is not a significant factor and that the improved training results from an improvement in the smoothness of the error function landscape.

7.4.3 Layer normalization

With batch normalization, if the batch size is too small then the estimates of the mean and variance become too noisy. Also, for very large training sets, the mini-batches may be split across different GPUs, making global normalization across the mini-batch inefficient. An alternative to normalizing across examples within a mini-batch for each hidden unit separately is to normalize across the hidden-unit values for each data point separately. This is known as *layer normalization* (Ba, Kiros, and Hinton, 2016). It was introduced in the context of recurrent neural networks where the distributions change after each time step making batch normalization infeasible. However, it is useful in other architectures such as transformer networks.

Section 12.2.5

Chapter 12

By analogy with batch normalization, we therefore make the following transformation:

$$\mu_n = \frac{1}{M}\sum_{i=1}^{M} a_{ni} \tag{7.58}$$

$$\sigma_n^2 = \frac{1}{M}\sum_{i=1}^{M} (a_{ni} - \mu_i)^2 \tag{7.59}$$

$$\widehat{a}_{ni} = \frac{a_{ni} - \mu_n}{\sqrt{\sigma_n^2 + \delta}} \tag{7.60}$$

where the sums $i = 1, \ldots, M$ are taken over all hidden units in the layer. As with batch normalization, additional learnable mean and standard deviation parameters are introduced for each hidden unit separately in the form (7.55). Note that the same normalization function can be employed during training and during inference, and so there is no need to store moving averages. Layer normalization is compared with batch normalization in Figure 7.8.

Exercises

7.1 ($\star$) By substituting (7.10) into (7.7) and using (7.8) and (7.9), show that the error function (7.7) can be written in the form (7.11).

7.2 ($\star$) Consider a Hessian matrix $\mathbf{H}$ with eigenvector equation (7.8). By setting the vector $\mathbf{v}$ in (7.14) equal to each of the eigenvectors $\mathbf{u}_i$ in turn, show that $\mathbf{H}$ is positive definite if, and only if, all its eigenvalues are positive.

7.3 ($\star\star$) By considering the local Taylor expansion (7.7) of an error function about a stationary point $\mathbf{w}^\star$, show that the necessary and sufficient condition for the stationary point to be a local minimum of the error function is that the Hessian matrix $\mathbf{H}$, defined by (7.5) with $\widehat{\mathbf{w}} = \mathbf{w}^\star$, is positive definite.

7.4 ($\star\star$) Consider a linear regression model with a single input variable x and a single output variable y of the form

$$y(x, w, b) = wx + b \tag{7.61}$$

together with a sum-of-squares error function given by

$$E(w, b) = \frac{1}{2}\sum_{n=1}^{N} \left\{y(x_n, w, b) - t_n\right\}^2 . \tag{7.62}$$

Derive expressions for the elements of the 2×2 Hessian matrix given by the second derivatives of the error function with respect to the weight parameter w and bias parameter b. Show that the trace and the determinant of this Hessian are both positive. Since the trace represents the sum of the eigenvalue and the determinant corresponds to the product of the eigenvalues, then both eigenvalues are positive and hence the stationary point of the error function is a minimum.

Appendix A

7.5 ($\star\star$) Consider a single-layer classification model with a single input variable x and a single output variable y of the form

$$y(x, w, b) = \sigma(wx + b) \tag{7.63}$$

where $\sigma(\cdot)$ is the logistic sigmoid function defined by (5.42) together with a cross-entropy error function given by

$$E(w, b) = \sum_{n=1}^{N} \left\{t_n \ln y(x_n, w, b) + (1 - t_n)\ln(1 - y(x_n, w, b))\right\} . \tag{7.64}$$

Derive expressions for the elements of the 2×2 Hessian matrix given by the second derivatives of the error function with respect to the weight parameter w and bias parameter b. Show that the trace and the determinant of this Hessian are both positive.

Appendix A

Since the trace represents the sum of the eigenvalue and the determinant corresponds to the product of the eigenvalues, then both eigenvalues are positive and hence the stationary point of the error function is a minimum.

7.6 ($\star\star$) Consider a quadratic error function defined by (7.7) in which the Hessian matrix $\mathbf{H}$ has an eigenvalue equation given by (7.8). Show that the contours of constant error are ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ with lengths that are inversely proportional to the square roots of the corresponding eigenvalues λ_i.

7.7 ($\star$) Show that, as a consequence of the symmetry of the Hessian matrix $\mathbf{H}$, the number of independent elements in the quadratic error function (7.3) is given by $W(W+3)/2$.

7.8 ($\star$) Consider a set of values $x_1, \ldots, x_N$ drawn from a distribution with mean μ and variance σ^2, and define the sample mean to be

$$\overline{x} = \frac{1}{N}\sum_{n=1}^{N} x_n. \tag{7.65}$$

Show that the expectation of the squared error $(\overline{x} - \mu)^2$ with respect to the distribution from which the data is drawn is given by σ^2/N. This shows that the RMS error in the sample mean is given by $\sigma/\sqrt{N}$, which decreases relatively slowly as the sample size N increases.

7.9 ($\star\star$) Consider a layered network that computes the functions (7.19) and (7.20) in layer l. Suppose we initialize the weights using a Gaussian $\mathcal{N}(0, \epsilon^2)$, and suppose that the outputs $z_j^{(l-1)}$ of the units in layer $l-1$ have variance λ^2. By using the form of the ReLU activation function, show that the mean and variance of the outputs in layer l are given by (7.21) and (7.22), respectively. Hence, show that if we want the units in layer l also to have pre-activations with variance λ^2 then the value of ϵ should be given by (7.23).

7.10 ($\star\star$) By making use of (7.7), (7.8), and (7.10), derive the results (7.24) and (7.25), which express the gradient vector and a general weight update, as expansions in the eigenvectors of the Hessian matrix. Use these results, together with the eigenvector orthonormality relation (7.9) and the batch gradient descent formula (7.16), to derive the result (7.26) for the batch gradient descent update expressed in terms of the coefficients $\{\alpha_i\}$.

7.11 ($\star$) Consider a smoothly varying error surface with low curvature such that the gradient varies only slowly with position. Show that, for small values of the learning rate and momentum parameters, the Nesterov momentum gradient update defined by (7.34) is equivalent to the standard gradient descent with momentum defined by (7.31).

7.12 ($\star\star$) Consider a sequence of values $\{x_1, \ldots, x_N\}$ of some variable x, and suppose we compute an exponentially weighted moving average using the formula

$$\mu_n = \beta\mu_{n-1} + (1-\beta)x_n \tag{7.66}$$

where $0 \leqslant \beta \leqslant 1$. By making use of the following result for the sum of a finite geometric series

$$\sum_{k=1}^{n} \beta^{k-1} = \frac{1-\beta^n}{1-\beta} \tag{7.67}$$

show that if the sequence of averages is initialized using $\mu_0 = 0$, then the estimators are biased and that the bias can be corrected using

$$\widehat{\mu}_n = \frac{\mu_n}{1-\beta^n}. \tag{7.68}$$

7.13 ($\star$) In gradient descent, the weight vector $\mathbf{w}$ is updated by taking a step in weight space in the direction of the negative gradient governed by a learning rate parameter η. Suppose instead that we choose a direction $\mathbf{d}$ in weight space along which we minimize the error function, given the current weight vector $\mathbf{w}^{(\tau)}$. This involves minimizing the quantity

$$E(\mathbf{w}^{(\tau)} + \lambda \mathbf{d}) \tag{7.69}$$

as a function of λ to give a value $\lambda^\star$ corresponding to a new weight vector $\mathbf{w}^{(\tau+1)}$. Show that the gradient of $E(\mathbf{w})$ at $\mathbf{w}^{(\tau+1)}$ is orthogonal to the vector $\mathbf{d}$. This is known as a 'line search' method and it forms the basis for a variety of numerical optimization algorithms (Bishop, 1995b).

7.14 ($\star$) Show that the renormalized input variables defined by (7.50), where μ_i is defined by (7.48) and σ_i^2 is defined by (7.49), have zero mean and unit variance.

8 Backpropagation

Our goal in this chapter is to find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network. We will see that this can be achieved using a local message-passing scheme in which information is sent backwards through the network and is known as *error backpropagation*, or sometimes simply as *backprop*.

Historically, the backpropagation equations would have been derived by hand and then implemented in software alongside the forward propagation equations, with both steps taking time and being prone to mistakes. Modern neural network software environments, however, allow virtually any derivatives of interest to be calculated efficiently with only minimal effort beyond that of coding up the original network function. This idea, called *automatic differentiation*, plays a key role in modern deep learning. However, it is valuable to understand how the calculations are performed so that we are not relying on 'black box' software solutions. In this chapter we

Section 8.2