Chapter 7

Gradients and initialization

Chapter 6 introduced iterative optimization algorithms. These are general-purpose methods for finding the minimum of a function. In the context of neural networks, they find parameters that minimize the loss so that the model accurately predicts the training outputs from the inputs. The basic approach is to choose initial parameters randomly and then make a series of small changes that decrease the loss on average. Each change is based on the gradient of the loss with respect to the parameters at the current position.

This chapter discusses two issues that are specific to neural networks. First, we consider how to calculate the gradients efficiently. This is a serious challenge since the largest models at the time of writing have $\sim 10^{12}$ parameters, and the gradient needs to be computed for every parameter at every iteration of the training algorithm. Second, we consider how to initialize the parameters. If this is not done carefully, the initial losses and their gradients can be extremely large or small. In either case, this impedes the training process.

7.1 Problem definitions

Consider a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with multivariate input $\mathbf{x}$, parameters $\boldsymbol{\phi}$, and three hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \\ \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3, \end{aligned} \tag{7.1}$$

where the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of the input. The model parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_0, \boldsymbol{\Omega}_0, \boldsymbol{\beta}_1, \boldsymbol{\Omega}_1, \boldsymbol{\beta}_2, \boldsymbol{\Omega}_2, \boldsymbol{\beta}_3, \boldsymbol{\Omega}_3\}$ consist of the bias vectors $\boldsymbol{\beta}_k$ and weight matrices $\boldsymbol{\Omega}_k$ between every layer (figure 7.1).

We also have individual loss terms ℓ_i, which return the negative log-likelihood of the ground truth label y_i given the model prediction $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for training input $\mathbf{x}_i$. For example, this might be the least squares loss $\ell_i = (\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}] - y_i)^2$. The total loss is the sum of these terms over the training data:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i. \tag{7.2}$$

The most commonly used optimization algorithm for training neural networks is stochastic gradient descent (SGD), which updates the parameters as:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{7.3}$$

where α is the learning rate, and $\mathcal{B}_t$ contains the batch indices at iteration t. To compute this update, we need to calculate the derivatives:

$$\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} \quad \text{and} \quad \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k}, \tag{7.4}$$

for the parameters $\{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$ at every layer $k \in \{0, 1, \ldots, K\}$ and for each index i in the batch. The first part of this chapter describes the *backpropagation algorithm*, which computes these derivatives efficiently.

Problem 7.1

In the second part of the chapter, we consider how to initialize the network parameters before we commence training. We describe methods to choose the initial weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$ so that training is stable.

7.2 Computing derivatives

The derivatives of the loss tell us how the loss changes when we make a small change to the parameters. Optimization algorithms exploit this information to manipulate the parameters so that the loss becomes smaller. The *backpropagation algorithm* computes these derivatives. The mathematical details are somewhat involved, so we first make two observations that provide some intuition.

Observation 1: Each weight (element of $\boldsymbol{\Omega}_k$) multiplies the activation at a source hidden unit and adds the result to a destination hidden unit in the next layer. It follows that the effect of any small change to the weight is amplified or attenuated by the activation at the source hidden unit. Hence, we run the network for each data example in the batch and store the activations of all the hidden units. This is known as the *forward pass* (figure 7.1). The stored activations will subsequently be used to compute the gradients.

Observation 2: A small change in a bias or weight causes a ripple effect of changes through the subsequent network. The change modifies the value of its destination hidden

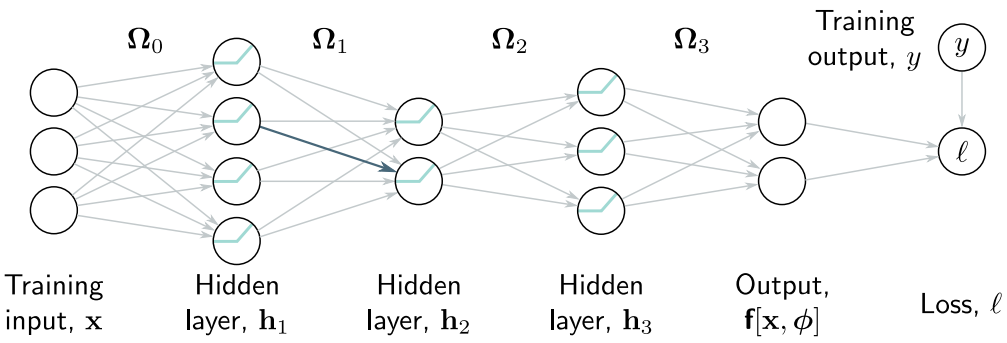

Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss ℓ with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the blue weight is applied to the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the blue weight will double too. Hence, to compute the derivatives of the weights, we need to calculate and store the activations at the hidden layers. This is known as the *forward pass* since it involves running the network equations sequentially.

unit. This, in turn, changes the values of the hidden units in the subsequent layer, which will change the hidden units in the layer after that, and so on, until a change is made to the model output and, finally, the loss.

Hence, to know how changing a parameter modifies the loss, we also need to know how changes to every subsequent hidden layer will, in turn, modify their successor. These same quantities are required when considering other parameters in the same or earlier layers. It follows that we can calculate them once and reuse them. For example, consider computing the effect of a small change in weights that feed into hidden layers $\mathbf{h}_3$, $\mathbf{h}_2$, and $\mathbf{h}_1$, respectively:

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_3$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_3$ changes the model output $\mathbf{f}$, and (ii) how a change in this output changes the loss ℓ (figure 7.2a).

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_2$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_2$ affects $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes the model output, and (iii) how this output changes the loss (figure 7.2b).

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_1$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_1$ affects layer $\mathbf{h}_2$, (ii) how a change in layer $\mathbf{h}_2$ affects layer $\mathbf{h}_3$, (iii) how layer $\mathbf{h}_3$ changes the model output, and (iv) how the model output changes the loss (figure 7.2c).

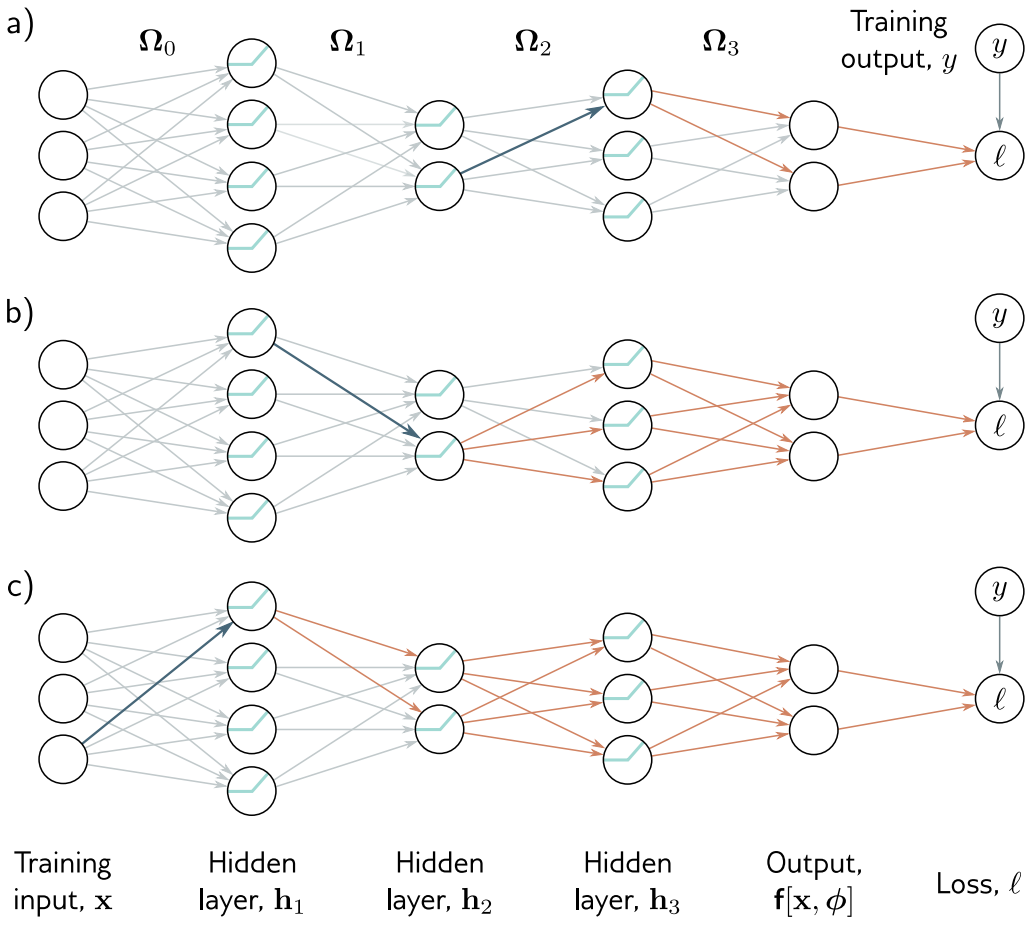

Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

As we move backward through the network, we see that most of the terms we need were already calculated in the previous step, so we do not need to re-compute them. Proceeding backward through the network in this way to compute the derivatives is known as the *backward pass*.

The ideas behind backpropagation are relatively easy to understand. However, the derivation requires matrix calculus because the bias and weight terms are vectors and matrices, respectively. To help grasp the underlying mechanics, the following section derives backpropagation for a simpler toy model with scalar parameters. We then apply the same approach to a deep neural network in section 7.4.

7.3 Toy example

Consider a model $\mathrm{f}[x, \boldsymbol{\phi}]$ with eight scalar parameters $\boldsymbol{\phi} = \{\beta_0, \omega_0, \beta_1, \omega_1, \beta_2, \omega_2, \beta_3, \omega_3\}$ that consists of a composition of the functions $\sin[\bullet], \exp[\bullet]$, and $\cos[\bullet]$:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x]\big]\Big], \tag{7.5}$$

and a least squares loss function $L[\boldsymbol{\phi}] = \sum_i \ell_i$ with individual terms:

$$\ell_i = (\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i)^2, \tag{7.6}$$

where, as usual, x_i is the i^{th} training input, and y_i is the i^{th} training output. You can think of this as a simple neural network with one input, one output, one hidden unit at each layer, and different activation functions $\sin[\bullet], \exp[\bullet]$, and $\cos[\bullet]$ between each layer.

We aim to compute the derivatives:

$$\frac{\partial \ell_i}{\partial \beta_0}, \quad \frac{\partial \ell_i}{\partial \omega_0}, \quad \frac{\partial \ell_i}{\partial \beta_1}, \quad \frac{\partial \ell_i}{\partial \omega_1}, \quad \frac{\partial \ell_i}{\partial \beta_2}, \quad \frac{\partial \ell_i}{\partial \omega_2}, \quad \frac{\partial \ell_i}{\partial \beta_3}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial \omega_3}.$$

Of course, we could find expressions for these derivatives by hand and compute them directly. However, some of these expressions are quite complex. For example:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \omega_0} = \; & -2\Big(\beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\big]\Big] - y_i\Big) \\ & \cdot \omega_1\omega_2\omega_3 \cdot x_i \cdot \cos[\beta_0 + \omega_0 \cdot x_i] \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big] \\ & \cdot \sin\Big[\beta_2 + \omega_2 \cdot \exp\Big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\Big]\Big].\end{aligned} \tag{7.7}$$

Such expressions are awkward to derive and code without mistakes and do not exploit the inherent redundancy; notice that the three exponential terms are the same.

The backpropagation algorithm is an efficient method for computing all of these derivatives at once. It consists of (i) a forward pass, in which we compute and store a series of intermediate values and the network output, and (ii) a backward pass, in which

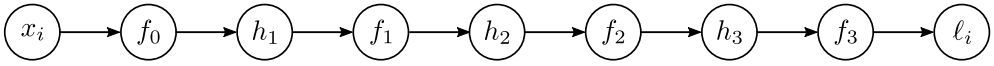

Figure 7.3 Backpropagation forward pass. We compute and store each of the intermediate variables in turn until we finally calculate the loss.

we calculate the derivatives of each parameter, starting at the end of the network, and reusing previous calculations as we move toward the start.

Forward pass: We treat the computation of the loss as a series of calculations:

$$\begin{aligned} f_0 &= \beta_0 + \omega_0 \cdot x_i \\ h_1 &= \sin[f_0] \\ f_1 &= \beta_1 + \omega_1 \cdot h_1 \\ h_2 &= \exp[f_1] \\ f_2 &= \beta_2 + \omega_2 \cdot h_2 \\ h_3 &= \cos[f_2] \\ f_3 &= \beta_3 + \omega_3 \cdot h_3 \\ \ell_i &= (f_3 - y_i)^2. \end{aligned} \tag{7.8}$$

We compute and store the values of the intermediate variables f_k and h_k (figure 7.3).

Backward pass #1: We now compute the derivatives of ℓ_i with respect to these intermediate variables, but in reverse order:

$$\frac{\partial \ell_i}{\partial f_3}, \quad \frac{\partial \ell_i}{\partial h_3}, \quad \frac{\partial \ell_i}{\partial f_2}, \quad \frac{\partial \ell_i}{\partial h_2}, \quad \frac{\partial \ell_i}{\partial f_1}, \quad \frac{\partial \ell_i}{\partial h_1}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial f_0}. \tag{7.9}$$

The first of these derivatives is straightforward:

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y_i). \tag{7.10}$$

The next derivative can be calculated using the chain rule:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}. \tag{7.11}$$

The left-hand side asks how ℓ_i changes when h_3 changes. The right-hand side says we can decompose this into (i) how f_3 changes when h_3 changes and (ii) how ℓ_i changes when f_3 changes. In the original equations, h_3 changes f_3, which changes ℓ_i, and the derivatives

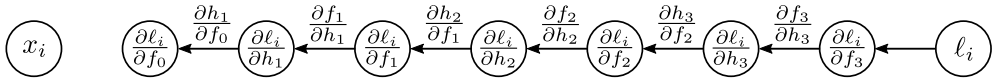

Figure 7.4 Backpropagation backward pass #1. We work backward from the end of the function computing the derivatives $\partial \ell_i / \partial f_k$ and $\partial \ell_i / \partial h_k$ of the loss with respect to the intermediate quantities. Each derivative is computed from the previous one by multiplying by terms of the form $\partial f_k / \partial h_k$ or $\partial h_k / \partial f_{k-1}$.

represent the effects of this chain. Notice that we already computed the second of these derivatives, and the other is the derivative of $\beta_3 + \omega_3 \cdot h_3$ with respect to h_3, which is ω_3.

We continue in this way, computing the derivatives of the output with respect to these intermediate quantities (figure 7.4):

$$\begin{aligned}
\frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right).
\end{aligned} \tag{7.12}$$

Problem 7.2

In each case, we have already computed the quantities in the brackets in the previous step, and the last term has a simple expression. These equations embody Observation 2 from the previous section (figure 7.2); we can reuse the previously computed derivatives if we calculate them in reverse order.

Backward pass #2: Finally, we consider how the loss ℓ_i changes when we change the parameters $\{\beta_k\}$ and $\{\omega_k\}$. Once more, we apply the chain rule (figure 7.5):

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\
\frac{\partial \ell_i}{\partial \omega_k} &= \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}.
\end{aligned} \tag{7.13}$$

In each case, the second term on the right-hand side was computed in equation 7.12. When $k > 0$, we have $f_k = \beta_k + \omega_k \cdot h_k$, so:

$$\frac{\partial f_k}{\partial \beta_k} = 1 \qquad \text{and} \qquad \frac{\partial f_k}{\partial \omega_k} = h_k. \tag{7.14}$$

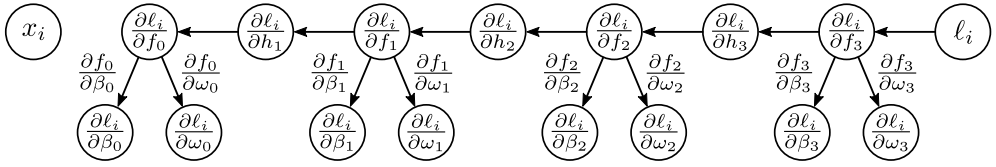

Figure 7.5 Backpropagation backward pass #2. Finally, we compute the derivatives $\partial \ell_i / \partial \beta_k$ and $\partial \ell_i / \partial \omega_k$. Each derivative is computed by multiplying the term $\partial \ell_i / \partial f_k$ by $\partial f_k / \partial \beta_k$ or $\partial f_k / \partial \omega_k$ as appropriate.

This is consistent with Observation 1 from the previous section; the effect of a change in the weight ω_k is proportional to the value of the source variable h_k (which was stored in the forward pass). The final derivatives from the term $f_0 = \beta_0 + \omega_0 \cdot x_i$ are:

Notebook 7.1 Backpropagation in toy model

$$\frac{\partial f_0}{\partial \beta_0} = 1 \qquad \text{and} \qquad \frac{\partial f_0}{\partial \omega_0} = x_i. \tag{7.15}$$

Backpropagation is both simpler and more efficient than computing the derivatives individually, as in equation 7.7.[1]

7.4 Backpropagation algorithm

Now we repeat this process for a three-layer network (figure 7.1). The intuition and much of the algebra are identical. The main differences are that intermediate variables $\mathbf{f}_k, \mathbf{h}_k$ are vectors, the biases $\boldsymbol{\beta}_k$ are vectors, the weights $\boldsymbol{\Omega}_k$ are matrices, and we are using ReLU functions rather than simple algebraic functions like $\cos[\bullet]$.

Forward pass: We write the network as a series of sequential calculations:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_1 &= \mathbf{a}[\mathbf{f}_0] \\
\mathbf{f}_1 &= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1 \\
\mathbf{h}_2 &= \mathbf{a}[\mathbf{f}_1] \\
\mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2 \\
\mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] \\
\mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3 \\
\ell_i &= \mathrm{l}[\mathbf{f}_3, y_i],
\end{aligned} \tag{7.16}$$

[1] Note that we did not actually need the derivatives $\partial l_i / \partial h_k$ of the loss with respect to the activations. In the final backpropagation algorithm, we will not compute these explicitly.

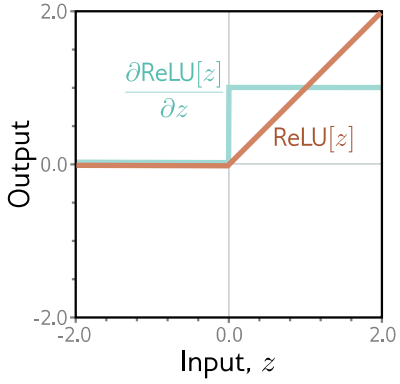

Figure 7.6 Derivative of rectified linear unit. The rectified linear unit (orange curve) returns zero when the input is less than zero and returns the input otherwise. Its derivative (cyan curve) returns zero when the input is less than zero (since the slope here is zero) and one when the input is greater than zero (since the slope here is one).

where $\mathbf{f}_{k-1}$ represents the pre-activations at the k^{th} hidden layer (i.e., the values before the ReLU function $\mathbf{a}[\bullet]$) and $\mathbf{h}_k$ contains the activations at the k^{th} hidden layer (i.e., after the ReLU function). The term $l[\mathbf{f}_3, y_i]$ represents the loss function (e.g., least squares or binary cross-entropy loss). In the forward pass, we work through these calculations and store all the intermediate quantities.

Backward pass #1: Now let's consider how the loss changes when the pre-activations $\mathbf{f}_0, \mathbf{f}_1, \mathbf{f}_2$ change. Applying the chain rule, the expression for the derivative of the loss ℓ_i with respect to $\mathbf{f}_2$ is:

Appendix B.5 Matrix calculus

$$\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}. \tag{7.17}$$

The three terms on the right-hand side have sizes $D_3 \times D_3, D_3 \times D_f$, and $D_f \times 1$, respectively, where D_3 is the number of hidden units in the third layer, and D_f is the dimensionality of the model output $\mathbf{f}_3$.

Similarly, we can compute how the loss changes when we change $\mathbf{f}_1$ and $\mathbf{f}_0$:

$$\frac{\partial \ell_i}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}\right) \tag{7.18}$$

$$\frac{\partial \ell_i}{\partial \mathbf{f}_0} = \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_0}\frac{\partial \mathbf{f}_1}{\partial \mathbf{h}_1}\left(\frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}\right). \tag{7.19}$$

Problem 7.3

Note that in each case, the term in brackets was computed in the previous step. By working backward through the network, we can reuse the previous computations.

Problems 7.4–7.5

Moreover, the terms themselves are simple. Working backward through the right-hand side of equation 7.17, we have:

- The derivative $\partial \ell_i / \partial \mathbf{f}_3$ of the loss ℓ_i with respect to the network output $\mathbf{f}_3$ will depend on the loss function but usually has a simple form.
- The derivative $\partial \mathbf{f}_3 / \partial \mathbf{h}_3$ of the network output with respect to hidden layer $\mathbf{h}_3$ is:

$$\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} = \frac{\partial}{\partial \mathbf{h}_3}\left(\boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3\right) = \boldsymbol{\Omega}_3^T. \tag{7.20}$$

If you are unfamiliar with matrix calculus, this result is not obvious. It is explored in problem 7.6.

Problem 7.6

- The derivative $\partial \mathbf{h}_3/\partial \mathbf{f}_2$ of the output $\mathbf{h}_3$ of the activation function with respect to its input $\mathbf{f}_2$ will depend on the activation function. It will be a diagonal matrix since each activation only depends on the corresponding pre-activation. For ReLU functions, the diagonal terms are zero everywhere $\mathbf{f}_2$ is less than zero and one otherwise (figure 7.6). Rather than multiply by this matrix, we extract the diagonal terms as a vector $\mathbb{I}[\mathbf{f}_2 > 0]$ and pointwise multiply, which is more efficient.

Problems 7.7–7.8

The terms on the right-hand side of equations 7.18 and 7.19 have similar forms. As we progress back through the network, we alternately (i) multiply by the transpose of the weight matrices $\boldsymbol{\Omega}_k^T$ and (ii) threshold based on the inputs $\mathbf{f}_{k-1}$ to the hidden layer. These inputs were stored during the forward pass.

Backward pass #2: Now that we know how to compute $\partial \ell_i/\partial \mathbf{f}_k$, we can focus on calculating the derivatives of the loss with respect to the weights and biases. To calculate the derivatives of the loss with respect to the biases $\boldsymbol{\beta}_k$, we again use the chain rule:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\beta}_k}\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial}{\partial \boldsymbol{\beta}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial \ell_i}{\partial \mathbf{f}_k},\end{aligned} \tag{7.21}$$

which we already calculated in equations 7.17 and 7.18.

Similarly, the derivative for the weights matrix $\boldsymbol{\Omega}_k$, is given by:

$$\begin{aligned}\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\Omega}_k}\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial}{\partial \boldsymbol{\Omega}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial \ell_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial \ell_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T.\end{aligned} \tag{7.22}$$

Again, the progression from line two to line three is not obvious and is explored in problem 7.9. However, the result makes sense. The final line is a matrix of the same size as $\boldsymbol{\Omega}_k$. It depends linearly on $\mathbf{h}_k$, which was multiplied by $\boldsymbol{\Omega}_k$ in the original expression. This is also consistent with the initial intuition that the derivative of the weights in $\boldsymbol{\Omega}_k$ will be proportional to the values of the hidden units $\mathbf{h}_k$ that they multiply. Recall that we already computed these during the forward pass.

Problem 7.9

7.4.1 Backpropagation algorithm summary

We now briefly summarize the final backpropagation algorithm. Consider a deep neural network $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that takes input $\mathbf{x}_i$, has K hidden layers with ReLU activations, and individual loss term $\ell_i = \mathrm{l}[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], \mathbf{y}_i]$. The goal of backpropagation is to compute the derivatives $\partial \ell_i / \partial \boldsymbol{\beta}_k$ and $\partial \ell_i / \partial \boldsymbol{\Omega}_k$ with respect to the biases $\boldsymbol{\beta}_k$ and weights $\boldsymbol{\Omega}_k$.

Forward pass: We compute and store the following quantities:

$$\begin{aligned} \mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\ \mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] \qquad\qquad k \in \{1, 2, \ldots, K\} \\ \mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k. \qquad\qquad k \in \{1, 2, \ldots, K\} \end{aligned} \tag{7.23}$$

Backward pass: We start with the derivative $\partial \ell_i / \partial \mathbf{f}_K$ of the loss function ℓ_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned} \frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \qquad\qquad k \in \{K, K-1, \ldots, 1\} \\ \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T \qquad\qquad k \in \{K, K-1, \ldots, 1\} \\ \frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right), \qquad k \in \{K, K-1, \ldots, 1\} \end{aligned} \tag{7.24}$$

where $\odot$ denotes pointwise multiplication, and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned} \frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\ \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T. \end{aligned} \tag{7.25}$$

Problem 7.10

We calculate these derivatives for every training example in the batch and sum them together to retrieve the gradient for the SGD update.

Notebook 7.2 Backpropagation

Note that the backpropagation algorithm is extremely efficient; the most demanding computational step in both the forward and backward pass is matrix multiplication (by $\boldsymbol{\Omega}$ and $\boldsymbol{\Omega}^T$, respectively) which only requires additions and multiplications. However, it is not memory efficient; the intermediate values in the forward pass must all be stored, and this can limit the size of the model we can train.

7.4.2 Algorithmic differentiation

Although it's important to understand the backpropagation algorithm, it's unlikely that you will need to code it in practice. Modern deep learning frameworks such as PyTorch

and TensorFlow calculate the derivatives automatically, given the model specification. This is known as *algorithmic differentiation*.

Each functional component (linear transform, ReLU activation, loss function) in the framework knows how to compute its own derivative. For example, the PyTorch ReLU function $\mathbf{z}_{out} = \mathbf{relu}[\mathbf{z}_{in}]$ knows how to compute the derivative of its output $\mathbf{z}_{out}$ with respect to its input $\mathbf{z}_{in}$. Similarly, a linear function $\mathbf{z}_{out} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{z}_{in}$ knows how to compute the derivatives of the output $\mathbf{z}_{out}$ with respect to the input $\mathbf{z}_{in}$ and with respect to the parameters $\boldsymbol{\beta}$ and $\boldsymbol{\Omega}$. The algorithmic differentiation framework also knows the sequence of operations in the network and thus has all the information required to perform the forward and backward passes.

These frameworks exploit the massive parallelism of modern graphics processing units (GPUs). Computations such as matrix multiplication (which features in both the forward and backward pass) are naturally amenable to parallelization. Moreover, it's possible to perform the forward and backward passes for the entire batch in parallel if the model and intermediate results in the forward pass do not exceed the available memory.

Problem 7.11

Since the training algorithm now processes the entire batch in parallel, the input becomes a multi-dimensional *tensor*. In this context, a tensor can be considered the generalization of a matrix to arbitrary dimensions. Hence, a vector is a 1D tensor, a matrix is a 2D tensor, and a 3D tensor is a 3D grid of numbers. Until now, the training data have been 1D, so the input for backpropagation would be a 2D tensor where the first dimension indexes the batch element and the second indexes the data dimension. In subsequent chapters, we will encounter more complex structured input data. For example, in models where the input is an RGB image, the original data examples are 3D (height $\times$ width $\times$ channel). Here, the input to the learning framework would be a 4D tensor, where the extra dimension indexes the batch element.

7.4.3 Extension to arbitrary computational graphs

We have described backpropagation in a deep neural network that is naturally sequential; we calculate the intermediate quantities $\mathbf{f}_0, \mathbf{h}_1, \mathbf{f}_1, \mathbf{h}_2 \ldots, \mathbf{f}_k$ in turn. However, models need not be restricted to sequential computation. Later in this book, we will meet models with branching structures. For example, we might take the values in a hidden layer and process them through two different sub-networks before recombining.

Problems 7.12–7.13

Fortunately, the ideas of backpropagation still hold if the computational graph is acyclic. Modern algorithmic differentiation frameworks such as PyTorch and TensorFlow can handle arbitrary acyclic computational graphs.

7.5 Parameter initialization

The backpropagation algorithm computes the derivatives that are used by stochastic gradient descent and Adam to train the model. We now address how to initialize the parameters before we start training. To see why this is crucial, consider that during the forward pass, each set of pre-activations $\mathbf{f}_k$ is computed as:

$$\begin{aligned}\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k \\ &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathrm{a}[\mathbf{f}_{k-1}],\end{aligned} \tag{7.26}$$

where $\mathbf{a}[\bullet]$ applies the ReLU functions and $\boldsymbol{\Omega}_k$ and $\boldsymbol{\beta}_k$ are the weights and biases, respectively. Imagine that we initialize all the biases to zero and the elements of $\boldsymbol{\Omega}_k$ according to a normal distribution with mean zero and variance σ^2. Consider two scenarios:

- If the variance σ^2 is very small (e.g., 10^{-5}), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very small; the result will likely have a smaller magnitude than the input. In addition, the ReLU function clips values less than zero, so the range of $\mathbf{h}_k$ will be half that of $\mathbf{f}_{k-1}$. Consequently, the magnitudes of the pre-activations at the hidden layers will get smaller and smaller as we progress through the network.
- If the variance σ^2 is very large (e.g., 10^5), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very large; the result is likely to have a much larger magnitude than the input. The ReLU function halves the range of the inputs, but if σ^2 is large enough, the magnitudes of the pre-activations will still get larger as we progress through the network.

In these two situations, the values at the pre-activations can become so small or so large that they cannot be represented with finite precision floating point arithmetic.

Even if the forward pass is tractable, the same logic applies to the backward pass. Each gradient update (equation 7.24) consists of multiplying by $\boldsymbol{\Omega}^T$. If the values of $\boldsymbol{\Omega}$ are not initialized sensibly, then the gradient magnitudes may decrease or increase uncontrollably during the backward pass. These cases are known as the *vanishing gradient problem* and the *exploding gradient problem*, respectively. In the former case, updates to the model become vanishingly small. In the latter case, they become unstable.

7.5.1 Initialization for forward pass

We now present a mathematical version of the same argument. Consider the computation between adjacent pre-activations $\mathbf{f}$ and $\mathbf{f}'$ with dimensions D_h and $D_{h'}$, respectively:

$$\begin{aligned}\mathbf{h} &= \mathbf{a}[\mathbf{f}], \\ \mathbf{f}' &= \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}\end{aligned} \tag{7.27}$$

where $\mathbf{f}$ represents the pre-activations, $\boldsymbol{\Omega}$, and $\boldsymbol{\beta}$ represent the weights and biases, and $\mathbf{a}[\bullet]$ is the activation function.

Assume the pre-activations f_j in the input layer $\mathbf{f}$ have variance $\sigma_{f_j}^2$. Consider initializing the biases β_i to zero and the weights Ω_{ij} as normally distributed with mean zero and variance σ_Ω^2. Now we derive expressions for the mean and variance of the pre-activations $\mathbf{f}'$ in the subsequent layer.

The expectation (mean) $\mathbb{E}[f_i']$ of the intermediate values f_i' is:

Appendix C.2 Expectation

$$\begin{aligned}\mathbb{E}[f_i'] &= \mathbb{E}\left[\beta_i + \sum_{j=1}^{D_h}\Omega_{ij}h_j\right] \\ &= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}h_j\right] \\ &= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}\right]\mathbb{E}\left[h_j\right] \\ &= 0 + \sum_{j=1}^{D_h} 0 \cdot \mathbb{E}\left[h_j\right] = 0, \end{aligned} \tag{7.28}$$

where D_h is the dimensionality of the input layer $\mathbf{h}$. We have used the rules for manipulating expectations, and we have assumed that the distributions over the hidden units h_j and the network weights Ω_{ij} are independent between the second and third lines.

Appendix C.2.1 Expectation rules

Using this result, we see that the variance $\sigma^2_{f_i'}$ of the pre-activations f_i' is:

$$\begin{aligned}\sigma^2_{f_i'} &= \mathbb{E}[f_i'^2] - \mathbb{E}[f_i']^2 \\ &= \mathbb{E}\left[\left(\beta_i + \sum_{j=1}^{D_h}\Omega_{ij}h_j\right)^2\right] - 0 \\ &= \mathbb{E}\left[\left(\sum_{j=1}^{D_h}\Omega_{ij}h_j\right)^2\right] \\ &= \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}^2\right]\mathbb{E}\left[h_j^2\right] \\ &= \sum_{j=1}^{D_h}\sigma_\Omega^2\mathbb{E}\left[h_j^2\right] = \sigma_\Omega^2\sum_{j=1}^{D_h}\mathbb{E}\left[h_j^2\right], \end{aligned} \tag{7.29}$$

where we have used the variance identity $\sigma^2 = \mathbb{E}[(z-\mathbb{E}[z])^2] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$. We have assumed once more that the distributions of the weights Ω_{ij} and the hidden units h_j are independent between lines three and four.

Appendix C.2.3 Variance identity

Assuming that the input distribution of pre-activations f_j is symmetric about zero, half of these pre-activations will be clipped by the ReLU function, and the second moment $\mathbb{E}[h_j^2]$ will be half the variance σ_f^2 of f_j (see problem 7.14):

Problem 7.14

$$\sigma^2_{f_i'} = \sigma_\Omega^2\sum_{j=1}^{D_h}\frac{\sigma_f^2}{2} = \frac{1}{2}D_h\sigma_\Omega^2\sigma_f^2. \tag{7.30}$$

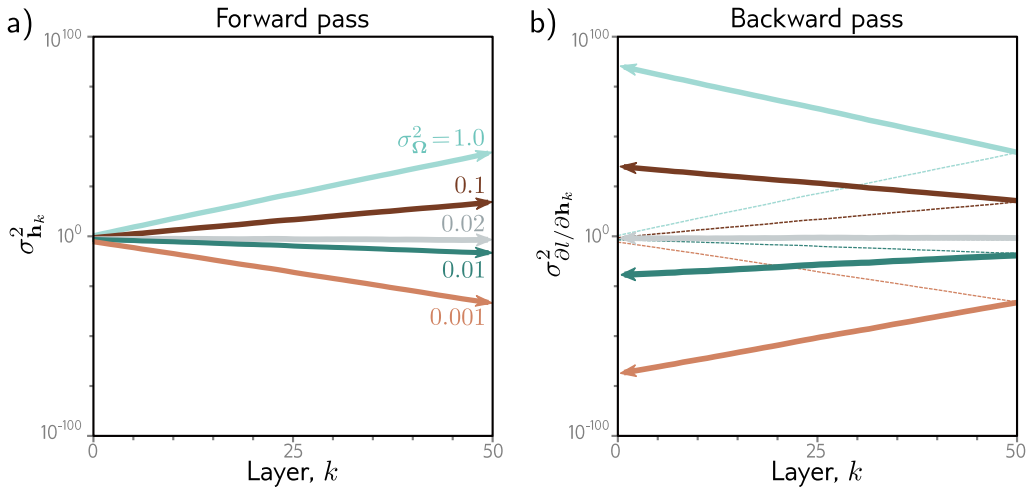

Figure 7.7 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100-dimensional input $\mathbf{x}$ initialized from a standard normal distribution, a single fixed target $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero, and the weight matrices $\boldsymbol{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_\Omega \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a) Variance of hidden unit activations computed in forward pass as a function of the network layer. For He initialization ($\sigma^2_\Omega = 2/D_h = 0.02$), the variance is stable. However, for larger values, it increases rapidly, and for smaller values, it decreases rapidly (note log scale). b) The variance of the gradients in the backward pass (solid lines) continues this trend; if we initialize with a value larger than 0.02, the magnitude of the gradients increases rapidly as we pass back through the network. If we initialize with a value smaller, then the magnitude decreases. These are known as the *exploding gradient* and *vanishing gradient* problems, respectively.

This, in turn, implies that if we want the variance $\sigma^2_{f'}$ of the subsequent pre-activations $\mathbf{f}'$ to be the same as the variance σ^2_f of the original pre-activations $\mathbf{f}$ during the forward pass, we should set:

$$\sigma^2_\Omega = \frac{2}{D_h}, \tag{7.31}$$

where D_h is the dimension of the original layer to which the weights were applied. This is known as *He initialization*.

7.5.2 Initialization for backward pass

A similar argument establishes how the variance of the gradients $\partial l/\partial f_k$ changes during the backward pass. During the backward pass, we multiply by the transpose $\boldsymbol{\Omega}^T$ of the weight matrix (equation 7.24), so the equivalent expression becomes:

$$\sigma_\Omega^2 = \frac{2}{D_{h'}}, \tag{7.32}$$

where $D_{h'}$ is the dimension of the layer that the weights feed into.

7.5.3 Initialization for both forward and backward pass

If the weight matrix $\boldsymbol{\Omega}$ is not square (i.e., there are different numbers of hidden units in the two adjacent layers, so D_h and $D_{h'}$ differ), then it is not possible to choose the variance to satisfy both equations 7.31 and 7.32 simultaneously. One possible compromise is to use the mean $(D_h + D_{h'})/2$ as a proxy for the number of terms, which gives:

$$\sigma_\Omega^2 = \frac{4}{D_h + D_{h'}}. \tag{7.33}$$

Figure 7.7 shows empirically that both the variance of the hidden units in the forward pass and the variance of the gradients in the backward pass remain stable when the parameters are initialized appropriately.

Problem 7.15

Notebook 7.3 Initialization

7.6 Example training code

The primary focus of this book is scientific; this is not a guide for implementing deep learning models. Nonetheless, in figure 7.8, we present PyTorch code that implements the ideas explored in this book so far. The code defines a neural network and initializes the weights. It creates random input and output datasets and defines a least squares loss function. The model is trained from the data using SGD with momentum in batches of size 10 over 100 epochs. The learning rate starts at 0.01 and halves every 10 epochs.

Problems 7.16–7.17

The takeaway is that although the underlying ideas in deep learning are quite complex, implementation is relatively simple. For example, all of the details of the backpropagation are hidden in the single line of code: `loss.backward()`.

7.7 Summary

The previous chapter introduced stochastic gradient descent (SGD), an iterative optimization algorithm that aims to find the minimum of a function. In the context of neural networks, this algorithm finds the parameters that minimize the loss function. SGD relies on the gradient of the loss function with respect to the parameters, which must be initialized before optimization. This chapter has addressed these two problems for deep neural networks.

The gradients must be evaluated for a very large number of parameters, for each member of the batch, and at each SGD iteration. It is hence imperative that the gradient

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
    nn.Linear(D_i, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
    if isinstance(layer_in, nn.Linear):
        nn.init.kaiming_normal_(layer_in.weight)
        layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 random data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

# loop over the dataset 100 times
for epoch in range(100):
    epoch_loss = 0.0
    # loop over batches
    for i, data in enumerate(data_loader):
        # retrieve inputs and labels for this batch
        x_batch, y_batch = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward pass
        pred = model(x_batch)
        loss = criterion(pred, y_batch)
        # backward pass
        loss.backward()
        # SGD update
        optimizer.step()
        # update statistics
        epoch_loss += loss.item()
    # print error
    print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')
    # tell scheduler to consider updating learning rate
    scheduler.step()
```

Figure 7.8 Sample code for training two-layer network on random data.

computation is efficient, and to this end, the backpropagation algorithm was introduced. Careful parameter initialization is also critical. The magnitudes of the hidden unit activations can either decrease or increase exponentially in the forward pass. The same is true of the gradient magnitudes in the backward pass, where these behaviors are known as the vanishing gradient and exploding gradient problems. Both impede training but can be avoided with appropriate initialization.

We've now defined the model and the loss function, and we can train a model for a given task. The next chapter discusses how to measure the model performance.

Notes

Backpropagation: Efficient reuse of partial computations while calculating gradients in computational graphs has been repeatedly discovered, including by Werbos (1974), Bryson et al. (1979), LeCun (1985), and Parker (1985). However, the most celebrated description of this idea was by Rumelhart et al. (1985) and Rumelhart et al. (1986), who also coined the term "backpropagation." This latter work kick-started a new phase of neural network research in the eighties and nineties; for the first time, it was practical to train networks with hidden layers. However, progress stalled due (in retrospect) to a lack of training data, limited computational power, and the use of sigmoid activations. Areas such as natural language processing and computer vision did not rely on neural network models until the remarkable image classification results of Krizhevsky et al. (2012) ushered in the modern era of deep learning.

The implementation of backpropagation in modern deep learning frameworks such as PyTorch and TensorFlow is an example of reverse-mode algorithmic differentiation. This is distinguished from forward-mode algorithmic differentiation in which the derivatives from the chain rule are accumulated while moving forward through the computational graph (see problem 7.13). Further information about algorithmic differentiation can be found in Griewank & Walther (2008) and Baydin et al. (2018).

Initialization: He initialization was first introduced by He et al. (2015). It follows closely from *Glorot* or *Xavier* initialization (Glorot & Bengio, 2010), which is very similar but does not consider the effect of the ReLU layer and so differs by a factor of two. Essentially the same method was proposed much earlier by LeCun et al. (2012) but with a slightly different motivation; in this case, sigmoidal activation functions were used, which naturally normalize the range of outputs at each layer, and hence help prevent an exponential increase in the magnitudes of the hidden units. However, if the pre-activations are too large, they fall into the flat regions of the sigmoid function and result in very small gradients. Hence, it is still important to initialize the weights sensibly. Klambauer et al. (2017) introduce the scaled exponential linear unit (SeLU) and show that, within a certain range of inputs, this activation function tends to make the activations in network layers automatically converge to mean zero and unit variance.

A completely different approach is to pass data through the network and then normalize by the empirically observed variance. *Layer-sequential unit variance initialization* (Mishkin & Matas, 2016) is an example of this kind of method, in which the weight matrices are initialized as orthonormal. GradInit (Zhu et al., 2021) randomizes the initial weights and temporarily fixes them while it learns non-negative scaling factors for each weight matrix. These factors are selected to maximize the decrease in the loss for a fixed learning rate subject to a constraint on the maximum gradient norm. *Activation normalization* or *ActNorm* adds a learnable scaling and offset parameter after each network layer at each hidden unit. They run an initial batch through the network and then choose the offset and scale so that the mean of the activations is zero and the variance one. After this, these extra parameters are learned as part of the model.

Closely related to these methods are schemes such as *BatchNorm* (Ioffe & Szegedy, 2015), in which the network normalizes the variance of each batch as part of its processing at every step. BatchNorm and its variants are discussed in chapter 11. Other initialization schemes have been proposed for specific architectures, including the *ConvolutionOrthogonal* initializer (Xiao et al., 2018a) for convolutional networks, *Fixup* (Zhang et al., 2019a) for residual networks, and *TFixup* (Huang et al., 2020a) and *DTFixup* (Xu et al., 2021b) for transformers.

Reducing memory requirements: Training neural networks is memory intensive. We must store both the model parameters and the pre-activations at the hidden units for every member of the batch during the forward pass. Two methods that decrease memory requirements are *gradient checkpointing* (Chen et al., 2016a) and *micro-batching* (Huang et al., 2019). In gradient checkpointing, the activations are only stored every N layers during the forward pass. During the backward pass, the intermediate missing activations are recalculated from the nearest checkpoint. In this manner, we can drastically reduce the memory requirements at the computational cost of performing the forward pass twice (problem 7.11). In micro-batching, the batch is subdivided into smaller parts, and the gradient updates are aggregated from each sub-batch before being applied to the network. A completely different approach is to build a reversible network (e.g., Gomez et al., 2017), in which the activations at the previous layer can be computed from the activations at the current one, so there is no need to cache anything during the forward pass (see chapter 16). Sohoni et al. (2019) review approaches to reducing memory requirements.

Distributed training: For sufficiently large models, the memory requirements or total required time may be too much for a single processor. In this case, we must use *distributed training*, in which training takes place in parallel across multiple processors. There are several approaches to parallelism. In *data parallelism*, each processor or *node* contains a full copy of the model but runs a subset of the batch (see Xing et al., 2015; Li et al., 2020b). The gradients from each node are aggregated centrally and then redistributed back to each node to ensure that the models remain consistent. This is known as *synchronous training.* The synchronization required to aggregate and redistribute the gradients can be a performance bottleneck, and this leads to the idea of asynchronous training. For example, in the *Hogwild!* algorithm (Recht et al., 2011), the gradient from a node is used to update a central model whenever it is ready. The updated model is then redistributed to the node. This means that each node may have a slightly different version of the model at any given time, so the gradient updates may be stale; however, it works well in practice. Other decentralized schemes have also been developed. For example, in Zhang et al. (2016a), the individual nodes update one another in a ring structure.

Data parallelism methods still assume that the entire model can be held in the memory of a single node. *Pipeline model parallelism* stores different layers of the network on different nodes and hence does not have this requirement. In a naïve implementation, the first node runs the forward pass for the batch on the first few layers and passes the result to the next node, which runs the forward pass on the next few layers and so on. In the backward pass, the gradients are updated in the opposite order. The obvious disadvantage of this approach is that each machine lies idle for most of the cycle. Various schemes revolving around each node processing micro-batches sequentially have been proposed to reduce this inefficiency (e.g., Huang et al., 2019; Narayanan et al., 2021a). Finally, in *tensor model parallelism*, computation at a single network layer is distributed across nodes (e.g., Shoeybi et al., 2019). A good overview of distributed training methods can be found in Narayanan et al. (2021b), who combine tensor, pipeline, and data parallelism to train a language model with one trillion parameters on 3072 GPUs.

Problems

Problem 7.1 A two-layer network with two hidden units in each layer can be defined as:

$$\begin{aligned} y &= \phi_0 + \phi_1 \mathrm{a}\Big[\psi_{01} + \psi_{11}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{21}\mathrm{a}[\theta_{02} + \theta_{12}x]\Big] \\ &\quad + \phi_2 \mathrm{a}\Big[\psi_{02} + \psi_{12}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{02} + \theta_{12}x]\Big], \end{aligned} \tag{7.34}$$

where the functions $a[\bullet]$ are ReLU functions. Compute the derivatives of the output y with respect to each of the 13 parameters $\boldsymbol{\phi}_\bullet$, $\theta_{\bullet\bullet}$, and $\psi_{\bullet\bullet}$ directly (i.e., not using the backpropagation algorithm). The derivative of the ReLU function with respect to its input $\partial a[z]/\partial z$ is the indicator function $\mathbb{I}[z > 0]$, which returns one if the argument is greater than zero and zero otherwise (figure 7.6).

Problem 7.2 Find an expression for the final term in each of the five chains of derivatives in equation 7.12.

Problem 7.3 What size are each of the terms in equation 7.19?

Problem 7.4 Calculate the derivative $\partial \ell_i / \partial f[\mathbf{x}_i, \boldsymbol{\phi}]$ for the least squares loss function:

$$\ell_i = (y_i - \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2. \tag{7.35}$$

Problem 7.5 Calculate the derivative $\partial \ell_i / \partial \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for the binary classification loss function:

$$\ell_i = -(1 - y_i)\log\Big[1 - \mathrm{sig}\big[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]\big]\Big] - y_i \log\Big[\mathrm{sig}\big[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]\big]\Big], \tag{7.36}$$

where the function $\mathrm{sig}[\bullet]$ is the logistic sigmoid and is defined as:

$$\mathrm{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{7.37}$$

Problem 7.6* Show that for $\mathbf{z} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{h}} = \boldsymbol{\Omega}^T,$$

where $\partial \mathbf{z}/\partial \mathbf{h}$ is a matrix containing the term $\partial z_i / \partial h_j$ in its i^{th} column and j^{th} row. To do this, first find an expression for the constituent elements $\partial z_i / \partial h_j$, and then consider the form that the matrix $\partial \mathbf{z}/\partial \mathbf{h}$ must take.

Problem 7.7 Consider the case where we use the logistic sigmoid (see equation 7.37) as an activation function, so $h = \mathrm{sig}[f]$. Compute the derivative $\partial h / \partial f$ for this activation function. What happens to the derivative when the input takes (i) a large positive value and (ii) a large negative value?

Problem 7.8 Consider using (i) the Heaviside function and (ii) the rectangular function as activation functions:

$$\mathrm{Heaviside}[z] = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}, \tag{7.38}$$

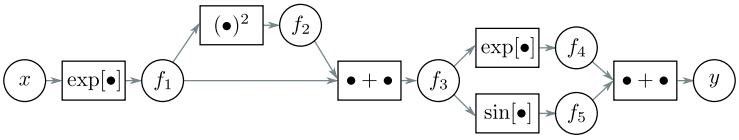

Figure 7.9 Computational graph for problem 7.12 and problem 7.13. Adapted from Domke (2010).

and

$$\text{rect}[z] = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}. \tag{7.39}$$

Discuss why these functions are problematic for neural network training with gradient-based optimization methods.

Problem 7.9* Consider a loss function $\ell[\mathbf{f}]$, where $\mathbf{f} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h}$. We want to find how the loss ℓ changes when we change $\boldsymbol{\Omega}$, which we'll express with a matrix that contains the derivative $\partial\ell/\partial\Omega_{ij}$ at the i^{th} row and j^{th} column. Find an expression for $\partial f_i/\partial\Omega_{ij}$ and, using the chain rule, show that:

$$\frac{\partial \ell}{\partial \boldsymbol{\Omega}} = \frac{\partial \ell}{\partial \mathbf{f}}\mathbf{h}^T. \tag{7.40}$$

Problem 7.10* Derive the equations for the backward pass of the backpropagation algorithm for a network that uses leaky ReLU activations, which are defined as:

$$\text{a}[z] = \text{ReLU}[z] = \begin{cases} \alpha \cdot z & z < 0 \\ z & z \geq 0 \end{cases}, \tag{7.41}$$

where α is a small positive constant (typically 0.1).

Problem 7.11 Consider training a network with fifty layers, where we only have enough memory to store the pre-activations at every tenth hidden layer during the forward pass. Explain how to compute the derivatives in this situation using gradient checkpointing.

Problem 7.12* This problem explores computing derivatives on general acyclic computational graphs. Consider the function:

$$y = \exp\left[\exp[x] + \exp[x]^2\right] + \sin[\exp[x] + \exp[x]^2]. \tag{7.42}$$

We can break this down into a series of intermediate computations so that:

$$\begin{aligned} f_1 &= \exp[x] \\ f_2 &= f_1^2 \\ f_3 &= f_1 + f_2 \\ f_4 &= \exp[f_3] \\ f_5 &= \sin[f_3] \\ y &= f_4 + f_5. \end{aligned} \tag{7.43}$$

The associated computational graph is depicted in figure 7.9. Compute the derivative $\partial y/\partial x$ by *reverse-mode differentiation*. In other words, compute in order:

$$\frac{\partial y}{\partial f_5}, \frac{\partial y}{\partial f_4}, \frac{\partial y}{\partial f_3}, \frac{\partial y}{\partial f_2}, \frac{\partial y}{\partial f_1} \text{ and } \frac{\partial y}{\partial x}, \tag{7.44}$$

using the chain rule in each case to make use of the derivatives already computed.

Problem 7.13* For the same function as in problem 7.12, compute the derivative $\partial y/\partial x$ by *forward-mode differentiation*. In other words, compute in order:

$$\frac{\partial f_1}{\partial x}, \frac{\partial f_2}{\partial x}, \frac{\partial f_3}{\partial x}, \frac{\partial f_4}{\partial x}, \frac{\partial f_5}{\partial x}, \text{ and } \frac{\partial y}{\partial x}, \tag{7.45}$$

using the chain rule in each case to make use of the derivatives already computed. Why do we not use forward-mode differentiation when we calculate the parameter gradients for deep networks?

Problem 7.14 Consider a random variable a with variance $\text{Var}[a] = \sigma^2$ and a symmetrical distribution around the mean $\mathbb{E}[a] = 0$. Prove that if we pass this variable through the ReLU function:

$$b = \text{ReLU}[a] = \begin{cases} 0 & a < 0 \\ a & a \geq 0 \end{cases}, \tag{7.46}$$

then the second moment of the transformed variable is $\mathbb{E}[b^2] = \sigma^2/2$.

Problem 7.15 What would you expect to happen if we initialized all of the weights and biases in the network to zero?

Problem 7.16 Implement the code in figure 7.8 in PyTorch and plot the training loss as a function of the number of epochs.

Problem 7.17 Change the code in figure 7.8 to tackle a binary classification problem. You will need to (i) change the targets y so they are binary, (ii) change the network to predict numbers between zero and one (iii) change the loss function appropriately.