Chapter 6

Fitting models

Chapters 3 and 4 described shallow and deep neural networks. These represent families of piecewise linear functions, where the parameters determine the particular function. Chapter 5 introduced the loss — a single number representing the mismatch between the network predictions and the ground truth for a training set.

The loss depends on the network parameters, and this chapter considers how to find the parameter values that minimize this loss. This is known as *learning* the network's parameters or simply as *training* or *fitting* the model. The process is to choose initial parameter values and then iterate the following two steps: (i) compute the derivatives (gradients) of the loss with respect to the parameters, and (ii) adjust the parameters based on the gradients to decrease the loss. After many iterations, we hope to reach the overall minimum of the loss function.

This chapter tackles the second of these steps; we consider algorithms that adjust the parameters to decrease the loss. Chapter 7 discusses how to initialize the parameters and compute the gradients for neural networks.

6.1 Gradient descent

To fit a model, we need a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek parameters $\boldsymbol{\phi}$ for the model $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that map the inputs $\mathbf{x}_i$ to the outputs $\mathbf{y}_i$ as well as possible. To this end, we define a loss function $L[\boldsymbol{\phi}]$ that returns a single number that quantifies the mismatch in this mapping. The goal of an *optimization algorithm* is to find parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L[\boldsymbol{\phi}]\Big]. \tag{6.1}$$

There are many families of optimization algorithms, but the standard methods for training neural networks are iterative. These algorithms initialize the parameters heuristically and then adjust them repeatedly in such a way that the loss decreases.

The simplest method in this class is *gradient descent*. This starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates two steps:

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{6.2}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{6.3}$$

where the positive scalar α determines the magnitude of the change.

The first step computes the gradient of the loss function at the current position. This determines the *uphill* direction of the loss function. The second step moves a small distance α *downhill* (hence the negative sign). The parameter α may be fixed (in which case, we call it a *learning rate*), or we may perform a *line search* where we try several values of α to find the one that most decreases the loss.

Notebook 6.1
Line search

At the minimum of the loss function, the surface must be flat (or we could improve further by going downhill). Hence, the gradient will be zero, and the parameters will stop changing. In practice, we monitor the gradient magnitude and terminate the algorithm when it becomes too small.

6.1.1 Linear regression example

Consider applying gradient descent to the 1D linear regression model from chapter 2. The model $\mathrm{f}[x, \boldsymbol{\phi}]$ maps a scalar input x to a scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, which represent the y-intercept and the slope:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{6.4}$$

Given a dataset $\{x_i, y_i\}$ containing I input/output pairs, we choose the least squares loss function:

$$\begin{aligned} L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2, \end{aligned} \tag{6.5}$$

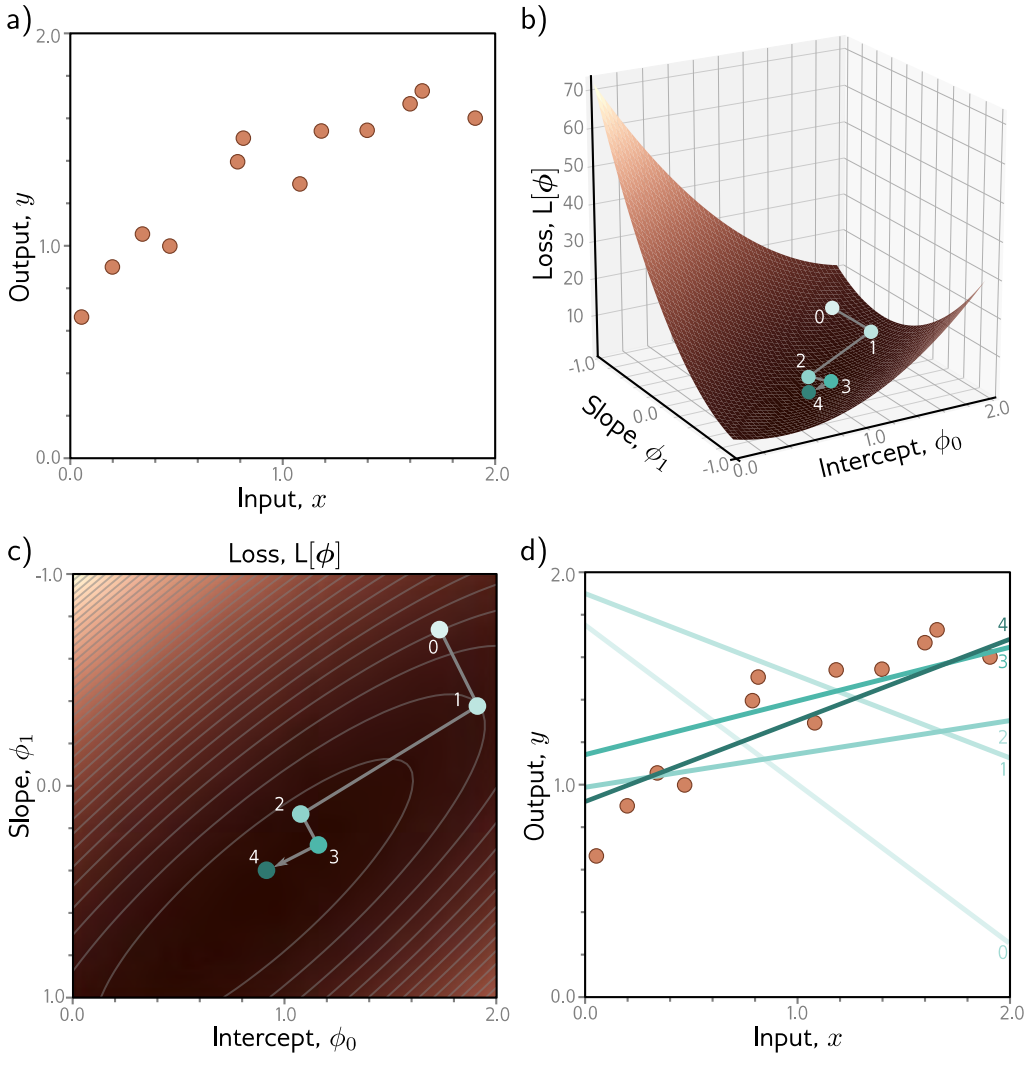

Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

where the term $\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2$ is the individual contribution to the loss from the i^{th} training example.

The derivative of the loss function with respect to the parameters can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} \frac{\partial \ell_i}{\partial \boldsymbol{\phi}}, \tag{6.6}$$

Problem 6.1

where these are given by:

$$\frac{\partial \ell_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial \ell_i}{\partial \phi_0} \\ \frac{\partial \ell_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}. \tag{6.7}$$

Notebook 6.2 Gradient descent

Figure 6.1 shows the progression of this algorithm as we iteratively compute the derivatives according to equations 6.6 and 6.7 and then update the parameters using the rule in equation 6.3. In this case, we have used a line search procedure to find the value of α that decreases the loss the most at each iteration.

6.1.2 Gabor model example

Problem 6.2

Loss functions for linear regression problems (figure 6.1c) always have a single well-defined global minimum. More formally, they are *convex*, which means that every chord (line segment between two points on the surface) lies above the function and does not intersect it. Convexity implies that wherever we initialize the parameters, we are bound to reach the minimum if we keep walking downhill; the training procedure can't fail.

Unfortunately, loss functions for most nonlinear models, including both shallow and deep networks, are *non-convex*. Visualizing neural network loss functions is challenging due to the number of parameters. Hence, we first explore a simpler nonlinear model with two parameters to gain insight into the properties of non-convex loss functions:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{32.0}\right). \tag{6.8}$$

Problems 6.3–6.5

This *Gabor model* maps scalar input x to scalar output y and consists of a sinusoidal component (creating an oscillatory function) multiplied by a negative exponential component (causing the amplitude to decrease as we move from the center). It has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where $\phi_0 \in \mathbb{R}$ determines the mean position of the function and $\phi_1 \in \mathbb{R}^+$ stretches or squeezes it along the x-axis (figure 6.2).

Consider a training set of I examples $\{x_i, y_i\}$ (figure 6.3). The least squares loss function for I training examples is defined as:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2. \tag{6.9}$$

Once more, the goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

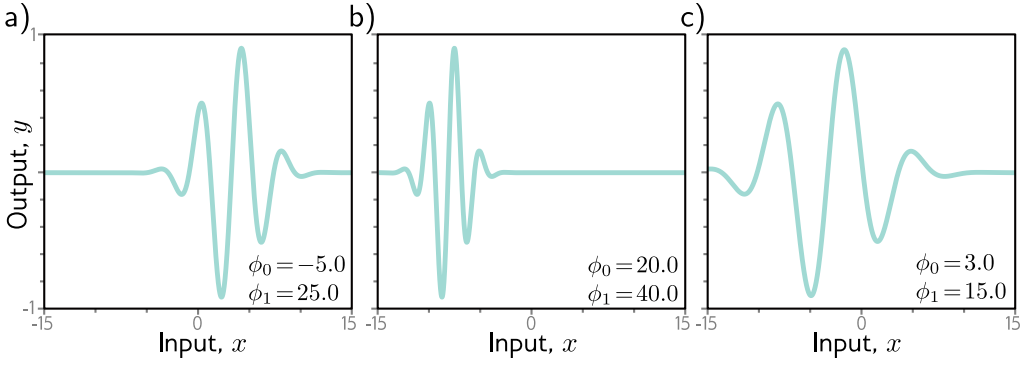

Figure 6.2 Gabor model. This nonlinear model maps scalar input x to scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$. It describes a sinusoidal function that decreases in amplitude with distance from its center. Parameter $\phi_0 \in \mathbb{R}$ determines the position of the center. As ϕ_0 increases, the function moves left. Parameter $\phi_1 \in \mathbb{R}^+$ squeezes the function along the x-axis relative to the center. As ϕ_1 increases, the function narrows. a–c) Model with different parameters.

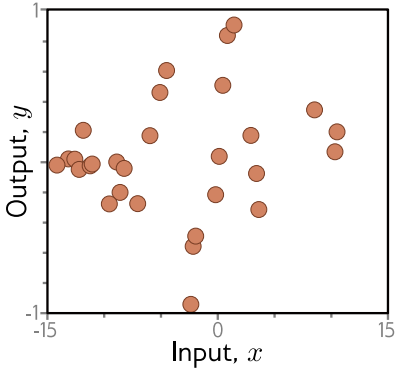

Figure 6.3 Training data for fitting the Gabor model. The training dataset contains 28 input/output examples $\{x_i, y_i\}$. These data were created by uniformly sampling $x_i \in [-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding normally distributed noise.

6.1.3 Local minima and saddle points

Figure 6.4 depicts the loss function associated with the Gabor model for this dataset. There are numerous *local minima* (cyan circles). Here the gradient is zero, and the loss increases if we move in any direction, but we are *not* at the overall minimum of the function. The point with the lowest loss is known as the *global minimum* and is depicted by the gray circle.

Problem 6.6

If we start in a random position and use gradient descent to go downhill, there is no guarantee that we will wind up at the global minimum and find the best parameters (figure 6.5a). It's equally or even more likely that the algorithm will terminate in one of the local minima. Furthermore, there is no way of knowing whether there is a better solution elsewhere.

Problems 6.7–6.8

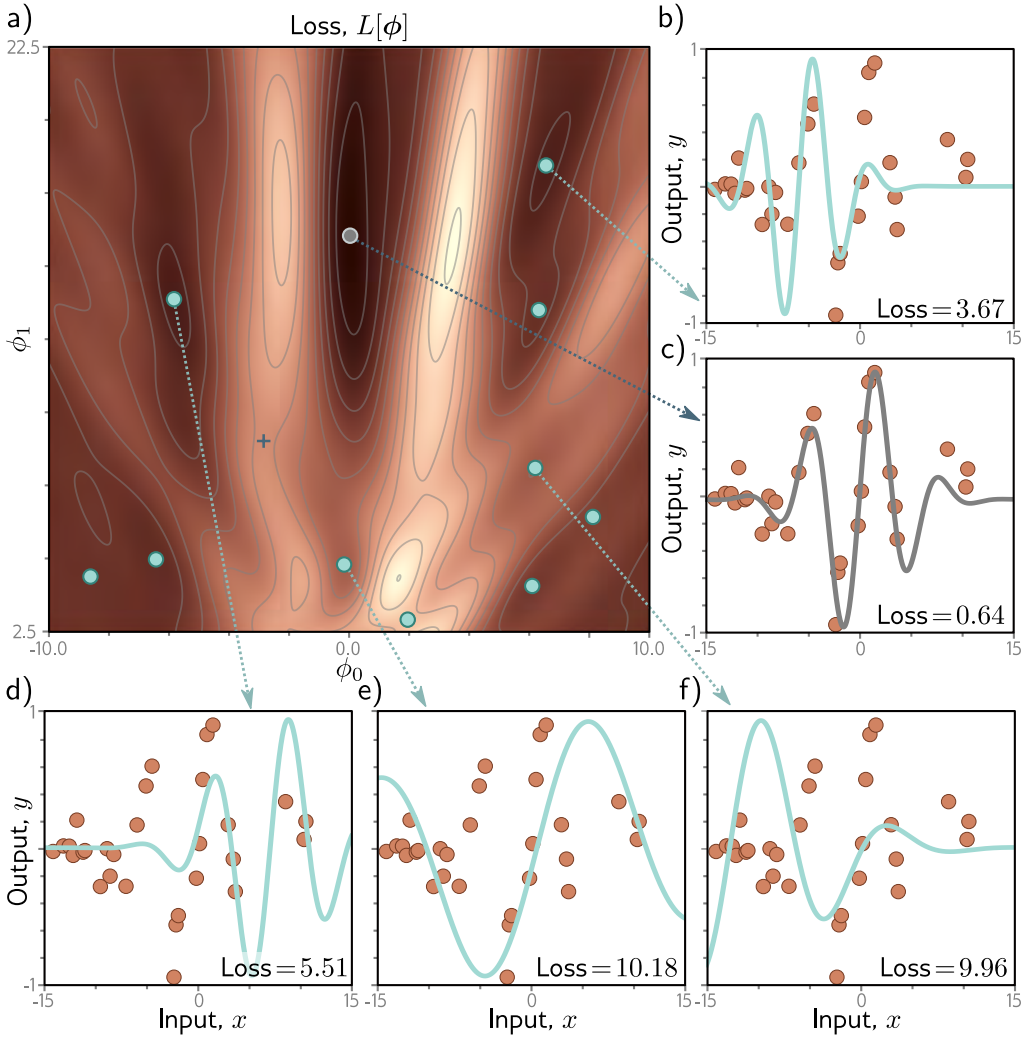

Figure 6.4 Loss function for the Gabor model. a) The loss function is non-convex, with multiple local minima (cyan circles) in addition to the global minimum (gray circle). It also contains saddle points where the gradient is locally zero, but the function increases in one direction and decreases in the other. The blue cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically. b–f) Models associated with the different minima. In each case, there is no small change that decreases the loss. Panel (c) shows the global minimum, which has a loss of 0.64.

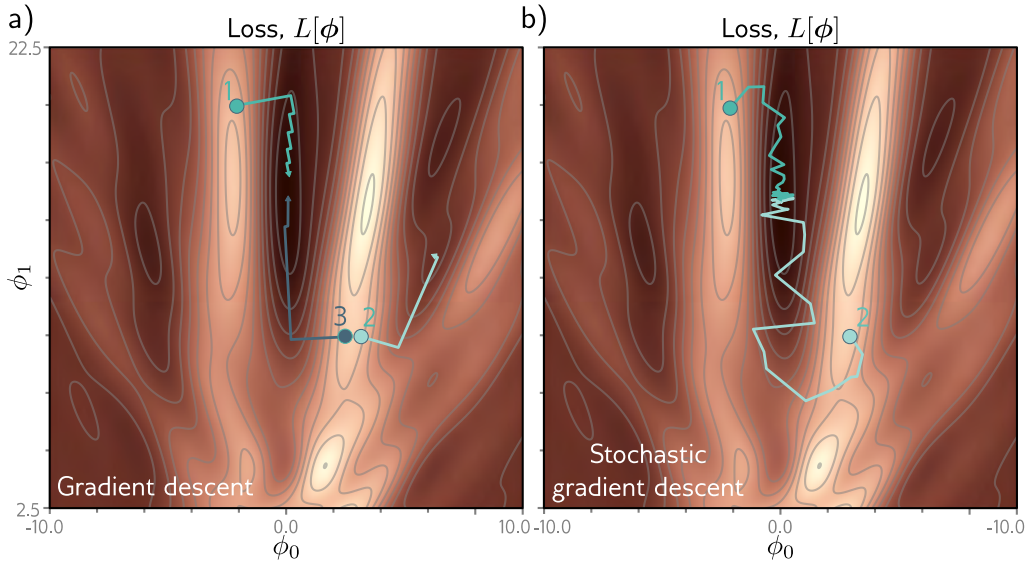

Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent with line search. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized outside this valley (e.g., point 2), it will descend toward one of the local minima. b) Stochastic gradient descent adds noise to the optimization process, so it is possible to escape from the wrong valley (e.g., point 2) and still reach the global minimum.

In addition, the loss function contains *saddle points* (e.g., the blue cross in figure 6.4). Here, the gradient is zero, but the function increases in some directions and decreases in others. If the current parameters are not exactly at the saddle point, then gradient descent can escape by moving downhill. However, the surface near the saddle point is flat, so it's hard to be sure that training hasn't converged; if we terminate the algorithm when the gradient is small, we may erroneously stop near a saddle point.

6.2 Stochastic gradient descent

The Gabor model has two parameters, so we could find the global minimum by either (i) exhaustively searching the parameter space or (ii) repeatedly starting gradient descent from different positions and choosing the result with the lowest loss. However, neural network models can have millions of parameters, so neither approach is practical. In short, using gradient descent to find the global optimum of a high-dimensional loss function is challenging. We can find *a* minimum, but there is no way to tell whether this

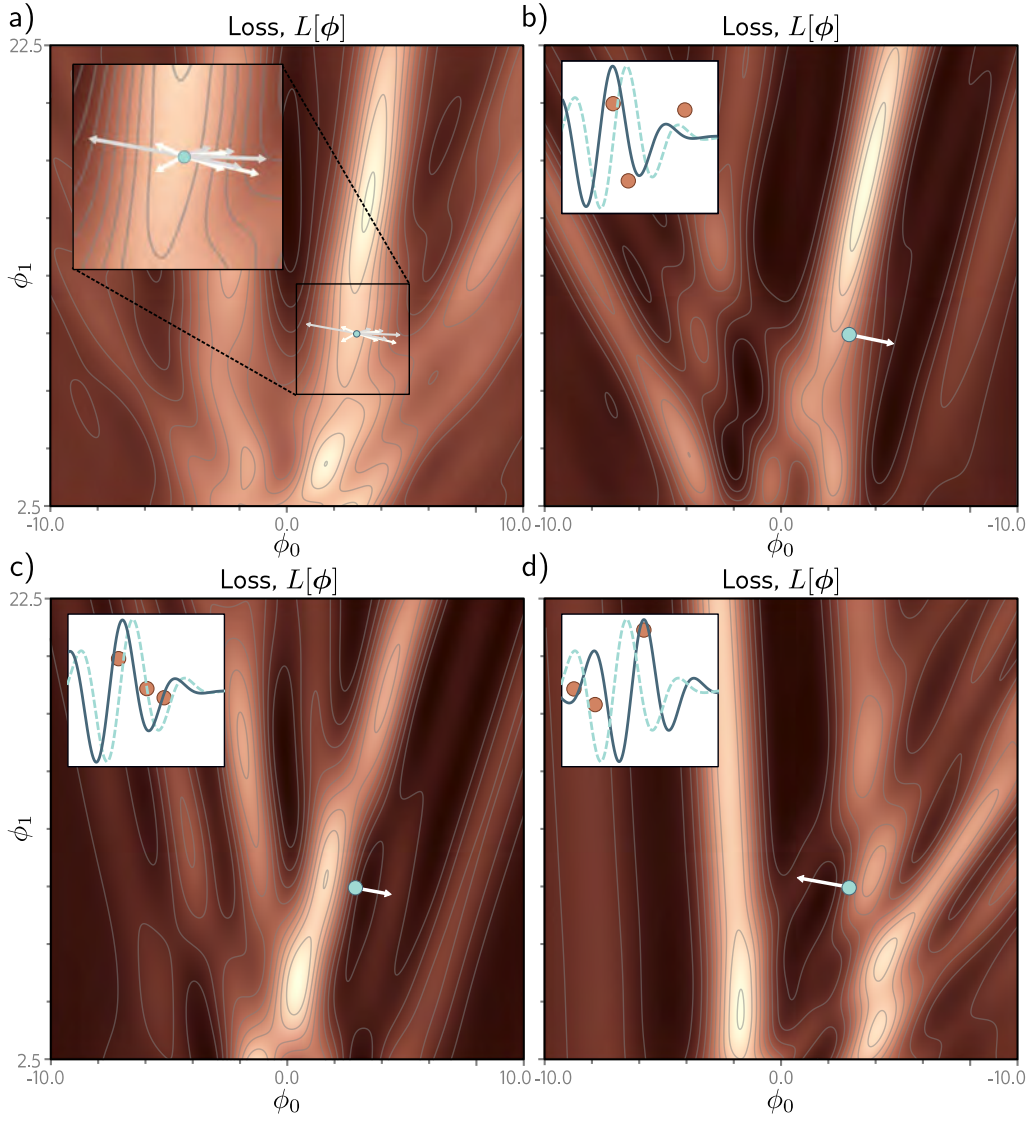

Figure 6.6 Alternative view of SGD for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. b) Loss function for one possible batch. The SGD algorithm moves in the downhill direction on this function for a distance that is determined by the learning rate and the local gradient magnitude. The current model (dashed function in inset) changes to better fit the batch data (solid function). c) A different batch creates a different loss function and results in a different update. d) For this batch, the algorithm moves *downhill* with respect to the batch loss function but *uphill* with respect to the global loss function in panel (a). This is how SGD can escape local minima.

is the global minimum or even a good one.

One of the main problems is that the final destination of a gradient descent algorithm is entirely determined by the starting point. *Stochastic gradient descent (SGD)* attempts to remedy this problem by adding some noise to the gradient at each step. The solution still moves downhill on average, but at any given iteration, the direction chosen is not necessarily in the steepest downhill direction. Indeed, it might not be downhill at all. The SGD algorithm has the possibility of moving temporarily uphill and hence jumping from one "valley" of the loss function to another (figure 6.5b).

Notebook 6.3 Stochastic gradient descent

6.2.1 Batches and epochs

The mechanism for introducing randomness is simple. At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone. This subset is known as a *minibatch* or *batch* for short. The update rule for the model parameters $\boldsymbol{\phi}_t$ at iteration t is hence:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{6.10}$$

where $\mathcal{B}_t$ is a set containing the indices of the input/output pairs in the current batch and, as before, ℓ_i is the loss due to the i^{th} pair. The term α is the learning rate, and together with the gradient magnitude, determines the distance moved at each iteration. The learning rate is chosen at the start of the procedure and does not depend on the local properties of the function.

The batches are usually drawn from the dataset without replacement. The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again. A single pass through the entire training dataset is referred to as an *epoch*. A batch may be as small as a single example or as large as the whole dataset. The latter case is called *full-batch gradient descent* and is identical to regular (non-stochastic) gradient descent.

Problem 6.9

An alternative interpretation of SGD is that it computes the gradient of a different loss function at each iteration; the loss function depends on both the model and the training data and hence will differ for each randomly selected batch. In this view, SGD performs deterministic gradient descent on a constantly changing loss function (figure 6.6). However, despite this variability, the expected loss and expected gradients at any point remain the same as for gradient descent.

6.2.2 Properties of stochastic gradient descent

SGD has several attractive features. First, although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration. Hence, the updates tend to be sensible even if they are not optimal. Second, because it draws training examples without replacement and iterates through the dataset, the training examples all still contribute equally. Third, it is less computationally expensive to compute the gradient

from just a subset of the training data. Fourth, it can (in principle) escape local minima. Fifth, it reduces the chances of getting stuck near saddle points; it is likely that at least some of the possible batches will have a significant gradient at any point on the loss function. Finally, there is some evidence that SGD finds parameters for neural networks that cause them to generalize well to new data in practice (see section 9.2).

SGD does not necessarily "converge" in the traditional sense. However, the hope is that when we are close to the global minimum, all the data points will be well described by the model. Consequently, the gradient will be small, whichever batch is chosen, and the parameters will cease to change much. In practice, SGD is often applied with a *learning rate schedule*. The learning rate α starts at a high value and is decreased by a constant factor every N epochs. The logic is that in the early stages of training, we want the algorithm to explore the parameter space, jumping from valley to valley to find a sensible region. In later stages, we are roughly in the right place and are more concerned with fine-tuning the parameters, so we decrease α to make smaller changes.

6.3 Momentum

A common modification to stochastic gradient descent is to add a *momentum* term. We update the parameters with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned} \tag{6.11}$$

where $\mathbf{m}_t$ is the momentum (which drives the update at iteration t), $\beta \in [0, 1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.

The recursive formulation of the momentum calculation means that the gradient step is an infinite weighted sum of all the previous gradients, where the weights get smaller as we move back in time. The effective learning rate increases if all these gradients are aligned over multiple iterations but decreases if the gradient direction repeatedly changes as the terms in the sum cancel out. The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys (figure 6.7).

Problem 6.10

6.3.1 Nesterov accelerated momentum

Notebook 6.4 Momentum

The momentum term can be considered a coarse prediction of where the SGD algorithm will move next. Nesterov accelerated momentum (figure 6.8) computes the gradients at this predicted point rather than at the current point:

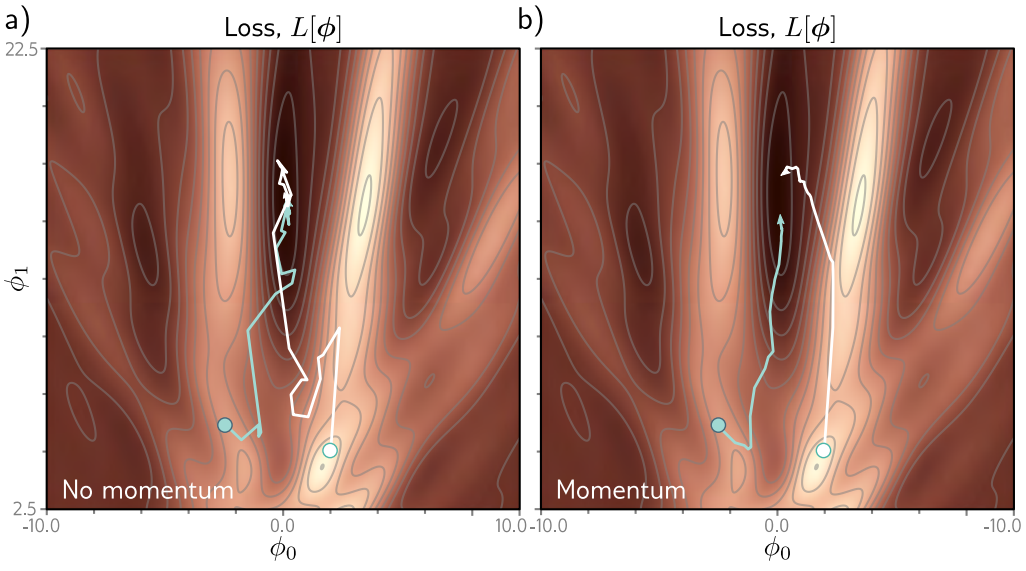

Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

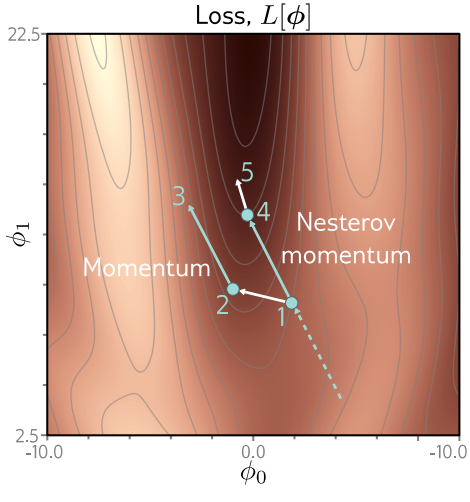

Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t - \alpha\beta\cdot\mathbf{m}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha\cdot\mathbf{m}_{t+1},\end{aligned} \tag{6.12}$$

where now the gradients are evaluated at $\boldsymbol{\phi}_t - \alpha\beta\cdot\mathbf{m}_t$. One way to think about this is that the gradient term now corrects the path provided by momentum alone.

6.4 Adam

Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further). When the gradient of the loss surface is much steeper in one direction than another, it is difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable (figures 6.9a–b).

A straightforward approach is to normalize the gradients so that we move a fixed distance (governed by the learning rate) in each direction. To do this, we first measure the gradient $\mathbf{m}_{t+1}$ and the pointwise squared gradient $\mathbf{v}_{t+1}$:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2.\end{aligned} \tag{6.13}$$

Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha\cdot\frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}}+\epsilon}, \tag{6.14}$$

where the square root and division are both pointwise, α is the learning rate, and ϵ is a small constant that prevents division by zero when the gradient magnitude is zero. The term $\mathbf{v}_{t+1}$ is the squared gradient, and the positive root of this is used to normalize the gradient itself, so all that remains is the sign in each coordinate direction. The result is that the algorithm moves a fixed distance α along each coordinate, where the direction is determined by whichever way is downhill (figure 6.9c). This simple algorithm makes good progress in both directions but will not converge unless it happens to land exactly at the minimum. Instead, it will bounce back and forth around the minimum.

Adaptive moment estimation, or *Adam*, takes this idea and adds momentum to both the estimate of the gradient and the squared gradient:

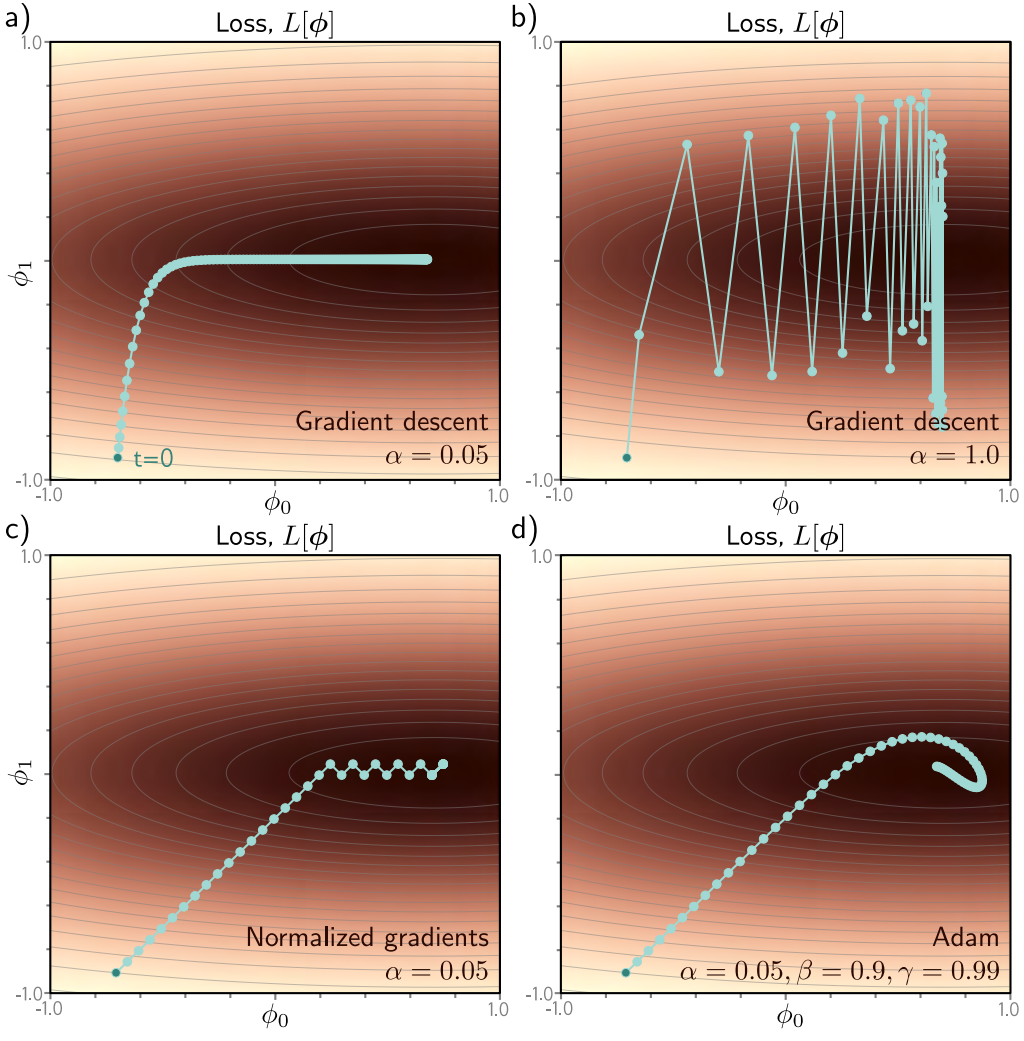

Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned} \tag{6.15}$$

where β and γ are the momentum coefficients for the two statistics.

Using momentum is equivalent to taking a weighted average over the history of each of these statistics. At the start of the procedure, all the previous measurements are effectively zero, resulting in unrealistically small estimates. Consequently, we modify these statistics using the rule:

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned} \tag{6.16}$$

Since β and γ are in the range $[0, 1)$, the terms with exponents $t+1$ become smaller with each time step, the denominators become closer to one, and this modification has a diminishing effect.

Finally, we update the parameters as before, but with the modified terms:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}. \tag{6.17}$$

Notebook 6.5 Adam

The result is an algorithm that can converge to the overall minimum and makes good progress in every direction in the parameter space. Note that Adam is usually used in a stochastic setting where the gradients and their squares are computed from mini-batches:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned} \tag{6.18}$$

and so the trajectory is noisy in practice.

As we shall see in chapter 7, the gradient magnitudes of neural network parameters can depend on their depth in the network. Adam helps compensate for this tendency and balances out changes across the different layers. In practice, Adam also has the advantage of being less sensitive to the initial learning rate because it avoids situations like those in figures 6.9a–b, so it doesn't need complex learning rate schedules.

6.5 Training algorithm hyperparameters

The choices of learning algorithm, batch size, learning rate schedule, and momentum coefficients are all considered *hyperparameters* of the training algorithm; these directly affect the final model performance but are distinct from the model parameters. Choosing these can be more art than science, and it's common to train many models with different hyperparameters and choose the best one. This is known as *hyperparameter search*. We return to this issue in chapter 8.

6.6 Summary

This chapter discussed model training. This problem was framed as finding parameters $\boldsymbol{\phi}$ that corresponded to the minimum of a loss function $L[\boldsymbol{\phi}]$. The gradient descent method measures the gradient of the loss function for the current parameters (i.e., how the loss changes when we make a small change to the parameters). Then it moves the parameters in the direction that decreases the loss fastest. This is repeated until convergence.

For nonlinear functions, the loss function may have both local minima (where gradient descent gets trapped) and saddle points (where gradient descent may appear to have converged but has not). Stochastic gradient descent helps mitigate these problems.[1] At each iteration, we use a different random subset of the data (a batch) to compute the gradient. This adds noise to the process and helps prevent the algorithm from getting trapped in a sub-optimal region of parameter space. Each iteration is also computationally cheaper since it only uses a subset of the data. We saw that adding a momentum term makes convergence more efficient. Finally, we introduced the Adam algorithm.

The ideas in this chapter apply to optimizing *any* model. The next chapter tackles two aspects of training specific to neural networks. First, we address how to compute the gradients of the loss with respect to the parameters of a neural network. This is accomplished using the famous backpropagation algorithm. Second, we discuss how to initialize the network parameters before optimization begins. Without careful initialization, the gradients used by the optimization can become extremely large or extremely small, which can hinder the training process.

Notes

Optimization algorithms: Optimization algorithms are used extensively throughout engineering, and it is generally more typical to use the term *objective function* rather than loss function or cost function. Gradient descent was invented by Cauchy (1847), and stochastic gradient descent dates back to at least Robbins & Monro (1951). A modern compromise between the two is stochastic variance-reduced descent (Johnson & Zhang, 2013), in which the full gradient is computed periodically, with stochastic updates interspersed. Reviews of optimization algorithms for neural networks can be found in Ruder (2016), Bottou et al. (2018), and Sun (2020). Bottou (2012) discusses best practice for SGD, including shuffling without replacement.

[1] Chapter 20 discusses the extent to which saddle points and local minima really *are* problems in deep learning. In practice, deep networks are surprisingly easy to train.

Convexity, minima, and saddle points: A function is convex if every chord (line segment between two points on the surface) lies above the function and does not intersect it. This can be tested algebraically by considering the *Hessian matrix* (the matrix of second derivatives):

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_N} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} & \cdots & \frac{\partial^2 L}{\partial \phi_1 \partial \phi_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial \phi_N \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_N \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_N^2} \end{bmatrix}. \tag{6.19}$$

Appendix B.3.7 Eigenvalues

If the Hessian matrix is positive definite (has positive eigenvalues) for all possible parameter values, then the function is convex; the loss function will look like a smooth bowl (as in figure 6.1c), so training will be relatively easy. There will be a single global minimum and no local minima or saddle points.

For any loss function, the eigenvalues of the Hessian matrix at places where the gradient is zero allow us to classify this position as (i) a minimum (the eigenvalues are all positive), (ii) a maximum (the eigenvalues are all negative), or (iii) a saddle point (positive eigenvalues are associated with directions in which we are at a minimum and negative ones with directions where we are at a maximum).

Line search: Gradient descent with a fixed step size is inefficient because the distance moved depends entirely on the magnitude of the gradient. It moves a long distance when the function is changing fast (where perhaps it should be more cautious) but a short distance when the function is changing slowly (where perhaps it should explore further). For this reason, gradient descent methods are usually combined with a line search procedure in which we sample the function along the desired direction to try to find the optimal step size. One such approach is bracketing (figure 6.10). Another problem with gradient descent is that it tends to lead to inefficient oscillatory behavior when descending valleys (e.g., path 1 in figure 6.5a).

Beyond gradient descent: Numerous algorithms have been developed that remedy the problems of gradient descent. Most notable is the Newton method, which takes the curvature of the surface into account using the inverse of the Hessian matrix; if the gradient of the function is changing quickly, then it applies a more cautious update. This method eliminates the need for line search and does not suffer from oscillatory behavior. However, it has its own problems; in its simplest form, it moves toward the nearest extremum, but this may be a maximum if we are closer to the top of a hill than we are to the bottom of a valley. Moreover, computing the inverse Hessian is intractable when the number of parameters is large, as in neural networks.

Problem 6.11

Properties of SGD: The limit of SGD as the learning rate tends to zero is a stochastic differential equation. Jastrzębski et al. (2018) showed that this equation relies on the learning-rate to batch size ratio and that there is a relation between the learning rate to batch size ratio and the width of the minimum found. Wider minima are considered more desirable; if the loss function for test data is similar, then small errors in the parameter estimates will have little effect on test performance. He et al. (2019) prove a generalization bound for SGD that has a positive correlation with the ratio of batch size to learning rate. They train a large number of models on different architectures and datasets and find empirical evidence that test accuracy improves when the ratio of batch size to learning rate is low. Smith et al. (2018) and Goyal et al. (2018) also identified the ratio of batch size to learning rate as being important for generalization (see figure 20.10).

Momentum: The idea of using momentum to speed up optimization dates to Polyak (1964). Goh (2017) presents an in-depth discussion of the properties of momentum. The Nesterov

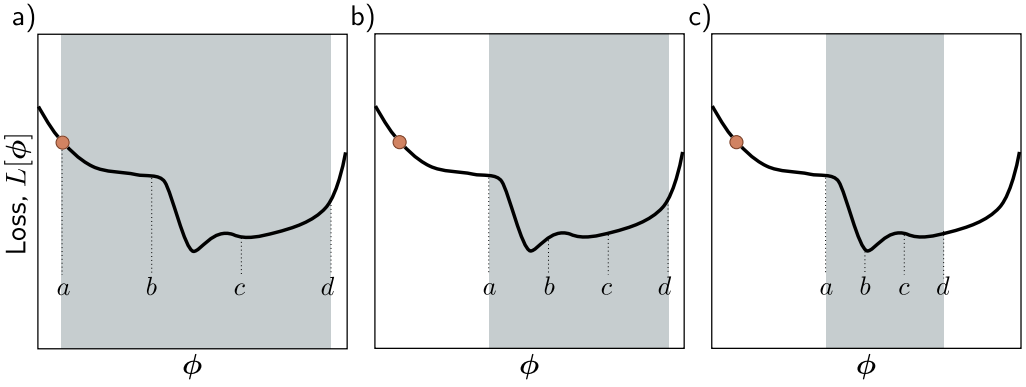

Figure 6.10 Line search using the bracketing approach. a) The current solution is at position a (orange point), and we wish to search the region $[a, d]$ (gray shaded area). We define two points b, c interior to the search region and evaluate the loss function at these points. Here $L[b] > L[c]$, so we eliminate the range $[a, b]$. b) We now repeat this procedure in the refined search region and find that $L[b] < L[c]$, so we eliminate the range $[c, d]$. c) We repeat this process until this minimum is closely bracketed.

accelerated gradient method was introduced by Nesterov (1983). Nesterov momentum was first applied in the context of stochastic gradient descent by Sutskever et al. (2013).

Adaptive training algorithms: AdaGrad (Duchi et al., 2011) is an optimization algorithm that addresses the possibility that some parameters may have to move further than others by assigning a different learning rate to each parameter. AdaGrad uses the cumulative squared gradient for each parameter to attenuate its learning rate. This has the disadvantage that the learning rates decrease over time, and learning can halt before the minimum is found. RMSProp (Hinton et al., 2012a) and AdaDelta (Zeiler, 2012) modified this algorithm to help prevent these problems by recursively updating the squared gradient term.

By far the most widely used adaptive training algorithm is adaptive moment optimization or Adam (Kingma & Ba, 2015). This combines the ideas of momentum (in which the gradient vector is averaged over time) and AdaGrad, AdaDelta, and RMSProp (in which a smoothed squared gradient term is used to modify the learning rate for each parameter). The original paper on the Adam algorithm provided a convergence proof for convex loss functions, but a counterexample was identified by Reddi et al. (2018), who developed a modification of Adam called AMSGrad, which does converge. Of course, in deep learning, the loss functions are non-convex, and Zaheer et al. (2018) subsequently developed an adaptive algorithm called YOGI and proved that it converges in this scenario. Regardless of these theoretical objections, the original Adam algorithm works well in practice and is widely used, not least because it works well over a broad range of hyperparameters and makes rapid initial progress.

One potential problem with adaptive training algorithms is that the learning rates are based on accumulated statistics of the observed gradients. At the start of training, when there are few samples, these statistics may be very noisy. This can be remedied by *learning rate warm-up* (Goyal et al., 2018), in which the learning rates are gradually increased over the first few thousand iterations. An alternative solution is rectified Adam (Liu et al., 2021a), which gradually

changes the momentum term over time in a way that helps avoid high variance. Dozat (2016) incorporated Nesterov momentum into the Adam algorithm.

SGD vs. Adam: There has been a lively discussion about the relative merits of SGD and Adam. Wilson et al. (2017) provided evidence that SGD with momentum can find lower minima than Adam, which generalizes better over a variety of deep learning tasks. However, this is strange since SGD is a special case of Adam (when $\beta = \gamma = 0$) once the modification term (equation 6.16) becomes one, which happens quickly. It is hence more likely that SGD outperforms Adam *when we use Adam's default hyperparameters*. Loshchilov & Hutter (2019) proposed AdamW, which substantially improves the performance of Adam in the presence of L2 regularization (see section 9.1). Choi et al. (2019) provide evidence that if we search for the best Adam hyperparameters, it performs just as well as SGD and converges faster. Keskar & Socher (2017) proposed a method called SWATS that starts using Adam (to make rapid initial progress) and then switches to SGD (to get better final generalization performance).

Exhaustive search: All the algorithms discussed in this chapter are iterative. A completely different approach is to quantize the network parameters and exhaustively search the resulting discretized parameter space using SAT solvers (Mézard & Mora, 2009). This approach has the potential to find the global minimum and provide a guarantee that there is no lower loss elsewhere but is only practical for very small models.

Problems

Problem 6.1 Show that the derivatives of the least squares loss function in equation 6.5 are given by the expressions in equation 6.7.

Problem 6.2 A surface is convex if the eigenvalues of the Hessian $\mathbf{H}[\boldsymbol{\phi}]$ are positive everywhere. In this case, the surface has a unique minimum, and optimization is easy. Find an algebraic expression for the Hessian matrix,

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} \end{bmatrix}, \tag{6.20}$$

Appendix B.3.7 Eigenvalues

Appendix B.3.8 Trace

Appendix B.3.8 Determinant

for the linear regression model (equation 6.5). Prove that this function is convex by showing that the eigenvalues are always positive. This can be done by showing that both the trace and the determinant of the matrix are positive.

Problem 6.3 Compute the derivatives of the least squares loss $L[\boldsymbol{\phi}]$ with respect to the parameters ϕ_0 and ϕ_1 for the Gabor model (equation 6.8).

Problem 6.4* The logistic regression model uses a linear function to assign an input $\mathbf{x}$ to one of two classes $y \in \{0, 1\}$. For a 1D input and a 1D output, it has two parameters, ϕ_0 and ϕ_1, and is defined by:

$$Pr(y = 1|x) = \text{sig}[\phi_0 + \phi_1 x], \tag{6.21}$$

where $\text{sig}[\bullet]$ is the logistic sigmoid function:

$$\text{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{6.22}$$

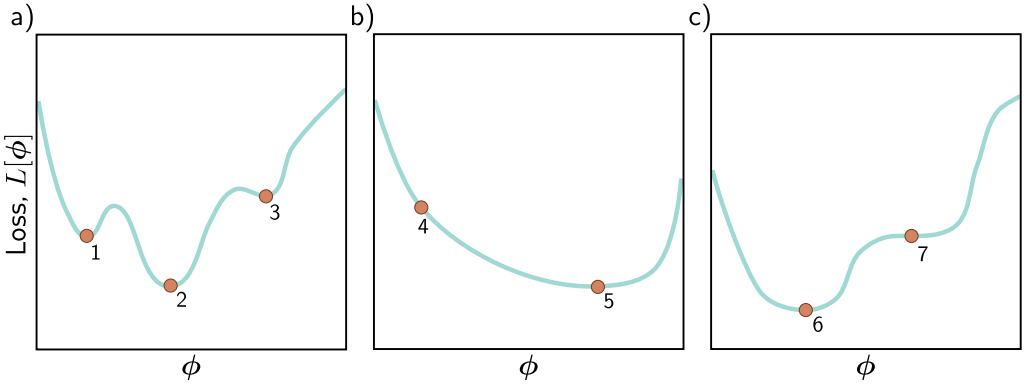

Figure 6.11 Three 1D loss functions for problem 6.6.

(i) Plot y against x for this model for different values of ϕ_0 and ϕ_1 and explain the qualitative meaning of each parameter. (ii) What is a suitable loss function for this model? (iii) Compute the derivatives of this loss function with respect to the parameters. (iv) Generate ten data points from a normal distribution with mean -1 and standard deviation 1 and assign them the label $y = 0$. Generate another ten data points from a normal distribution with mean 1 and standard deviation 1 and assign these the label $y = 1$. Plot the loss as a heatmap in terms of the two parameters ϕ_0 and ϕ_1. (v) Is this loss function convex? How could you prove this?

Problem 6.5* Compute the derivatives of the least squares loss with respect to the ten parameters of the simple neural network model introduced in equation 3.1:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x]. \tag{6.23}$$

Think carefully about what the derivative of the ReLU function $a[\bullet]$ will be.

Problem 6.6 Which of the functions in figure 6.11 is convex? Justify your answer. Characterize each of the points 1–7 as (i) a local minimum, (ii) the global minimum, or (iii) neither.

Problem 6.7* The gradient descent trajectory for path 1 in figure 6.5a oscillates back and forth inefficiently as it moves down the valley toward the minimum. It's also notable that it turns at right angles to the previous direction at each step. Provide a qualitative explanation for these phenomena. Propose a solution that might help prevent this behavior.

Problem 6.8* Can (non-stochastic) gradient descent with a *fixed* learning rate escape local minima?

Problem 6.9 We run the stochastic gradient descent algorithm for 1,000 iterations on a dataset of size 100 with a batch size of 20. For how many epochs did we train the model?

Problem 6.10 Show that the momentum term $\mathbf{m}_t$ (equation 6.11) is an infinite weighted sum of the gradients at the previous iterations and derive an expression for the coefficients (weights) of that sum.

Problem 6.11 What dimensions will the Hessian have if the model has one million parameters?