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

C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_7

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.

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 *back-propagation*, 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.

Chapter 8

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.

Section 9.3.2

Chapter 9

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

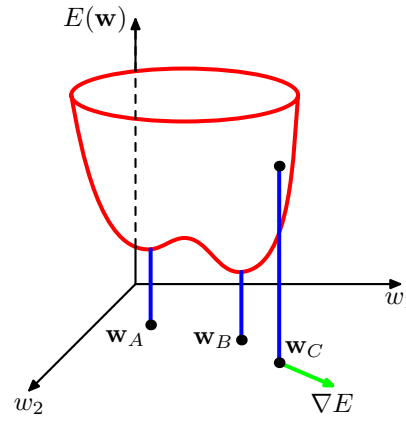

Figure 7.1 Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum, so that $E(\mathbf{w}_\mathrm{A}) > E(\mathbf{w}_\mathrm{B})$. At any point $\mathbf{w}_C$, the local gradient of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

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

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 *Appendix A* 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 *Exercise 7.1* 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,

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

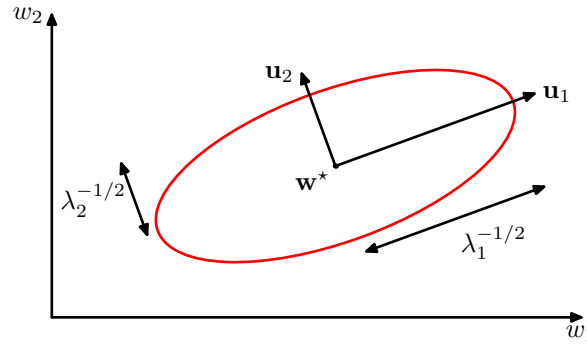

Figure 7.2 In the neighbourhood of a minimum $\mathbf{w}^\star$, the error function can be approximated by a quadratic. Contours of constant error are then ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ of the Hessian matrix, with lengths that are inversely proportional to the square 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 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 *Exercise 7.3* 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 *Exercise 7.6* 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

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.

Chapter 8

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 $\mathcal{O}(W^3)$.

Exercise 7.7

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.

Chapter 8

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 $n \in \{1, \ldots, N\}$
Error function per data point $E_n(\mathbf{w})$
Learning rate parameter η
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`

  $n \leftarrow n + 1 (\text{mod } N)$ `// 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 η
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

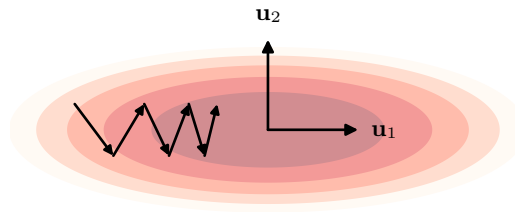

Figure 7.3 Schematic illustration of fixed-step gradient descent for an error function that has substantially different curvatures along different directions. The error surface E has the form of a long valley, as depicted by the ellipses. Note that, for most points in weight space, the local negative 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. From (7.7), (7.8), and (7.10), the gradient of the error function in this approximation

Section 7.1.1

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

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

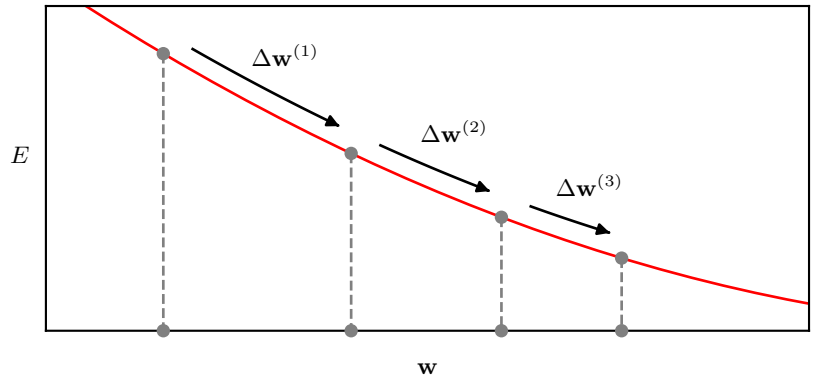

Figure 7.4 With a fixed learning rate parameter, gradient descent down a surface with low curvature leads to successively smaller steps corresponding to linear convergence. In such a situation, the effect of a momentum term is like an increase in the effective learning rate parameter.

where $\lambda_{\min}$ is the smallest eigenvalue. If the ratio $\lambda_{\min}/\lambda_{\max}$ (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

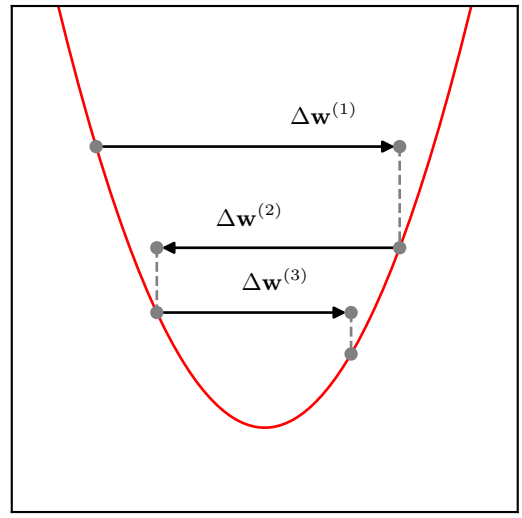

Figure 7.5 For a situation in which successive steps of gradient descent are oscillatory, a momentum term has little influence on the effective value of the learning rate parameter.

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

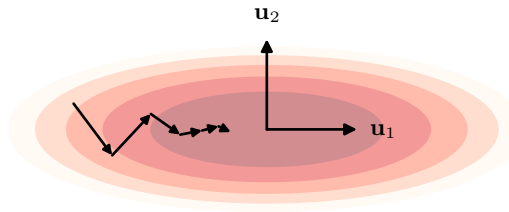

Figure 7.6 Illustration of the effect of adding a momentum term to the gradient descent algorithm, showing the more rapid progress along the valley of the error function, compared with the unmodified gradient descent shown in Figure 7.3.

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 η
Momentum parameter μ
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

Section 7.3.1

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:

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

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

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

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

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

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

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \eta \frac{\widehat{s}_i^{\,\tau}}{\sqrt{\widehat{r}_i^{\,\tau}} + \delta}. \tag{7.47}$$

Exercise 7.12

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.

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 $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Decay parameters β_1 and β_2
Stabilization parameter δ

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.

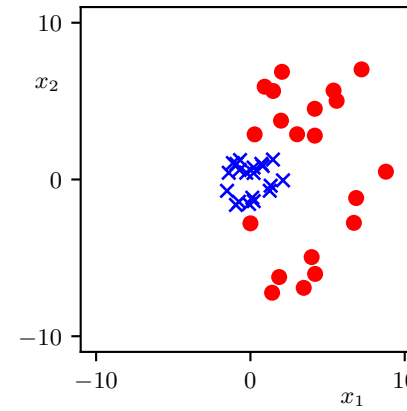

Figure 7.7 Illustration of the effect of input data normalization. The red circles show the original data points for a data set with two variables. The blue crosses show the data set after normalization such that each variable now has zero mean and unit variance across the data set.

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

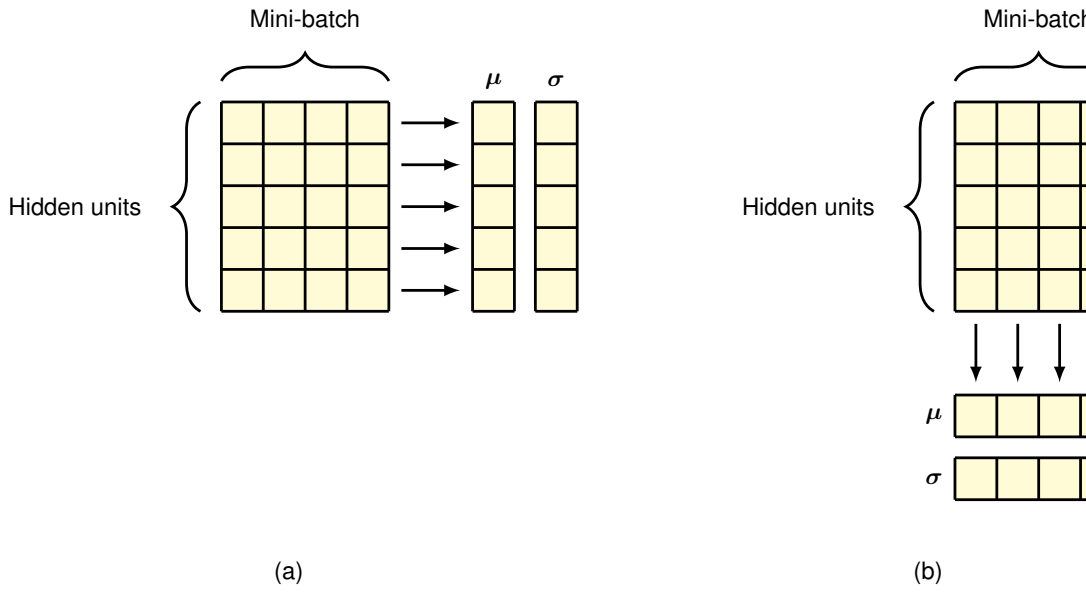

Figure 7.8 Illustration of batch normalization and layer normalization in a neural network. In batch normalization, shown in (a), the mean and variance are computed across the mini-batch separately for each hidden unit. In layer normalization, shown in (b), the mean and variance are computed across the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

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

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 (⋆⋆) 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

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

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.