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

therefore explain the key concepts of backpropagation, and explore the framework of automatic differentiation in detail.

Note that the term ‘backpropagation’ is used in the neural computing literature in a variety of different ways. For instance, a feed-forward architecture may be called a backpropagation network. Also the term ‘backpropagation’ is sometimes used to describe the end-to-end training procedure for a neural network including the gradient descent parameter updates. In this book we will use ‘backpropagation’ specifically to describe the computational procedure used in the numerical evaluation of derivatives such as the gradient of the error function with respect to the weights and biases of a network. This procedure can also be applied to the evaluation of other important derivatives such as the Jacobian and Hessian matrices.

8.1. Evaluation of Gradients

We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function. The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units together with a sum-of-squares error.

Many error functions of practical interest, for instance those defined by maximum likelihood for a set of i.i.d. data, comprise a sum of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{8.1}$$

Here we will consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function. This may be used directly for stochastic gradient descent, or the results could be accumulated over a set of training data points for batch or mini-batch methods.

8.1.1 Single-layer networks

Consider first a simple linear model in which the outputs y_k are linear combinations of the input variables x_i so that

$$y_k = \sum_i w_{ki} x_i \tag{8.2}$$

together with a sum-of-squares error function that, for a particular input data point n, takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{8.3}$$

where $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$, and t_{nk} is the associated target value. The gradient of this error function with respect to a weight w_{ji} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj})x_{ni}. \tag{8.4}$$

This can be interpreted as a 'local' computation involving the product of an 'error signal' $y_{nj} - t_{nj}$ associated with the output end of the link w_{ji} and the variable x_{ni} associated with the input end of the link. In Section 5.4.3, we saw how a similar formula arises with the logistic-sigmoid activation function together with the cross-entropy error function and similarly for the softmax activation function together with its matching multivariate cross-entropy error function. We will now see how this simple result extends to the more complex setting of multilayer feed-forward networks.

8.1.2 General feed-forward networks

In general, a feed-forward network consists of a set of units each of which computes a weighted sum of its inputs:

$$a_j = \sum_i w_{ji} z_i \tag{8.5}$$

where z_i is either the activation of another unit or an input unit that sends a connection to unit j, and w_{ji} is the weight associated with that connection. Biases can be included in this sum by introducing an extra unit, or input, with activation fixed at $+1$, and so we do not need to deal with biases explicitly. The sum in (8.5), known as a pre-activation, is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

Section 6.2

$$z_j = h(a_j). \tag{8.6}$$

Note that one or more of the variables z_i in the sum in (8.5) could be an input, and similarly, the unit j in (8.6) could be an output.

For each data point in the training set, we will suppose that we have supplied the corresponding input vector to the network and calculated the activations of all the hidden and output units in the network by successive application of (8.5) and (8.6). This process is called *forward propagation* because it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input data point n. However, to keep the notation uncluttered, we will omit the subscript n from the network variables. First note that E_n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{8.7}$$

We now introduce a useful notation:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{8.8}$$

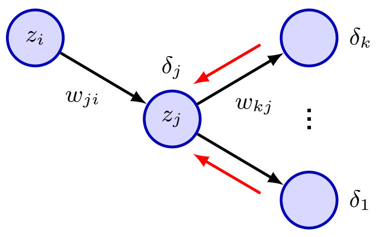


Figure 8.1 Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The black arrows denote the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

where the δ's are often referred to as *errors* for reasons we will see shortly. Using (8.5), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{8.9}$$

Substituting (8.8) and (8.9) into (8.7), we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{8.10}$$

Equation (8.10) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ for a bias). Note that this takes the same form as that found for the simple linear model in (8.4). Thus, to evaluate the derivatives, we need calculate only the value of δ_j for each hidden and output unit in the network and then apply (8.10).

As we have seen already, for the output units, we have

$$\delta_k = y_k - t_k \tag{8.11}$$

Section 5.4.6

provided we are using the canonical link as the output-unit activation function. To evaluate the δ's for hidden units, we again make use of the chain rule for partial derivatives:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{8.12}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 8.1. Note that the units labelled k include other hidden units and/or output units. In writing down (8.12), we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k.

If we now substitute the definition of δ_j given by (8.8) into (8.12) and make use

Exercise 8.1

of (8.5) and (8.6), we obtain the following *backpropagation* formula:

$$\delta_j = h'(a_j)\sum_k w_{kj}\delta_k, \tag{8.13}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated

Algorithm 8.1: Backpropagation

```
Input: Input vector $\mathbf{x}_n$
       Network parameters $\mathbf{w}$
       Error function $E_n(\mathbf{w})$ for input $x_n$
       Activation function $h(a)$
Output: Error function derivatives $\{\partial E_n / \partial w_{ji}\}$
// Forward propagation
for $j \in$ all hidden and output units do
    $a_j \leftarrow \sum_i w_{ji} z_i$ // $\{z_i\}$ includes inputs $\{x_i\}$
    $z_j \leftarrow h(a_j)$ // activation function
end for
// Error evaluation
for $k \in$ all output units do
    $\delta_k \leftarrow \frac{\partial E_n}{\partial a_k}$ // compute errors
end for
// Backward propagation, in reverse order
for $j \in$ all hidden units do
    $\delta_j \leftarrow h'(a_j) \sum_k w_{kj} \delta_k$ // recursive backward evaluation
    $\frac{\partial E_n}{\partial w_{ji}} \leftarrow \delta_j z_i$ // evaluate derivatives
end for
return $\left\{ \frac{\partial E_n}{\partial w_{ji}} \right\}$
```

in Figure 8.1. Note that the summation in (8.13) is taken over the first index on w_{kj} (corresponding to backward propagation of information through the network), whereas in the forward propagation equation (8.5), it is taken over the second index. *Exercise 8.2* Because we already know the values of the δ's for the output units, it follows that by recursively applying (8.13), we can evaluate the δ's for all the hidden units in a feed-forward network, regardless of its topology. The backpropagation procedure is summarized in Algorithm 8.1.

For batch methods, the derivative of the total error E can then be obtained by repeating the above steps for each data point in the training set and then summing over all data points in the batch or mini-batch:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}. \tag{8.14}$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $h(\cdot)$. However, the derivation is easily generalized to allow different units to have individual activation functions, simply by keeping track of which form of $h(\cdot)$ goes with which unit.

8.1.3 A simple example

The above derivation of the backpropagation procedure allowed for general forms for the error function, the activation functions, and the network topology. To illustrate the application of this algorithm, we consider a two-layer network of the form illustrated in Figure 6.9, together with a sum-of-squares error. The output units have linear activation functions, so that $y_k = a_k$, and the hidden units have sigmoidal activation functions given by

$$h(a) \equiv \tanh(a) \tag{8.15}$$

where $\tanh(a)$ is defined by (6.14). A useful feature of this function is that its derivative can be expressed in a particularly simple form:

$$h'(a) = 1 - h(a)^2. \tag{8.16}$$

We also consider a sum-of-squares error function, so that for data point n the error is given by

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2 \tag{8.17}$$

where y_k is the activation of output unit k, and t_k is the corresponding target value for a particular input vector $\mathbf{x}_n$.

For each data point in the training set in turn, we first perform a forward propagation using

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{8.18}$$

$$z_j = \tanh(a_j) \tag{8.19}$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \tag{8.20}$$

where D is the dimensionality of the input vector $\mathbf{x}$ and M is the total number of hidden units. Also we have used $x_0 = z_0 = 1$ to allow bias parameters to be included in the weights. Next we compute the δ's for each output unit using

$$\delta_k = y_k - t_k. \tag{8.21}$$

Then, we backpropagate these errors to obtain δ's for the hidden units using

$$\delta_j = (1 - z_j^2)\sum_{k=1}^{K} w_{kj}^{(2)} \delta_k, \tag{8.22}$$

which follows from (8.13) and (8.16). Finally, the derivatives with respect to the first-layer and second-layer weights are given by

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i, \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j. \tag{8.23}$$

8.1.4 Numerical differentiation

One of the most important aspects of backpropagation is its computational efficiency. To understand this, let us examine how the number of compute operations required to evaluate the derivatives of the error function scales with the total number W of weights and biases in the network.

A single evaluation of the error function (for a given input data point) would require $\mathcal{O}(W)$ operations, for sufficiently large W. This follows because, except for a network with very sparse connections, the number of weights is typically much greater than the number of units, and so the bulk of the computational effort in forward propagation arises from evaluation of the sums in (8.5), with the evaluation of the activation functions representing a small overhead. Each term in the sum in (8.5) requires one multiplication and one addition, leading to an overall computational cost that is $\mathcal{O}(W)$.

An alternative approach to backpropagation for computing the derivatives of the error function is to use finite differences. This can be done by perturbing each weight in turn and approximating the derivatives by using the expression

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji}+\epsilon) - E_n(w_{ji})}{\epsilon} + \mathcal{O}(\epsilon) \tag{8.24}$$

where $\epsilon \ll 1$. In a software simulation, the accuracy of the approximation to the derivatives can be improved by making ϵ smaller, until numerical round-off problems arise. The accuracy of the finite differences method can be improved significantly by using symmetrical *central differences* of the form

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji}+\epsilon) - E_n(w_{ji}-\epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2). \tag{8.25}$$

Exercise 8.3

In this case, the $\mathcal{O}(\epsilon)$ corrections cancel, as can be verified by a Taylor expansion of the right-hand side of (8.25), and so the residual corrections are $\mathcal{O}(\epsilon^2)$. Note, however, that the number of computational steps is roughly doubled compared with (8.24). Figure 8.2 shows a plot of the error between a numerical evaluation of a gradient using both finite differences (8.24) and central differences (8.25) versus the analytical result, as a function of the value of the step size ϵ.

The main problem with numerical differentiation is that the highly desirable $\mathcal{O}(W)$ scaling has been lost. Each forward propagation requires $\mathcal{O}(W)$ steps, and there are W weights in the network each of which must be perturbed individually, so that the overall computational cost is $\mathcal{O}(W^2)$.

However, numerical differentiation can play a useful role in practice, because a comparison of the derivatives calculated from a direct implementation of backpropagation, or from automatic differentiation, with those obtained using central differences provides a powerful check on the correctness of the software.

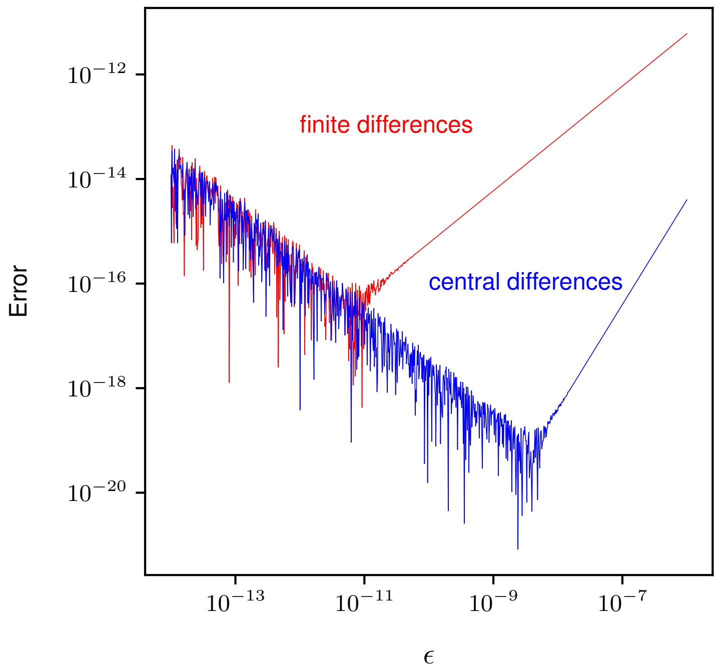


Figure 8.2 The red curve shows a plot of the error between the numerical evaluation of a gradient using finite differences (8.24) and the analytical result, as a function of ϵ. As ϵ decreases, the plot initially shows a linear decrease in error, and this represents a power law behaviour since the axes are logarithmic. The slope of this line is 1 which shows that this error behaves like $\mathcal{O}(\epsilon)$. At some point the evaluated gradient reaches the limit of numerical round-off and further reduction in ϵ leads to a noisy line, which again follows a power law but where the error now increases with decreasing ϵ. The blue curve shows the corresponding result for central differences (8.25). We see a much smaller error compared to finite differences, and the slope of the line is 2 which shows that the error is $\mathcal{O}(\epsilon^2)$.

8.1.5 The Jacobian matrix

We have seen how the derivatives of an error function with respect to the weights can be obtained by propagating errors backwards through the network. Backpropagation can also be used to calculate other derivatives. Here we consider the evaluation of the *Jacobian* matrix, whose elements are given by the derivatives of the network outputs with respect to the inputs:

$$J_{ki} \equiv \frac{\partial y_k}{\partial x_i} \tag{8.26}$$

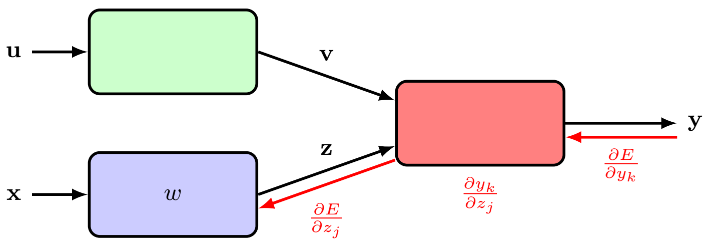
where each such derivative is evaluated with all other inputs held fixed. Jacobian matrices play a useful role in systems built from a number of distinct modules, as illustrated in Figure 8.3. Each module can comprise a fixed or learnable function, which can be linear or nonlinear, so long as it is differentiable.

Suppose we wish to minimize an error function E with respect to the parameter w in Figure 8.3. The derivative of the error function is given by

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w} \tag{8.27}$$

in which the Jacobian matrix for the red module in Figure 8.3 appears as the middle term on the right-hand side.

Because the Jacobian matrix provides a measure of the local sensitivity of the outputs to changes in each of the input variables, it also allows any known errors Δx_i



$\mathbf{u}$
$\mathbf{v}$
$\mathbf{y}$
$\mathbf{z}$
$\mathbf{x}$
w
$\frac{\partial E}{\partial y_k}$
$\frac{\partial y_k}{\partial z_j}$
$\frac{\partial E}{\partial z_j}$

Figure 8.3 Illustration of a modular deep learning architecture in which the Jacobian matrix can be used to back-propagate error signals from the outputs through to earlier modules in the system.

associated with the inputs to be propagated through the trained network to estimate their contribution Δy_k to the errors at the outputs, through the relation

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_i} \Delta x_i, \tag{8.28}$$

which assumes that the $|\Delta x_i|$ are small. In general, the network mapping represented by a trained neural network will be nonlinear, and so the elements of the Jacobian matrix will not be constants but will depend on the particular input vector used. Thus, (8.28) is valid only for small perturbations of the inputs, and the Jacobian itself must be re-evaluated for each new input vector.

The Jacobian matrix can be evaluated using a backpropagation procedure that is like the one derived earlier for evaluating the derivatives of an error function with respect to the weights. We start by writing the element J_{ki} in the form

$$\begin{aligned} J_{ki} = \frac{\partial y_k}{\partial x_i} &= \sum_j \frac{\partial y_k}{\partial a_j}\frac{\partial a_j}{\partial x_i} \\ &= \sum_j w_{ji}\frac{\partial y_k}{\partial a_j} \end{aligned} \tag{8.29}$$

where we have made use of (8.5). The sum in (8.29) runs over all units j to which the input unit i sends connections (for example, over all units in the first hidden layer in the layered topology considered earlier). We now write down a recursive backpropagation formula for the derivatives $\partial y_k/\partial a_j$:

$$\begin{aligned} \frac{\partial y_k}{\partial a_j} &= \sum_l \frac{\partial y_k}{\partial a_l}\frac{\partial a_l}{\partial a_j} \\ &= h'(a_j)\sum_l w_{lj}\frac{\partial y_k}{\partial a_l} \end{aligned} \tag{8.30}$$

where the sum runs over all units l to which unit j sends connections (corresponding to the first index of w_{lj}). Again, we have made use of (8.5) and (8.6). This back-propagation starts at the output units, for which the required derivatives can be found directly from the functional form of the output-unit activation function. For linear output units, we have

$$\frac{\partial y_k}{\partial a_l} = \delta_{kl} \tag{8.31}$$

where δ_{kl} are the elements of the identity matrix and are defined by

$$\delta_{kl} = \begin{cases} 1, & \text{if } k = l, \\ 0, & \text{otherwise.} \end{cases} \tag{8.32}$$

Section 3.4

If we have individual logistic sigmoid activation functions at each output unit, then

$$\frac{\partial y_k}{\partial a_l} = \delta_{kl}\sigma'(a_l) \tag{8.33}$$

Section 3.4

whereas for softmax outputs, we have

$$\frac{\partial y_k}{\partial a_l} = \delta_{kl}y_k - y_k y_l. \tag{8.34}$$

We can summarize the procedure for calculating the Jacobian matrix as follows. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be evaluated, and forward propagate in the usual way to obtain the states of all the hidden and output units in the network. Next, for each row k of the Jacobian matrix, corresponding to the output unit k, backpropagate using the recursive relation (8.30), starting with (8.31), (8.33) or (8.34), for all the hidden units in the network. Finally, use (8.29) for the backpropagation to the inputs. The Jacobian can also be evaluated using an alternative *forward* propagation formalism, which can be derived in an analogous way to the backpropagation approach given here.

Exercise 8.5

Again, the implementation of such algorithms can be checked using numerical differentiation in the form

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{8.35}$$

which involves $2D$ forward propagation passes for a network having D inputs and therefore requires $\mathcal{O}(DW)$ steps in total.

8.1.6 The Hessian matrix

We have shown how backpropagation can be used to obtain the first derivatives of an error function with respect to the weights in the network. Backpropagation can also be used to evaluate the second derivatives of the error, which are given by

$$\frac{\partial^2 E}{\partial w_{ji}\partial w_{lk}}. \tag{8.36}$$

It is often convenient to consider all the weight and bias parameters as elements w_i of a single vector, denoted $\mathbf{w}$, in which case the second derivatives form the elements H_{ij} of the *Hessian* matrix $\mathbf{H}$:

$$H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j} \tag{8.37}$$

where $i, j \in \{1, \ldots, W\}$ and W is the total number of weights and biases. The Hessian matrix arises in several nonlinear optimization algorithms used for training neural networks based on considerations of the second-order properties of the error surface (Bishop, 2006). It also plays a role in some Bayesian treatments of neural networks (MacKay, 1992; Bishop, 2006) and has been used to reduce the precision of the weights in large language models to lessen their memory footprint (Shen *et al.*, 2019).

An important consideration for many applications of the Hessian is the efficiency with which it can be evaluated. If there are W parameters (weights and biases) in the network, then the Hessian matrix has dimensions $W \times W$ and so the computational effort needed to evaluate the Hessian will scale like $\mathcal{O}(W^2)$ for each point in the data set. Extension of the backpropagation procedure (Bishop, 1992) allows the Hessian matrix to be evaluated efficiently with a scaling that is indeed $\mathcal{O}(W^2)$. *Exercise 8.6* Sometimes, we do not need the Hessian matrix explicitly but only the product $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ of the Hessian with some vector $\mathbf{v}$, and this product can be calculated efficiently in $\mathcal{O}(W)$ steps using an extension of backpropagation (Møller, 1993; Pearlmutter, 1994).

Since neural networks may contain millions or even billions of parameters, evaluating, or even just storing, the full Hessian matrix for many models is infeasible. Evaluating the inverse of the Hessian is even more demanding as this has $\mathcal{O}(W^3)$ computational scaling. Consequently there is interest in finding effective approximations to the full Hessian.

One approximation involves simply evaluating only the diagonal elements of the Hessian and implicitly setting the off-diagonal elements to zero. This requires $\mathcal{O}(W)$ storage and allows the inverse to be evaluated in $\mathcal{O}(W)$ steps but still requires $\mathcal{O}(W^2)$ computation (Ricotti, Ragazzini, and Martinelli, 1988), although with further approximation this can be reduced to $\mathcal{O}(W)$ steps (Becker and LeCun, 1989; LeCun, Denker, and Solla, 1990). In practice, however, the Hessian generally has significant off-diagonal terms, and so this approximation must be treated with care.

A more convincing approach, known as the *outer product approximation*, is obtained as follows. Consider a regression application using a sum-of-squares error function of the form

$$E = \frac{1}{2}\sum_{n=1}^{N}(y_n - t_n)^2 \tag{8.38}$$

where we have considered a single output to keep the notation simple (the extension to several outputs is straightforward). *Exercise 8.8* We can then write the Hessian matrix in the form

$$\mathbf{H} = \nabla\nabla E = \sum_{n=1}^{N}\nabla y_n(\nabla y_n)^{\mathrm{T}} + \sum_{n=1}^{N}(y_n - t_n)\nabla\nabla y_n \tag{8.39}$$

where ∇ denotes the gradient with respect to $\mathbf{w}$. If the network has been trained on the data set and its outputs y_n are very close to the target values t_n, then the final term in (8.39) will be small and can be neglected. More generally, however, it may be appropriate to neglect this term based on the following argument. Recall from Section 4.2 that the optimal function that minimizes a sum-of-squares loss is the conditional average of the target data. The quantity $(y_n - t_n)$ is then a random variable with zero mean. If we assume that its value is uncorrelated with the value of the second derivative term on the right-hand side of (8.39), then the whole term will average to zero in the summation over n.

Exercise 8.9

By neglecting the second term in (8.39), we arrive at the *Levenberg–Marquardt* approximation, also known as the *outer product* approximation because the Hessian matrix is built up from a sum of outer products of vectors, given by

$$\mathbf{H} \simeq \sum_{n=1}^{N} \nabla a_n \nabla a_n^{\mathrm{T}}. \tag{8.40}$$

Evaluating the outer product approximation for the Hessian is straightforward as it involves only first derivatives of the error function, which can be evaluated efficiently in $\mathcal{O}(W)$ steps using standard backpropagation. The elements of the matrix can then be found in $\mathcal{O}(W^2)$ steps by simple multiplication. It is important to emphasize that this approximation is likely to be valid only for a network that has been trained appropriately, and that for a general network mapping, the second derivative terms on the right-hand side of (8.39) will typically not be negligible.

For a cross-entropy error function for a network with logistic-sigmoid output-unit activation functions, the corresponding approximation is given by

Exercise 8.10

$$\mathbf{H} \simeq \sum_{n=1}^{N} y_n(1 - y_n) \nabla a_n \nabla a_n^{\mathrm{T}}. \tag{8.41}$$

An analogous result can be obtained for multi-class networks having softmax output-unit activation functions. The outer product approximation can also be used to develop an efficient sequential procedure for approximating the inverse of a Hessian (Hassibi and Stork, 1993).

Exercise 8.11
Exercise 8.12

8.2. Automatic Differentiation

We have seen the importance of using gradient information to train neural networks efficiently. There are essentially four ways in which the gradient of a neural network error function can be evaluated.

Section 7.2.1

The first approach, which formed the mainstay of neural networks for many years, is to derive the backpropagation equations by hand and then to implement them explicitly in software. If this is done carefully it results in efficient code that gives precise results that are accurate to numerical precision. However, the process of deriving the equations as well as the process of coding them both take time and are prone to errors. It also results in some redundancy in the code because the forward propagation equations are coded separately from the backpropagation equations. As these often involve duplicated calculations, then if the model is altered, both the forward and backward implementations need to be changed in unison. This effort can easily become a limitation on how quickly and effectively different architectures can be explored empirically.

A second approach is to evaluate the gradients numerically using finite differences. This requires only a software implementation of the forward propagation equations. One problem with numerical differentiation is that it has limited computational accuracy, although this is unlikely to be an issue for network training as we may be using stochastic gradient descent in which each evaluation is only a very noisy estimate of the local gradient. The main drawback of this approach is that it scales poorly with the size of the network. However, the technique is useful for debugging other approaches, because the gradients are evaluated using only the forward propagation code and so can be used to confirm the correctness of backpropagation or other code used to evaluate gradients.

Section 8.1.4

A third approach is called *symbolic differentiation* and makes use of specialist software to automate the analytical manipulations that are done by hand in the first approach. This process is an example of *computer algebra* or *symbolic computation* and involves the automatic application of the rules of calculus, such as the chain rule, in a completely mechanistic process. The resulting expressions are then implemented in standard software. An obvious advantage of this approach is that it avoids human error in the manual derivation of the backpropagation equations. Moreover, the gradients are again calculated to machine precision, and the poor scaling seen with numerical differentiation is avoided. The major downside of symbolic differentiation, however, is that the resulting expressions for derivatives can become exponentially longer than the original function, with correspondingly long evaluation times. Consider a function $f(x)$ given by the product of $u(x)$ and $v(x)$. The function and its derivative are given by

$$f(x) = u(x)v(x) \tag{8.42}$$

$$f'(x) = u'(x)v(x) + u(x)v'(x). \tag{8.43}$$

We see that there is redundant computation in that $u(x)$ and $v(x)$ must be evaluated both for the calculation of $f(x)$ and for $f'(x)$. If the factors $u(x)$ and $v(x)$ themselves involve factors, then we end up with a nested duplication of expressions, which rapidly grow in complexity. This problem is called *expression swell*.

As a further illustration, consider a function that is structured like two layers of a neural network (Grosse, 2018) with a single input x, a hidden unit with activation z, and an output y in which

$$z = h(w_1 x + b_1) \tag{8.44}$$

$$y = h(w_2 z + b_2) \tag{8.45}$$

where $h(a)$ is the soft ReLU:

$$\zeta(a) = \ln\left(1 + \exp(a)\right). \tag{8.46}$$

The overall function is therefore given by

$$y(x) = h\left(w_2 h(w_1 x + b_1) + b_2\right) \tag{8.47}$$

Exercise 8.13

and the derivative of the network output with respect to w_1, evaluated symbolically, is given by

$$\frac{\partial y}{\partial w_1} = \frac{w_2 x \exp\left(w_1 x + b_1 + b_2 + w_2 \ln[1 + e^{w_1 x + b_1}]\right)}{(1 + e^{w_1 x + b_1})\,(1 + \exp(b_2 + w_2 \ln[1 + e^{w_1 x + b_1}]))}. \tag{8.48}$$

As well as being significantly more complex than the original function, we also see redundant computation where expressions such as $w_1 x + b_1$ occur in several places.

A further major drawback with symbolic differentiation is that it requires that the expression to be differentiated is expressed in closed form. It therefore excludes important control flow operations such as loops, recursions, conditional execution, and procedure calls, which are valuable constructs that we might wish to use when defining the network function.

We therefore turn to the fourth technique for evaluating derivatives in neural networks called *automatic differentiation*, also known as 'autodiff' or 'algorithmic differentiation' (Baydin *et al.*, 2018). Unlike symbolic differentiation, the goal of automatic differentiation is not to find a mathematical expression for the derivatives but to have the computer automatically generate the code that implements the gradient calculations given only the code for the forward propagation equations. It is accurate to machine precision, just as with symbolic differentiation, but is more efficient because it is able to exploit intermediate variables used in the definition of the forward propagation equations and thereby avoid redundant evaluations. It is important to note that not only can automatic differentiation handle conventional closed-form mathematical expressions but it can also deal with flow control elements such as branches, loops, recursion, and procedure calls, and is therefore significantly more powerful than symbolic differentiation. Automatic differentiation is a well-established field with broad applicability that was developed largely outside of the machine learning community. Modern deep learning is a largely empirical process, involving evaluating and comparing different architectures, and automatic differentiation therefore plays a key role in enabling this experimentation to be done accurately and efficiently.

The key idea of automatic differentiation is to take the code that evaluates a function, for example the forward propagation equations that evaluate the error function for a neural network, and augment the code with additional variables whose values are accumulated during code execution to obtain the required derivatives. There are two principal forms of automatic differentiation, known as forward mode and reverse mode. We start by looking at forward mode, which is conceptually somewhat simpler.

8.2.1 Forward-mode automatic differentiation

In forward-mode automatic differentiation, we augment each intermediate variable z_i, known as a 'primal' variable, involved in the evaluation of a function, such as the error function of a neural network, with an additional variable representing the value of some derivative of that variable, which we can denote $\dot{z}_i$, known as a 'tangent' variable. The tangent variables and their associated code are generated

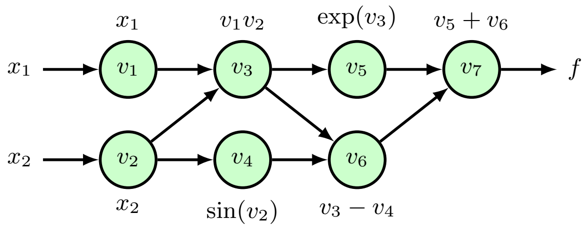


Figure 8.4 Evaluation trace diagram showing the steps involved in the numerical evaluation of the function (8.49) using the primal equations (8.50) to (8.56).

automatically by the software environment. Instead of simply doing forward propagation to compute $\{z_i\}$, the code now propagates tuples $(z_i, \dot{z}_i)$ so that variables and derivatives are evaluated in parallel. The original function is generally defined in terms of elementary operators consisting of arithmetic operations and negation as well as transcendental functions such as exponential, logarithm, and trigonometric functions, all of which have simple formulae for their derivatives. Using these derivatives in combination with the chain rule of calculus allows the code used to evaluate gradients to be constructed automatically.

As an example, consider the following function, which has two input variables:

$$f(x_1, x_2) = x_1x_2 + \exp(x_1x_2) - \sin(x_2). \tag{8.49}$$

When implemented in software, the code consists of a sequence of operations that can be expressed as an *evaluation trace* of the underlying elementary operations. This trace can be visualized in the form of a graph, as shown in Figure 8.4. Here we have defined the following primal variables

$$v_1 = x_1 \tag{8.50}$$
$$v_2 = x_2 \tag{8.51}$$
$$v_3 = v_1v_2 \tag{8.52}$$
$$v_4 = \sin(v_2) \tag{8.53}$$
$$v_5 = \exp(v_3) \tag{8.54}$$
$$v_6 = v_3 - v_4 \tag{8.55}$$
$$v_7 = v_5 + v_6. \tag{8.56}$$

Now suppose we wish to evaluate the derivative $\partial f/\partial x_1$. We define the tangent variables by $\dot{v}_i = \partial v_i/\partial x_1$. Expressions for evaluating these can be constructed automatically using the chain rule of calculus:

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} = \sum_{j \in \mathrm{pa}(i)} \frac{\partial v_j}{\partial x_1}\frac{\partial v_i}{\partial v_j} = \sum_{j \in \mathrm{pa}(i)} \dot{v}_j \frac{\partial v_i}{\partial v_j}. \tag{8.57}$$

where $\mathrm{pa}(i)$ denotes the set of *parents* of the node i in the evaluation trace diagram, that is the set of variables with arrows pointing to node i. For example, in Figure 8.4 the parents of node v_3 are nodes v_1 and v_2. Applying (8.57) to the evaluation trace

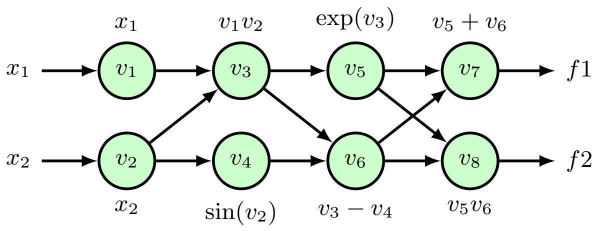


Figure 8.5 Extension of the example shown in Figure 8.4 to a function with two outputs f_1 and f_2.

equations (8.50) to (8.56), we obtain the following evaluation trace equations for the tangent variables

$$\dot{v}_1 = 1 \tag{8.58}$$
$$\dot{v}_2 = 0 \tag{8.59}$$
$$\dot{v}_3 = v_1\dot{v}_2 + \dot{v}_1 v_2 \tag{8.60}$$
$$\dot{v}_4 = \dot{v}_2 \cos(v_2) \tag{8.61}$$
$$\dot{v}_5 = \dot{v}_3 \exp(v_3) \tag{8.62}$$
$$\dot{v}_6 = \dot{v}_3 - \dot{v}_4 \tag{8.63}$$
$$\dot{v}_7 = \dot{v}_5 + \dot{v}_6. \tag{8.64}$$

We can summarize automatic differentiation for this example as follows. We first write code to implement the evaluation of the primal variables, given by (8.50) to (8.56). The associated equations and corresponding code for evaluating the tangent variables (8.58) to (8.64) are generated automatically. To evaluate the derivative $\partial f/\partial x_1$, we input specific values of x_1 and x_2 and the code then executes the primal and tangent equations, numerically evaluating the tuples $(v_i, \dot{v}_i)$ in sequence until we obtain $\dot{v}_5$, which is the required derivative.

Exercise 8.17

Now consider an example with two outputs $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ where $f_1(x_1, x_2)$ is defined by (8.49) and

$$f_2(x_1, x_2) = (x_1x_2 - \sin(x_2)) \exp(x_1x_2) \tag{8.65}$$

as illustrated by the evaluation trace diagram in Figure 8.5. We see that this involves only a small extension to the evaluation equations for the primal and tangent variables, and so both $\partial f_1/\partial x_1$ and $\partial f_2/\partial x_1$ can be evaluated together in a single forward pass. The downside, however, is that if we wish to evaluate derivatives with respect to a different input variable x_2 then we have to run a separate forward pass. In general, if we have a function with D inputs and K outputs then a single pass of forward-mode automatic differentiation produces a single column of the $K \times D$ Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_K}{\partial x_1} & \cdots & \dfrac{\partial f_K}{\partial x_D} \end{bmatrix}. \tag{8.66}$$

To compute column j of the Jacobian, we need to initialize the forward pass of the tangent equations by setting $\dot{x}_j = 1$ and $\dot{x}_i = 0$ for $i \neq j$. We can write this in vector form as $\dot{\mathbf{x}} = \mathbf{e}_i$ where $\mathbf{e}_i$ is the ith unit vector. To compute the full Jacobian matrix we need D forward-mode passes. However, if we wish to evaluate the product of the Jacobian with a vector $\mathbf{r} = (r_1, \ldots, r_D)^{\mathrm{T}}$:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_K}{\partial x_1} & \cdots & \frac{\partial f_K}{\partial x_D} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_D \end{bmatrix} \tag{8.67}$$

Exercise 8.18 then this can be done in single forward pass by setting $\dot{\mathbf{x}} = \mathbf{r}$.

We see that forward-mode automatic differentiation can evaluate the full $K \times D$ Jacobian matrix of derivatives using D forward passes. This is very efficient for networks with a few inputs and many outputs, such that $K \gg D$. However, we often operate in a regime where we often have just one function, namely the error function used for training, and large numbers of variables that we want to differentiate with respect to, comprising the weights and biases in the network, of which there may be millions or billions. In such situations, forward-mode automatic differentiation is extremely inefficient. We therefore turn to an alternative version of automatic differentiation based on the a backwards flow of derivative data through the evaluation trace graph.

8.2.2 Reverse-mode automatic differentiation

We can think of reverse-mode automatic differentiation as a generalization of the error backpropagation procedure. As with forward mode, we augment each intermediate variable v_i with additional variables, in this case called *adjoint* variables, denoted $\overline{v}_i$. Consider again a situation with a single output function f for which the adjoint variables are defined by

$$\overline{v}_i = \frac{\partial f}{\partial v_i}. \tag{8.68}$$

These can be evaluated sequentially starting with the output and working backwards by using the chain rule of calculus:

$$\overline{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j \in \mathrm{ch}(i)} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \sum_{j \in \mathrm{ch}(i)} \overline{v}_j \frac{\partial v_j}{\partial v_i}. \tag{8.69}$$

Here $\mathrm{ch}(i)$ denotes the *children* of node i in the evaluation trace graph, in other words the set of nodes that have arrows pointing into them from node i. The successive evaluation of the adjoint variables represents a flow of information backwards *Figure 8.1* through the graph, as we saw previously.

If we again consider the specific example function given by (8.50) to (8.56), we *Exercise 8.16* obtain the following evaluation equations for the evaluation of the adjoint variables

$$\overline{v}_7 = 1 \tag{8.70}$$

$$\overline{v}_6 = \overline{v}_7 \tag{8.71}$$

$$\overline{v}_5 = \overline{v}_7 \tag{8.72}$$

$$\overline{v}_4 = -\overline{v}_6 \tag{8.73}$$

$$\overline{v}_3 = \overline{v}_5 v_5 + \overline{v}_6 \tag{8.74}$$

$$\overline{v}_2 = \overline{v}_2 v_1 + \overline{v}_4 \cos(v_2) \tag{8.75}$$

$$\overline{v}_1 = \overline{v}_3 v_2. \tag{8.76}$$

Note that these start at the output and then flow backwards through the graph to the inputs. Even with multiple inputs, only a single backward pass is required to evaluate the derivatives. For a neural network error function, the derivatives of E with respect to the weight and biases are obtained as the corresponding adjoint variables. However, if we now have more than one output then we need to run a separate backward pass for each output variable.

Figure 8.5

Reverse mode is often more memory intensive than forward mode because all of the intermediate primal variables must be stored so that they will be available as needed when evaluating the adjoint variables during the backward pass. By contrast, with forward mode, the primal and tangent variables are computed together during the forward pass, and therefore variables can be discarded once they have been used. It is therefore also generally easier to implement forward mode compared to reverse mode.

For both forward-mode and reverse-mode automatic differentiation, a single pass through the network is guaranteed to take no more than 6 times the computational cost of a single function evaluation. In practice, the overhead is typically closer to a factor of 2 or 3 (Griewank and Walther, 2008). Hybrids of forward and reverse modes are also of interest. One situation in which this arises is in the evaluation of the product of a Hessian matrix with a vector, which can be calculated without explicit evaluation of the full Hessian (Pearlmutter, 1994). Here we can use reverse mode to calculate the gradient of code, which itself has been generated by the forward model. We start from a vector $\mathbf{b}$ and a point $\mathbf{x}$ at which the Hessian–vector product is to be evaluated. By setting $\dot{\mathbf{x}} = \mathbf{v}$ and using forward mode, we obtain the directional derivative $\mathbf{v}^{\mathrm{T}}\nabla f$. This is then differentiated using reverse mode to obtain $\nabla^2 f\mathbf{v} = \mathbf{H}\mathbf{v}$. If W is the number of parameters in the neural network then this evaluation has $\mathcal{O}(W)$ complexity even though the Hessian is of size $W \times W$. The Hessian itself can also be evaluated explicitly using automatic differentiation but this has $\mathcal{O}(W^2)$ complexity.

Exercises

8.1 ($\star$) By making use of (8.5), (8.6), (8.8), and (8.12), verify the backpropagation formula (8.13) for evaluating the derivatives of an error function.

8.2 ($\star\star$) Consider a network that consists of layers and rewrite the backpropagation formula (8.13) in matrix notation by starting with the forward propagation equation (6.19). Note that the result involves multiplication by the transposes of the matrices.