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

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:

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

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 no chord (line segment between two points on the surface) intersects the function. 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 6.3–6.6

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

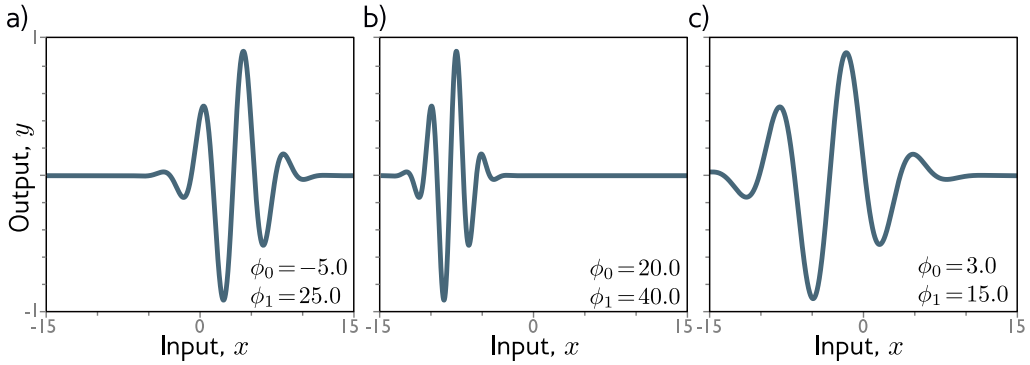


Figure 6.2 Gabor model. This nonlinear model maps scalar input x to scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$. It describes a sinusoidal function that decreases in amplitude with distance from its center. Parameter $\phi_0 \in \mathbb{R}$ determines the position of the 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.

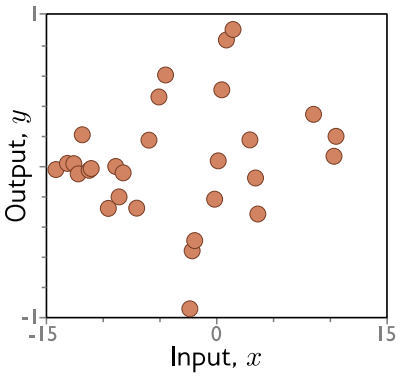


Figure 6.3 Training data for fitting the Gabor model. The training dataset consists of 28 input/output examples $\{x_i, y_i\}$. These were created by uniformly sampling $x_i \in [-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding normally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

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:

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

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

6.1.3 Local minima and saddle points

The loss function associated with the Gabor model for this dataset is depicted in figure 6.4. 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 blue point that is closest to the center of the plot.

Problem 6.4

Problem 6.7

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 (dark green point) 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.

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 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 very flat, and so it's hard to be sure that training hasn't converged; if we terminate our algorithm when the gradient is very small, then we may erroneously stop near a saddle point.

6.2 Stochastic gradient descent

The Gabor model has two parameters, and 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, and 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 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 moving from one "valley" of the loss function to another (figure 6.5b).

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:

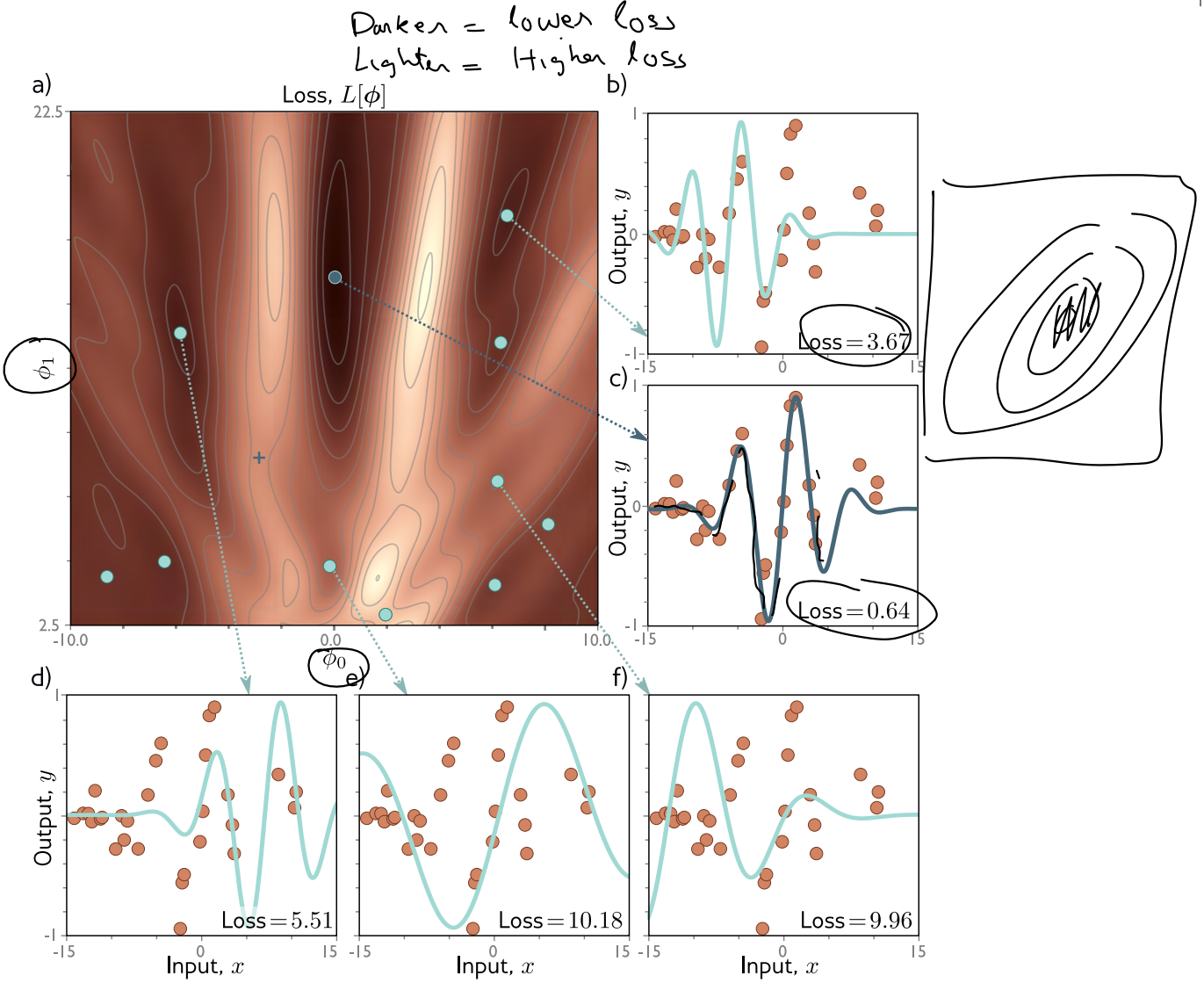


Figure 6.4 Loss function for the Gabor model. a) The loss function is non-convex, with multiple local minima (cyan points) in addition to the global minimum (blue/gray point). It also contains saddle points where the gradient is locally zero, but the function is increasing in one direction and decreasing in the other. The green 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.

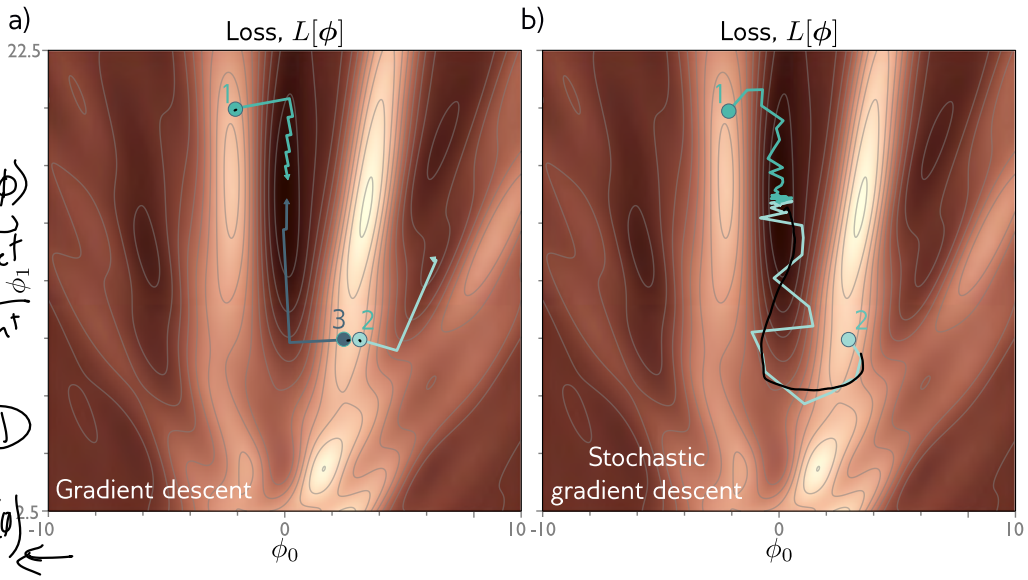
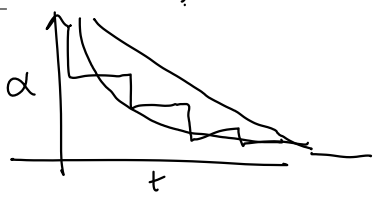
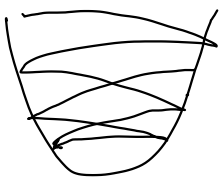


Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized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





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

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 all of entire training dataset is referred to as an *epoch*. A batch may be as small as a single example, or as large the entire dataset. The latter case is referred to as *full-batch gradient descent* and is identical to regular (nonstochastic) gradient descent.

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 so will be different 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 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 are described well 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 relatively 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, and 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 is increased if all these gradients

Problem 6.8

$$m_{t+1} = \beta\left(\beta m_{t-1} + (1-\beta)\sum_{B_{t-1}} \frac{\partial l}{\partial \phi}\right) + (1-\beta)\sum_{B_t} \frac{\partial l}{\partial \phi}$$

$$= \beta^2 m_{t-1} + (1-\beta)\beta \sum_{B_{t-1}} \frac{\partial l}{\partial \phi} + (1-\beta)\sum \frac{\partial l}{\partial \phi}$$

$$= \underbrace{\beta^t m_0} + \beta^{t-1}(1-\beta)\sum_{B_0} \frac{\partial l}{\partial \phi} + \underbrace{\beta^{t-2}}_{(0.9)^{98}}\underbrace{(1-\beta)^2}_{(0.1)^2}\sum_{B_1} \frac{\partial l}{\partial \phi} + \cdots + \underbrace{(1-\beta)}_{0.1}\sum_{B_t} \frac{\partial l}{\partial \phi}$$

$\beta \approx 0.9$

m_t = moving average of past Batch gradients

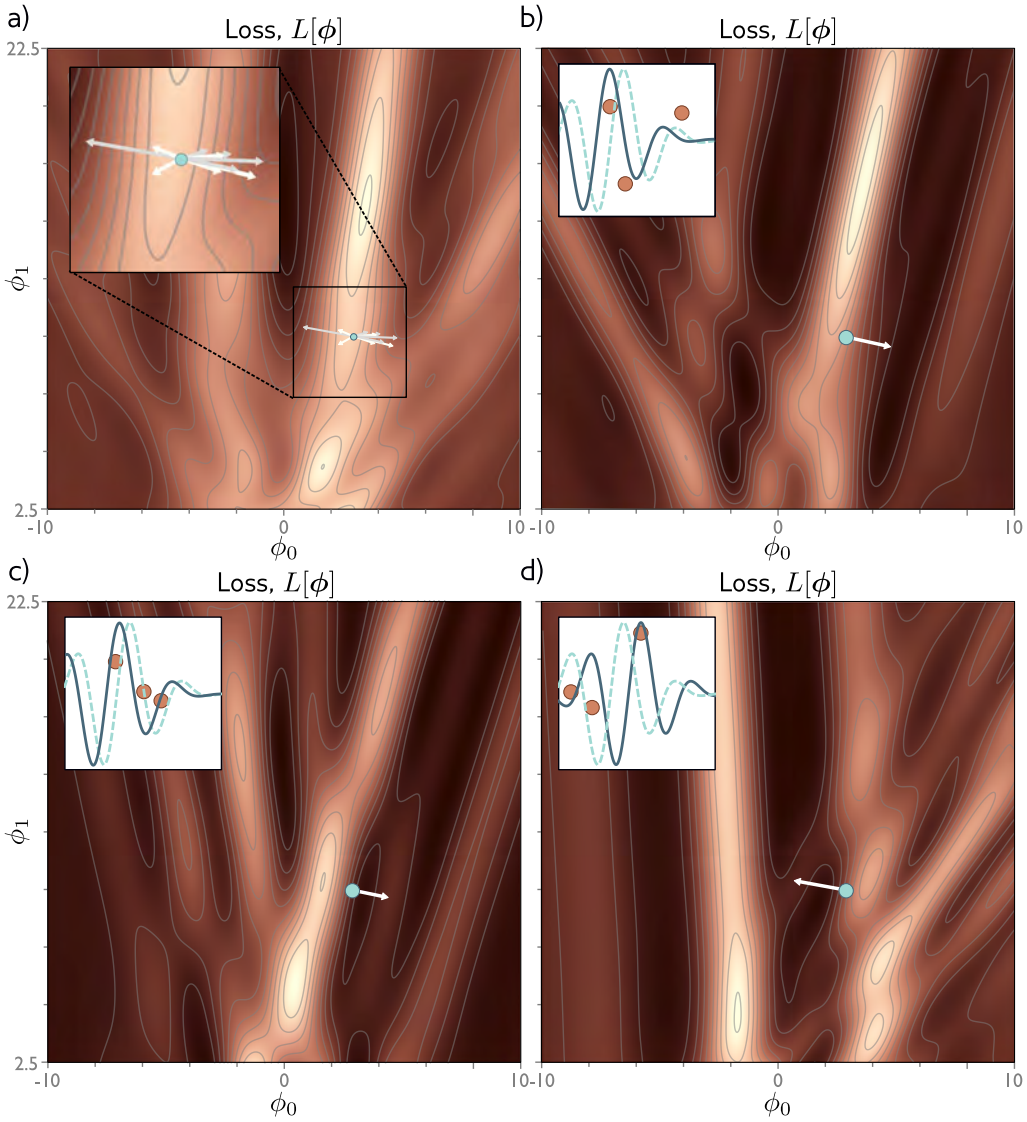


Figure 6.6 Alternative view of SGD for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. 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.

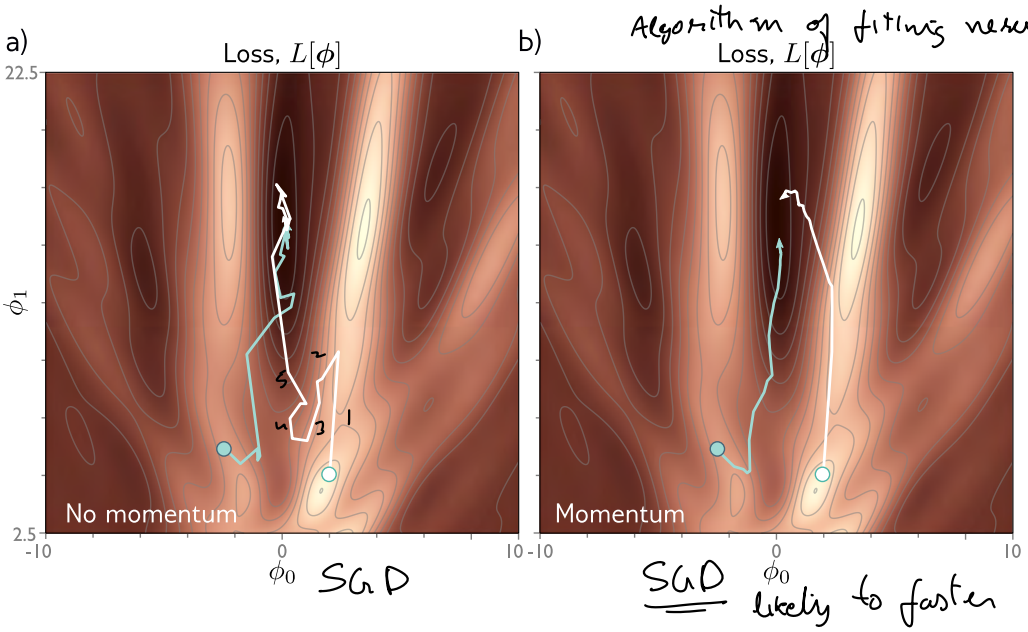


Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory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

are aligned over multiple iterations but decreased 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).

6.3.1 Nesterov accelerated momentum

The momentum term can be thought of as 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:

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

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

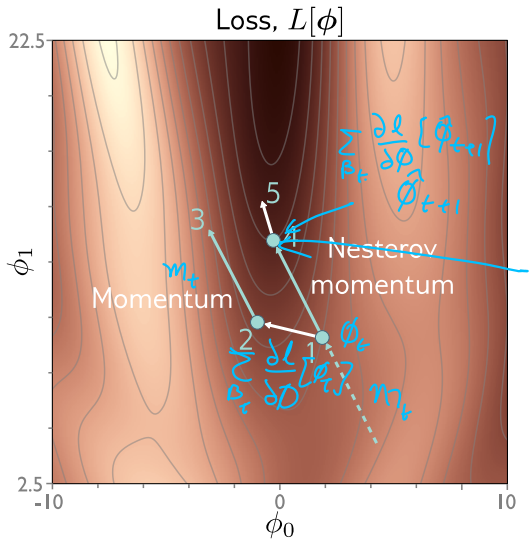


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same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

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 very simple 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 $\mathbf{v}_{t+1}$:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\phi_t]}{\partial \phi} \\ \mathbf{v}_{t+1} &\leftarrow \frac{\partial L[\phi_t]}{\partial \phi}^2 .\end{aligned} \tag{6.13}$$

Then we apply the update rule:

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

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

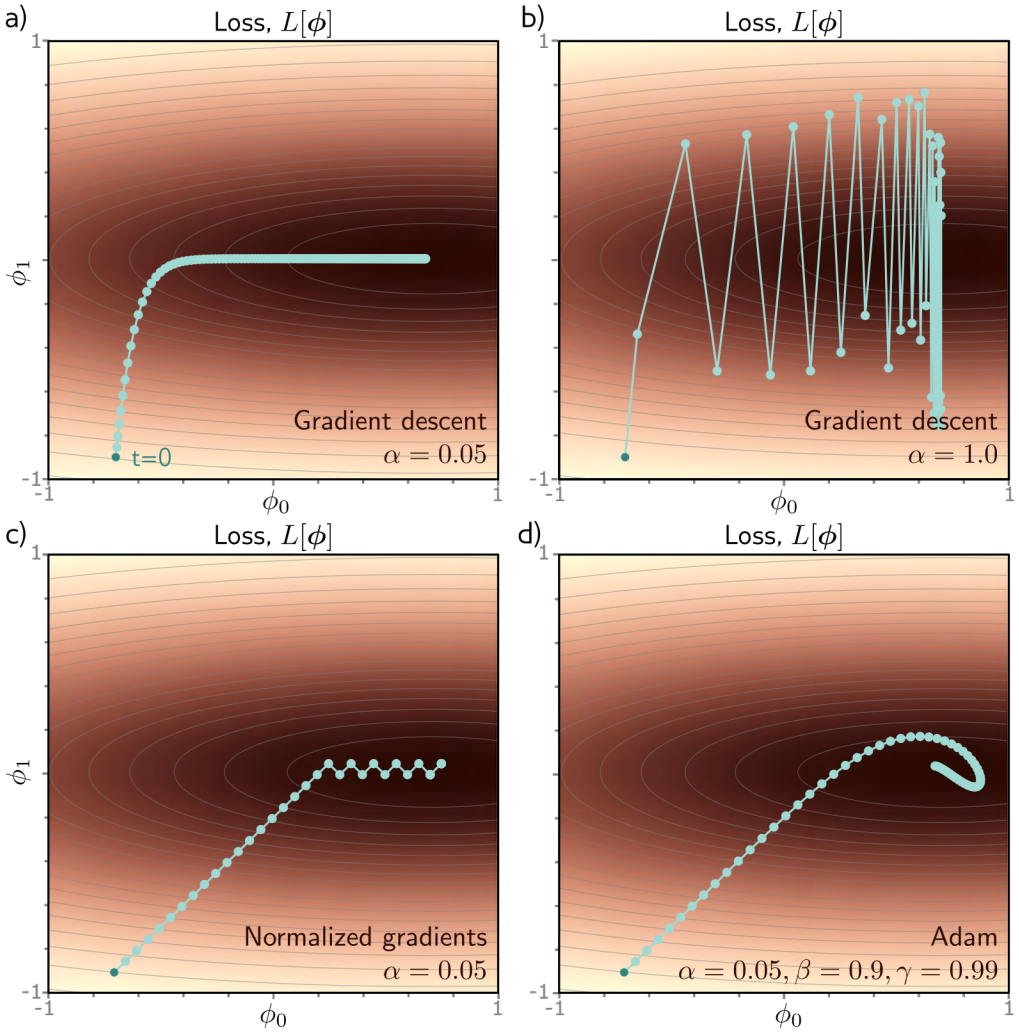


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction, but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, then it overshoots in the vertical direction and becomes unstable. c) A very simple approach is to move a fixed distance along each axis at each step in such a way 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 together create a smoother path.

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:

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

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, and so the denominator becomes closer to one and this modification has a diminishing effect.

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

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

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:

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

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 that it is less sensitive to the initial learning rate, because it avoids situations like those in figures 6.9a–b, and 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 quality of the final model 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 tuning*. We return to this issue in chapter 8.

6.6 Summary

This chapter discussed model training. This problem was framed as finding the 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 the fastest. This is repeated until convergence.

Unfortunately, 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. 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 are applicable to optimizing *any* model. The next chapter tackles two aspects of training that are 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, and this can hinder the training process.

Notes

Optimization algorithms: Optimization algorithms are used extensively throughout engineering although it is 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 algo-